האוניברסיטה העברית בירושלים
The Hebrew University of Jerusalem

Distributed Wireless Sensor Networks (WSNs) Bottleneck Detection

Thesis submitted in fulfillment of the requirement for the degree of Master of Science

By

David Karpf-Cogan

The Rachel and Selim Benin School of Computer Science and Engineering
The Hebrew University of Jerusalem

This work was carried out under the supervision of
Prof. Danny Dolev

September 2010

Abstract

In a world where wireless sensor networks (WSNs) are increasing in importance in diverse fields, such as, industry, military, medical and home facilities. There is a constant effort to improve the network's utilization and extend its life cycle. The detection and handling of the bottlenecks in the WSNs is one area in which a significant effort is being made in order to achieve that goal. Bottlenecks cause an increment in the number of messages *retransmitted* over the WSN and for *throughput* deterioration and packet transmission *latency* to increase. Moreover, in case a sensor that is part of the bottleneck chain dies, some parts of the network would become inaccessible causing the network to be partitioned into several sections. In a network that consists out of low power supplied nodes prone to failure and characterized by random scattering, it is important that we detect the bottleneck nodes to prevent the network being taken out of use.
The customary internet bottleneck detection tools and the latest WSNs detection tools, that I encountered, are not message complexity oriented. Unfortunately, this is an important character due to the extensive energy resources that are wasted in the course of packet transmission in a limited power supplied sensors network For example, the energy cost of transmitting one byte in MSP430 sensor for a range of up to 125 meters is equivalent to 4,000 CPU cycles and for a range of up to 300 meters is equivalent to 32,000 cycles.

This article presents a modification to Wang et al [1] algorithm, which detects the network boundary. Then, it suggests a new way to compute the network medial axis and finally suggests few methods to allocate the bottleneck by using the computed medial axis. The run time complexity of the algorithm is O(n) and the message complexity is bounded by the max(O($n\log(n)$) , O(m^{2})), where m represents the number of nodes constructing the medial axis.

Table of Contents

1 Introduction

The wireless sensor network (WSN) has become the talk of the day in the Distributed Systems Group (DSG) community and in the industry as well. In September 1999 [25], Business Week heralded it as one of the 21 most important technologies for the 21st century. As the sensors become highly available, widely spread, cheap and easy to use and deploy, the day when each of us will encounter sensor networks on daily basis is just a matter of time. The wireless sensor networks today are used for sensing a variety of environment conditions such as temperature, humidity, light and density of air pollutant, for early fire detection, smart homes, military needs and for other varied purposes. The main reasons for its popularity are the low price and the ease to form the network, where in most places it is just “plug and play” in WSN it is “scatter and play”.

The main drawbacks from the WSNs are the resource constraints they impose on us. Unlike the traditional WSN, in most applications of wireless sensor networks, WSNs have limited energy, low storage capacity, and weak computing capability. Many times, due to the sensors accessibility difficulties, the resources, especially the energy of sensors, may not be replaced or recharged. Hence, the lifetime of the WSN highly depends on the energy consumption of sensors. Like the other wireless networks, the position of sensor nodes has great impact on the performance of WSNs in terms of coverage, communication cost, network’s lifetime and resource management. Due to the randomness of sensors deployment, there might exist some nodes connecting two or more partitions without any backup nodes. This causes a delay of message propagation between the partitions. It might also cause a message loss and in the case when one of the nodes dies out, the whole network will be partitioned. These nodes are called bottleneck nodes, and it is important to locate the bottleneck nodes in order to prevent network partitioning or message transmission delay, that will lead to retransmission of messages in the future. A more formal definition of bottleneck can be found in Daganzo[18]: An active bottleneck is defined by an upstream queue and unrestricted flows present on downstream sections . According to Zhang and Levinson [19], three main traffic characteristics are present at bottlenecks: flow drop, queue discharge flow, and the pre-queue transition period.

The need for a new sensor bottleneck detection method arose, since the previous methods were not message transmission efficient algorithms, a crucial characteristic for low energy source devices, since the transmission energy cost for one bit equals to thousands of sensor's CPU cycles. The first two Internet bottleneck detection tools [20, 21] presented in the next section are so costly in message transmission aspects that they require ad hoc servers and routers in order to execute properly. The WSNs designated tools are much more efficient in that matter, but their complexity remains in the polynomial region, where the MINCUT [22] algorithm complexity is O(n^3) and the DBND [23] complexity is bounded by O(n^2). This paper will present an algorithm that its run time complexity is O(n) and the message complexity is bounded by the max(O(*nlog(n)*) , O($m^{2`}$)), where *m* represents the number of nodes constructing the medial axis.

The algorithm presented in this paper provides a new approach for network bottleneck discovery, where it suggests using the network medial axis for the bottleneck discovery. Furthermore, the technique used for the medial axis calculation and 'noises' removal is also unique. One of the first steps executed in order to discover the medial axis is the network boundary detection which is based on the boundary detection algorithm presented by Wang et al [1]. Modifications were also made at the Wang algorithm in order to execute in fully distributed way and the hops count were replaced by RSSI (Received Signal Strength Indication) measurements, which are much more accurate for sensors location indication. The rest of the article is divided as follows: Detailed related work in section 3, the proposed algorithm in section 4 and summary and future work in section 5.

2 Related work

Bottleneck nodes have been concerned and studied extensively in Internet and traditional wireless networks fields except WSNs. For Internet and traditional wireless networks, the goal of concern is to avoid the congestion and improve the capacity of data flow, resulting in the enhancement of network throughput. However, in WSNs, due to the different network properties, the most important objective is how to save and balance the energy consumption to prolong the lifetime of the whole network. So the related work for other networks may not be suited for WSNs.

In this section we survey two of the latest internet bottlenecks detection tools, BFind [20] and PathNeck [21], then we will present a centralized sensor network bottleneck detection tool based on min-cut algorithm [22] and we will conclude with the distributed sensor network bottleneck detection algorithm DBND [23] from 2010.

2.1 BFind

Akella et al [20] present bottleneck detection tool named BFind. Its design is motivated by TCP's property of gradually filling up the available capacity based on feedback from the network. First, BFind obtains the propagation delay of each hop to the destination. For each hop along the path, the minimum of the (nonnegative) measured delays along the hop is used as an estimate for the propagation delay on the hop. The minimum is taken over delay samples from five traceroutes. After this step, BFind starts a process that sends UDP traffic at a low sending rate (2 Mbps) to the destination. A trace process also starts running concurrently with the UDP process. The trace process repeatedly runs traceroutes to the destination. The hop-by-hop delays obtained by each of these traceroutes are combined with the raw propagation delay information (computed initially) to obtain rough estimates of the queue lengths on the path. The trace process concludes that the queue on a particular hop is potentially increasing if across three consecutive measurements, the queuing delay on the hop is at least as large as the maximum of 5ms and 20% of the raw propagation delay on the hop. This information, computed for each hop by the trace process, is constantly accessible to the UDP process. The UDP process uses this information (at the completion of each traceroute) to adjust its sending rate as described below. If the feedback from the trace process indicates no increase in the queues along any hop, the UDP process increases its rate by 200 Kbps (the rate change occurs once per feedback event, i.e., per traceroute). Essentially, BFind emulates the increase behavior of TCP, albeit more aggressively, while probing for available bandwidth. If, on the other hand, the trace process reports an increased delay on any hop(s), BFind flags the hop as being a potential bottleneck and the traceroutes continue monitoring the queues. In addition, the UDP process keeps the sending rate steady at the current value until one of the following things happen: (1) The hop continues to be flagged by BFind over *consecutive* measurements by the trace process and a threshold number (i.e. 15) of such observations are made for the hop. (2) The hop has been flagged a threshold number of times in total (i.e. 50). (3) BFind has run for a pre-defined maximum amount of total time (180 seconds). (4) The trace process reports that there is no queue build-up on any hop implying that the increasing queues were only a transient occurrence. In the first two cases, BFind quits and identifies the hop responsible for the tool quitting as being the bottleneck. In the third case, BFind quits without providing any reliable conclusion about bottlenecks along the path. In the fourth case, BFind continues to increase its sending rate at a steady pace in search of the bottleneck. If the trace process observes that the queues on the first three hops from the source are building, it quits immediately without providing any conclusion about bottlenecks, to avoid flooding the local network. That is due to the fact, that the first three hops almost always encompass all links along the path that belong to the source stub network.

2.2 PathNeck

Pathneck is an active probing tool presented by Ningning et al [21] that allows end users to efficiently and accurately locate the bottleneck link on an Internet path. Pathneck is based on a novel probing technique called Recursive Packet Train (RPT), which combines load and measurement packets. The load packets are UDP packets that are used to interact with background traffic and to obtain available bandwidth information. The measurement packets are organized as a packet train, which precede and succeed the load packets as shown in Figure 1. They are 60- byte UDP packets with the TTL fields set in such a way that at each hop along the path, the measurement packet at both the head and tail of the train will expire. This will trigger the transmission of two ICMP error packets to the source. The inter-arrival time (called the “gap value”) of the ICMP packets at the source can be used as an estimate of the packet train length at the router that generated the two ICMP packets. The resulting sequence of packet train lengths at each hop can be used to identify the hop that limits the available bandwidth on the path. Hops where the packet train length increases have an available bandwidth that is lower than the packet transmission rate at that hop—we will call these hops *choke points*. The downstream link of a choke point is called a *choke link*. The last choke link is the *bottleneck link*. In practice, queuing effects on both the forward and reverse paths and ICMP packet generation times can introduce noise in the train length measurements. To deal with this, Pathneck sends *n* consecutive RPTs (e.g., *n* = 10), called a probing set, and “averages” across these n probes. Only if a link repeatedly (e.g., more than half the probes) creates a significant increase in the train length (e.g., more than 10%) is it considered to be a valid choke point. This requirement is the main reason that Pathneck sometimes cannot identify a bottleneck. The last choke point on the path is typically the link with the lowest available bandwidth, i.e., the bottleneck.

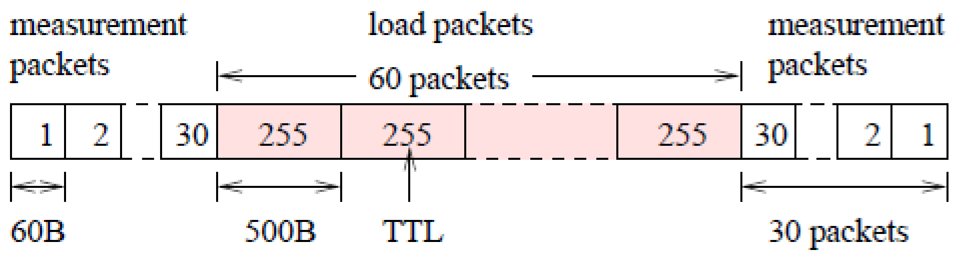


Figure 1: Recursive Packet Train (RPT).

2.3 Max-Flow MinCut algorithm

The maximum flow problem is intimately related to the minimum cut problem. A *cut* is ant set of directed arcs containing at least one arc in every path from the origin node to the destination node. In other words, if the arcs in the cut are removed, then flow from the origin to the destination is completely cut off. The *cut value* is the sum of the flow capacities in the origin-to-destination direction over all of the arcs in the cut. The minimum cut problem is to find the cut that has the minimum cut value over all possible cuts in the network. In order to find the minimum cut we will make use of the *max-flow / min-cut theorem*: for any network having a single origin node and a single destination node, the maximum possible flow from origin to destination equals cut value for all cuts in the network. To understand the relation between the max-flow and the minimum cut we can see that the maximum flow through a series of linked pipes equals the maximum flow in the smallest pipe in the series, i.e. the flow is limited by the bottleneck pipe. The minimum cut is just a kind of *distributed* bottleneck, a bottleneck for a whole network as opposed to a simple bottleneck for a series of pipes.

For WSNs, an algorithm called MINCUT was proposed by Mechthild et al [22]. In MINCUT algorithm, the course of execution can be divided into two phases: (a) Global information collection. All the nodes in the network have to broadcast their position information to sink node, this course can be as messages flooding. (b) After getting the global information, the sink node will compute and find the bottleneck node using MINCUT algorithm. The MINCUT algorithm is computed as follows:

```
MINIMUM_CUT_PHASE(G, w, a)
A ← {a}
while A ≠ V
add to A the most tightly connected vertex
store the cut-of-the-phase and shrink G by merging the two vertices added last
```

```
MINIMUM_CUT(G, w, a)
while |V| > 1
MINIMUM_CUT_PHASE(G, w, a)
```

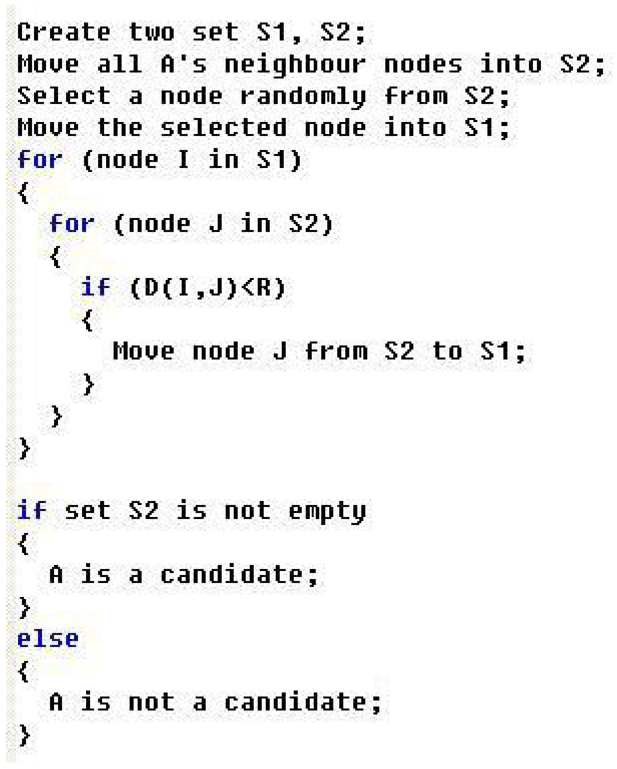
if the cut-of-the-phase is lighter than the current minimum cut then store the cut-of-the-phase as the current minimum cut

2.4 DBND (Distributed Bottleneck Node Detection) algorithm

The implementation of the DBDN algorithm, presented by Gou et al [23], can be divided into three phases. These include: (i) neighbor location information collection, (ii) bottleneck node candidate selection and (iii) bottleneck node confirmation as follows:

(i) *Neighbor location information collection* - In the neighbor location information collection phase, each node has to send its location information to its neighbors and finally, each node can acquire all its neighbor location information. However, so many nodes send packets simultaneously and each node has to switch between sending mode and receiving mode repeatedly, causing the collision rate to be very high. Therefore, an efficient way for the information collection is by dividing the collection phase into rounds and in each round a limited set of nodes is defined as transmitters. The following equation is taken from LEACH algorithm [24]: Each node is assigned with a random value between 0-1 in each round, and if this value is less than a calculated threshold, the node becomes a transmitter. The communication for each round includes the two phases: transmitter election and location information broadcasting.

(ii) *Bottleneck node candidate selection* - After neighbor location information collection, each node forms a neighborhood set and then performs the algorithm described in Figure 2 and decides whether it can be a candidate bottleneck node.



```
Create two set S1, S2;
Move all A's neighbour nodes into S2;
Select a node randomly from S2;
Move the selected node into S1;
for (node I in S1)
{
  for (node J in S2)
  {
    if (D(I,J)<R)
    {
      Move node J from S2 to S1;
    }
  }
}

if set S2 is not empty
{
  A is a candidate;
}
else
{
  A is not a candidate;
}
```

Figure 2: Process of candidate collection.

The logic that stands behind the algorithm is that if at the end of the run S2 is not empty, that means, the nodes at partition S1that wish to communicate with nodes at S2 has to, with high probability, pass through node A, forming node A, a bottleneck candidate.

(iii) *Bottleneck node confirmation* - After the candidate selection, some nodes out of the candidates are confirmed as the bottleneck nodes. If node A, as shown in Figure 3, is the only relay connecting the different neighbor partitions, A is a bottleneck node. On the other hand, if there is another area D, which can also connect the two different partitions, node A is not a bottleneck node. In this phase, the actual bottleneck nodes can be selected out of the candidate nodes with further confirmation. The confirmation process can be divided into two steps: (a) First, after the candidate node selection, each candidate will notify its neighbors. As shown in Figure 3, A is selected as a candidate, and it notifies all its neighbors in area B {B1, B2, B3, B4} and area C {C1, C2, C3}. (b) Second, the neighbor nodes will search the other possible ways to connect the different partitions, such as path B-D- C. In this step, one node in area B will be selected randomly to send a confirmation message to a random node in area C. Within a certain time threshold T, if the given node in area C receives the message, node A cannot be a bottleneck node. However, if it does not receive the message, node A is a bottleneck node. In order to reduce the hops and time of message transmission, the following rule will be used for relay node selection: (a) For the message transmission, the source node will form a set of its neighbor nodes, such as {N1, N2, N3, …}, and it computes the distance between every neighbor node and the given destination node according to the known position, forming a distance set {d1,d2, d3, …}. (b) The source node will choose the closest node to the destination as the next-hop node. (c) After the next-hop node receives the message, it repeats the steps (a), (b) till the message arrives at the destination node.

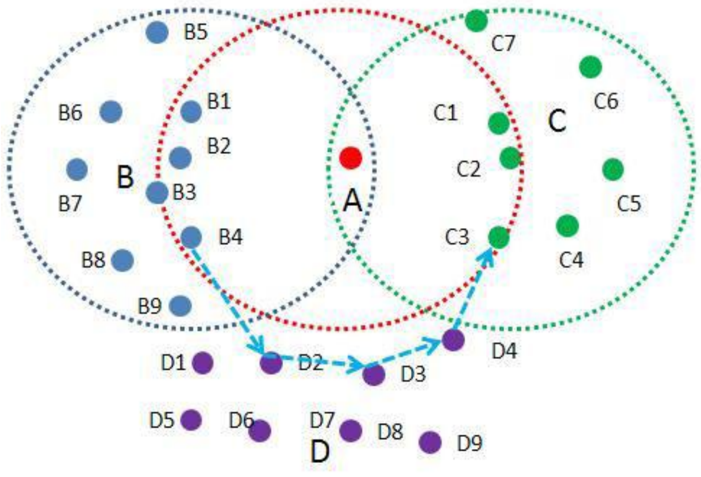


Figure 3: Process of bottleneck node confirmation.

3 Proposed algorithm

In this paper, a light-weight distributed bottleneck detection tool is provided, that places emphasis on the number of messages sent during that process. This section will deal with the presented solution and discuss in detail the different steps in the algorithm, starting with the network inner and outer boundaries detection and numbering, continues with finding the medial axis and concludes with network bottleneck detection according to the sensor network characteristics.

3.1 Find the network inner and outer boundaries

In order to find the network inner holes and outer boundaries, the following steps are proposed: elect a leader for the network, build a shortest path tree with the leader as the tree root, find cuts in the tree, detect the coarse inner boundary, find extremal nodes on the boundaries, find the outer boundary and refine the coarse inner boundary.

The following assumptions regarding this proposal are being made.

Assumption 1: It is deduced that the sensor field topology problem that was previously discussed will result in a simple geometric shape. Meaning that if the edge nodes in the network are surrounded, then a geometric shape consisting of one cycle where each node on the boundary has exactly one neighbor on each side will be formed.

Assumption 2: The closest neighboring boundary node to any other boundary node lies on the same boundary side.

3.1.1 Leader election in a multi-hop network

A network leader will be elected in order to span a shortest path tree with the leader as the network root. A brief summary of the algorithm that was described and proven in Cidon et al [12] will be provided.

3.1.1.1 The Propagation of Information with Feedback (PIF) algorithm

A message in this algorithm is of the form MSG (*target*, *l*, *parent*) where *target* specifies the target node or nodes. A null value in the target header field indicates a broadcast to all neighboring nodes and is used when broadcasting the message. The *parent* field specifies the identity of the parent of the node that sends the message. It is important to note that a node receives a message only when addressed in the *target* header field by its identification number or when this field is null. The *l* Field determines the sender's identity. The initiator called the source node broadcasts a message, thus starting the propagation. Each node upon receiving the message for the first time stores the identity of the sender from which it got the message which originated at the source and broadcasts the message. The feedback process starts at the leaf nodes which are childless nodes on the virtual tree spanned by the PIF algorithm A leaf node that is participating in the propagation from the source and has received the message from all of its neighboring nodes sends back an acknowledgment message called a feedback message which is directed to its parent node. A node that got feedback messages from all of its child nodes and has received the broadcasted message from all of its neighboring nodes sends the feedback message to its parent. The algorithm terminates when the source node gets broadcast messages from all of its neighboring nodes and feedback messages from all of its child nodes.

3.1.1.2 *Algorithm for a fragment in the graph*

A high level algorithm that operates at the fragment level is first presented here. In the next section, a distributed algorithm for the entire network based on the one fragment algorithm will be presented.

During the operation of the algorithm the nodes are partitioned into fragments. Each fragment is a collection of nodes consisting of a candidate node and its domain of supportive nodes. When the algorithm starts all the candidates in the graph are active. During the course of the algorithm a candidate may become inactive in which case its fragment joins an active candidate's fragment and no longer exists as an independent fragment. The algorithm terminates when there is only one candidate in the graph and its domain includes all of the nodes.

We define for each fragment an *identity* denoted by *id(F)* and a *state.* The *identity* of a fragment consists of the size of the fragment denoted by *id(F).size* and the candidate's identification number denoted by *id(F).identity*. The state of the fragment is work, *wait* or *leader*. We associate two variables with each edge in the graph its current state and its current direction. An edge can be either in the state internal in which case it connects two nodes that belong to the same fragment or in the state external when it connects two nodes that belong to different fragments. External edges will be directed in the following manner: Let *e* be an external edge that connects two different fragments F and F'. The algorithm follows these definitions for directing an edge in the graph:

Definition 1: The relation *id*(F1) > *id* (F2) holds
if *id*(F1).size > *id*(F2).size or if [*id*(F1).size = *id*(F2).size and *id*(F1).identity > *id*(F2).identity].

Definition 2: Let *e* be the directed edge (F1, F2) if *id*(F1) > *id*(F2) as defined in definition 1.

The edge *e* is considered an *outgoing* edge for fragment F1 and an *incoming* edge for fragment F2, in case the relation above holds. In addition, F1 and F2 are considered neighboring fragments.

When the algorithm starts each node is an active candidate with a fragment size of 1. The algorithm is described for every fragment in the graph by a state machine. A fragment may be in one of the following states *wait*, *work* or *leader*. A Fragment is in the virtual state *cease* to exist when it joins another candidate's fragment. The initial state for all fragments is *wait* and the algorithm terminates when there is a fragment in the *leader* state.

The State Machine Formal Description

1. A fragment F enters the wait state when it has at least one outgoing edge.
2. A Fragment F transfers to the *work* state from the wait state when all its external edges are *incoming* edges. In the work state it performs the following:
    a. Count the new number of nodes in its current domain. The new size is kept in the variable *new_size*.
    b. Compare its *new_size* to the size of its maximal neighbor fragment, F'.
        i. If the *new_size*(F) > X**id*(F').*size* then fragment F remains active., where X >1.
        Let *id*(F).*size* $\leftarrow$ *new_size*. F changes all of its external edges to the *outgoing* state. Clearly, definition 2 holds here and at this step, all of its neighbors become aware of its new size and F transfers to the *wait* state.
        ii. Else, fragment F ceases being an active fragment and becomes a part of its maximal neighbor fragment F'. External edges between F and F' will become internal edges of F'. F' does not change its *size* or *id*, but may have new external edges, which connect it through the former fragment F to other fragments. The new external edges' state and direction are calculated according to the

current size of F’. It is clear that all of them will be *outgoing* edges at this stage.

3. A fragment that has no external edges is in the *leader* state.

3.1.1.3 The distributed algorithm for nodes in the graph

The algorithm begins with an initialization phase in which every node is considered a fragment consists out of one node. Every node in this phase broadcasts its identity. During the course of the algorithm, a fragment that all of its edge nodes have heard PIF messages from all their neighbors enters state *work.* The fragment’s nodes report back to the *candidate* node the number of nodes in the fragment and the identity of the maximal neighbor. During the report the nodes also store a path to the edge node which is adjacent to the maximal neighbor. The *candidate* node, at this stage also the *source* node, compares the newly counted fragment size to the maximal known neighbor fragment size. If the newly counted size is not at least X times bigger than the size of the maximal neighbor fragment then the fragment becomes inactive and joins its maximal neighbor. It is done in the following manner: The candidate node sends a message on the stored path to the fragment’s edge node. This edge node becomes the fragment’s *source* node. It broadcasts the new fragment *identity*, which is the joined fragment *identity*. At this stage, neighboring nodes of the joined fragment disregard this message, thus the maximal neighbor fragment will not enter state *work.* The source node chooses one of its maximal fragment neighbor nodes as its parent node and later on will report to that node. From this point, the joining fragment broadcasts the new *identity* to all other neighbors. In case the newly counted size was at least X times that of the maximal neighboring fragment, the candidate updates the fragment’s identity accordingly and broadcasts it. Note, this is actually the beginning of a new fragment PIF cycle. The algorithm terminates when a candidate node learns that it has no neighbors.

3.1.1.4 Leader election complexity

The presented leader election algorithm runs in *O(n)* time and *O(n log(n)*) message transmission complexity. Proof can be found in Cidon et al [12].

3.1.2 Build a shortest path tree

The goal is to build a shortest path tree, by measuring the Received Signal Strength Indication (RSSI) values, where the network leader functions as the tree root.
The root initiates the propagation process by sending a message containing the accumulative RSSI (default set to zero) and the direct parent node ID (default set to the leader ID).

Each node receiving a message calculates the RSSI by adding the RSSI of the message with the accumulative RSSI stored in the message field. If this is the first message the node received or the new RSSI is smaller than any RSSI the node had seen thus far, the node performs the following two actions: 1) stores the new RSSI and the direct parent as the new node's parent and 2) broadcasts the message. Before the node broadcasts a message it stores its ID in the parent filed and the new RSSI in the accumulative RSSI field. Otherwise, if the node has a smaller RSSI value stored than the one received in the packet, the node discards the message.

Lemma 1: After O(*d*) time, where *d* is the network diameter and O(n) message sent, we have a shortest path tree rooted at the network leader.

Proof: Every node at stage *X* broadcasts the build construction message to all its neighbors. At stage *X+1* all nodes receiving the messages select their parent by choosing the closest node (by comparing the received RSSI) to be their direct parent. It is impossible that a node will choose a parent at stage *X+1* and change his election at a later stage. This is due to the fact that if nodes *b* and *c* choose node *a* as there parent, there will be no shorter path to *b* that goes through *c*. This can be proven by using the triangle inequality, which states that for any triangle, the sum of the lengths of any two sides must be greater than the length of the remaining side, e.g. in a triangle ABC: $C < A + B$ for every edges A,B and C. Therefore, each node would broadcast a message only once, since after decision of a direct parent on an arbitrary cycle, it would not receive messages with a smaller RSSI. Hence, the message complexity is O(n) messages. For the run time, the complexity is O(d), where d is the diameter, since the run time is derived from the number cycles executed until the tree is constructed and this equals to the tree depth.

3.1.3 Find cuts in the shortest path tree

In order to find cuts in the tree, we will execute the following steps: (i) use Alstrup et al [2] algorithm to label the nodes. (ii) Each node then will query its neighbors for their nearest common ancestor. (iii) Execute a test to verify whether a cut exists.
After a cut is detected, the nodes in the cut will connect themselves into connected components. Furthermore, each connected component agrees on the node closest to the network root (ties are broken by the smaller ID). Note, the distance from the root is well known to all nodes in the network, since we saved the accumulative RSSI from each node to the root during the construction of the shortest path tree.

3.1.3.1 Network cuts definition

Intuitively, we define a network cut as neighboring nodes pair, (p, q), in the network topology whose Least Common Ancestor, LCA(p, q), in the shortest path tree T is *far* from p and q, with *well separated* paths to the LCA(p, q).
We will give a formal definition:

Definition 3: A cut pair (p, q) is a pair of neighboring nodes in the network satisfying the following conditions: (i) The distance between p and q to LCA(p, q), y, is above a threshold δ_1. (ii) No node on the path (p-y) from p to y has a direct neighbor on the path (q-y) from q to y except the lower and upper δ_2 percent, that is the δ_2 nodes closest to the cut pair and the δ_2 nodes closest to the LCA(p,q).

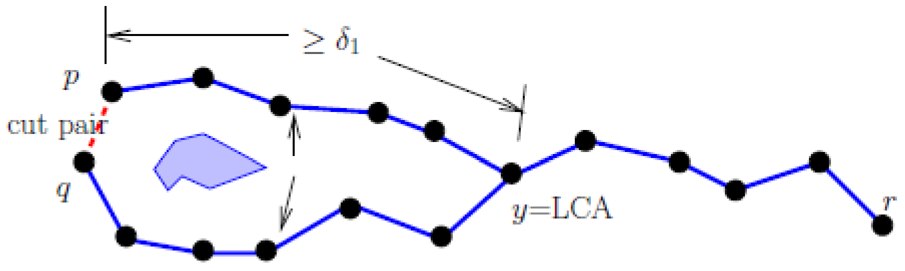


Figure 4: Definition of a cut pair (p,q).

The parameter δ_1 in the definition of a cut pair specify the minimum size of the holes to be detected, where the second condition assures us that the two paths are well separated. The δ_2 parameter allows one to detect among others, holes that spread out linearly, meaning, holes that start narrow and gradually become wider. Typically, one chooses δ_1 as some constant fraction of the diameter *d* of the sensor field. By triangle inequality, we learn that the diameter of the network is at most 2*d*, where the diameter *d* is the distance between the root *r* and the deepest node. In Wang et al [1] it can be seen that experiments reported to use $\delta_1 = 0.18d$. Intuitively, we will choose δ_2 as a fraction of δ_1, in our algorithm we chose $\delta_2 = 0.2\delta_1$, suggesting the majority of the hole to be well separated (emitting the lower and upper 20%, results with 60% well separated).

In order to keep the algorithm fully distributed the network root can disseminate δ_1 and δ_2 through a message in the tree after the shortest path tree is built and hence the network diameter is known.

3.1.3.2 Algorithm to find the network cuts

In order to find the cuts in the network one must first find the LCA of every two neighboring nodes and then check whether we can verify the two conditions declared in definition 3.

3.1.3.2.1 Find the LCA

Alstrup et al [2] gave a distributed algorithm to compute the LCA. The idea is to label the nodes of a rooted tree such that from the labels of two nodes alone one can compute in **constant** time the label of their nearest common ancestor by bit manipulations. The labels assigned by the algorithm are of size $O(log(n))$ bits where n is the number of nodes in the tree. The network labeling algorithm run and the message complexity are $O(n)$.

3.1.3.2.2 Verify distance to LCA

After assigning a label to each node, the nodes will be checked in order to satisfy the first condition of definition 3: each node finds the nearest common ancestor he shares with each of his neighbors. Finding the LCA between two nodes takes a constant time O(c) as we mentioned above, therefore, the message complexity and run time is O(k), where k is the connectivity level.

Since the results for the LCAs are nodes labels and identify nodes in the network by their IDs, there are one of two options: (i) Each node can store, during the labeling process a "translation table", which holds three values: the node's label, node's ID and the distance from the node, for better results in terms of message complexity and algorithm run time,. That way, two neighbors can check whether they fulfill the first condition in definition 3 in O(c) time, since the bit manipulation carried in order to get the LCA's label is constant and so is the translation to distance from LCA and the comparison to $\delta_{1.}$ (ii) In case one does not wish to create translation tables, an upward tree traversal after receiving the

LCA's label and accumulate the RSSI until it reaches the LCA is committed. Then, one can verify the first condition of the definition: whether the distance from the initiating nodes to the LCA exceeds $\delta_{1.}$

3.1.3.2.3 Verify the paths are "well separated"

The verification of the second condition, which confirms that there are no neighbors between the two paths, except for the upper and bottom δ_2 percent is done, iff the first condition was satisfied. In order to do so, both nodes of the cut pair (p,q) flood a message with their ID and *TTL* = 1. The direct parent in the path receiving the message broadcasts the received message after setting the *TTL* to 1 again and accumulates the RSSI in another field. The message would be passed upwards until we reach the LCA(p,q), which stops the flooding and sends back PIF message to the cut pair after receiving the messages from both paths initiated at p and q. If during that process there is a node on one of the paths that received both messages (one with the ID of p and another with the ID of q) and he is not in the upper or bottom δ_2 percent of the path: it sends a message back to the cut pair (p,q). When the cut pair receives a message from the LCA, it means that all nodes on the paths had received the messages. Hence, if the cut pair received a message from the LCA only and from no other nodes on the path, he concludes that the second condition had fulfilled and therefore, there is a cut.

In order for the nodes to know whether they are on the bottom or upper δ_2 percent of the path, the following will be done: when the cut pair (p and q) broadcast the message to their direct neighbors (TTL=1), each parent adds the RSSI of the received message to the accumulative RSSI. That way, each node receiving a message can calculate immediately whether he is in the closest δ_2 percent to the cut pair. For the closest δ_2 percent to the LCA, each node can calculate by one of the following ways: (i) Each node knows its distance from then LCA based on previous steps and therefore can calculate whether he is in δ_2 distance from the LCA. (ii) When the message returns from the LCA back to the initiating *p* and *q* nodes through the two paths, the RSSI from the LCA can be calculated

and the nodes which satisfy the second condition and are in a distance greater than δ_2 from the LCA will send a reject message.
In either way, the message propagates till the LCA and back, therefore the time and message complexity of that step is equal to the distance between the cut pair and their LCA, which is upper bounded by the network diameter *d*. That is for every hole in the network, hence, bounded by O(*h*d*), where *h* represents the number of holes in the network. Therefore, the total time and messages complexity of this step is *min(*O(*h*d*) , O*(n)).*

Lemma 2: If there are neighbors across the two paths, (p-y) and (q-y), they will be found by the above algorithm.

Proof: If it is assumed that exists a node p_1 on the path p-y that has a neighbor node q_1 on the path q-y. At the one hand, we know for sure, that all nodes on the same path will receive the message originating at the corresponding cut node, since the message is propagated upward from the beginning of the path until it reaches the LCA by traversing from descendent to its parent. In particular, node p_1 that is on the path p-y will receive the message. On the other hand, since the flooding process starting at p and q at roughly the same time, causing each node on the path to broadcast a message to all its one hop neighbors, node q_1 on the q-y path, will broadcast a message that p_1 will receive. Since, the LCA sends the PIF message back to the cut pair, only after receiving the broadcast messages from both paths, meaning all nodes at the two paths received, broadcasted and transferred the message upward, it is inevitable that node p_1 already sent node p a message indicating he has a direct neighbor on the parallel path (that is, if p_1 is not on one of the edges of the path defined by δ_2).

3.1.3.3 Multi hole case

When there are multiple holes and therefore multiple cuts in the network, one would artificially merge the holes by removing nodes on cut branches, until there is only one composite hole left. It would be necessary to remove all of the nodes on cut branches except the one branch furthest away from the root. The interior holes either connect to themselves or connect to the outer boundary. This is done in the following way: Each cut branch, once decided on a “leader” (the closest node to root) floods the network with the leader ID and distance from root. Each node in a cut branch that receives a message with a leader farther away from root than his leader, virtually removes himself from the network. This can be done by enabling a dead/alive bit. The outcome is, cut branches closer to the network boundaries will merge with the boundaries and inner cuts will merge with other interior holes, until there is one remaining hole. Thus, the multi-hole case is turned into a single hole scenario, that we proved its correctness earlier. The deleted nodes would be restored and the composite hole would be refined later on.

3.1.3.4 Single hole case

In case there are no internal holes in the network, an arbitrary node would be chosen as the inner boundary and can skip the next step of the algorithm (the detection of coarse inner boundary). Since the shortest path tree will be built from the inner boundary to all other nodes in the next step of the algorithm, the network root node *r* would be chosen as the inner boundary R for performance efficiency. That is, because the network is aware of the root selection, thus no need to select an arbitrary node and inform the network. Moreover, the shortest path tree to all nodes in the network rooted at *r*, has already been created.
Alternatively, the values of δ_1 and δ_2 can be decreased until artificial holes can be found and continue the algorithm from there.

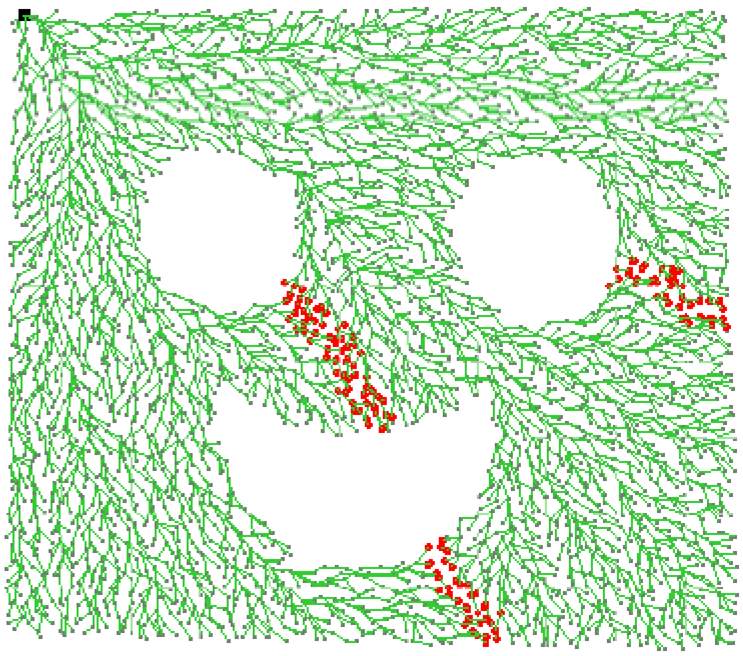


Figure 5: The cut pairs (in red) in a multi hole example.

3.1.4 Detect a coarse inner boundary

After the cut nodes have been detected, a coarse inner boundary R would have to be located that encloses the (composite) interior hole. A coarse inner boundary R is a shortest cycle enclosing the interior hole in the sensor field. Therefore, any multi-hole sensor field has been turned into a single composite hole sensor field by removing all cut branches except one and all the nodes in the remaining cut branch have the ID of the node closest to the root. Thus, the closest node to root will find the shortest path to its neighbor in the cut pair that does not go through any cut node. This can be implemented in two different ways:

(i) Use the shortest path algorithm to find this path, while removing all the edges between cut pairs in order to prevent the path from going through any cut nodes. We presented in previous sections that building a shortest path tree is upper bounded by O(*n*) messages.

(ii) Alternatively, the two shortest paths from p and q to LCA(p, q) that have already been obtained in previous steps can be used. Together with the edge pq, a cycle has been obtained that encloses the hole. This cycle is not necessarily the shortest cycle. But it can be greedily shrink to be as tight as possible, by the following k-distance shrinking process. For any two nodes that are within k distance on the cycle, one checks whether exists a shorter path between them. If so, the corresponding segment is replaced in the original cycle with the shorter path and by that the total length has been shorten. This step is repeated until no further replacements can be done.

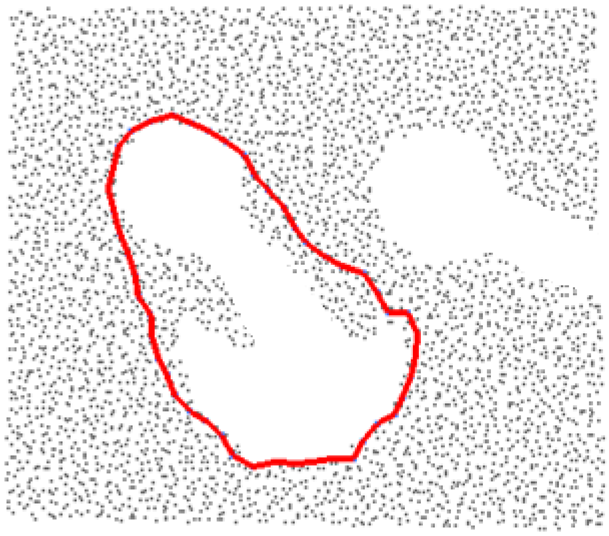


Figure 6: The coarse inner boundary a multi-hole scenario with all but one cut branch removed, one interior hole connected to the outer boundary.

Lemma 3: Building a shortest path tree between the two nodes (p,q), the closets nodes in the furthest cut component from the root, will result in a coarse inner boundary delimiting all inner holes in the sensor network.

Proof: All inner holes, except the furthest one from the network root, connect themselves, either to a network boundary or to another inner hole/s when their cut pairs are removed. We will prove that the shortest path between p and q, the two closets nodes in the furthest component (a.k.a. component *C*) to the root creates a ring circling all the inner nodes not connected to a boundary (called the composite hole *H*). We will assume without loss of generality, that the Component *C* is composed of three inner holes *x*,*y* and *z*, where z is the furthest away from the network root and (p,q) is the closest cut pair in the cut component defining hole *z*. (see Figure 7).

Since all edges between the cut pairs in the component *C* are removed, a gap will be formed from the inner hole *z* till the network boundary, forcing any continuous path p-q to circle the inner hole *z*. We will assume that exists a path p-q that circles only the hole *z* and does not circle the other holes *x* and *y*. But that stands in contradiction to our definition of the composite hole *C* that connects holes *x*,*y* and *z*. Therefore, every path that circles hole *z*, will circle *x* and *y* as well.

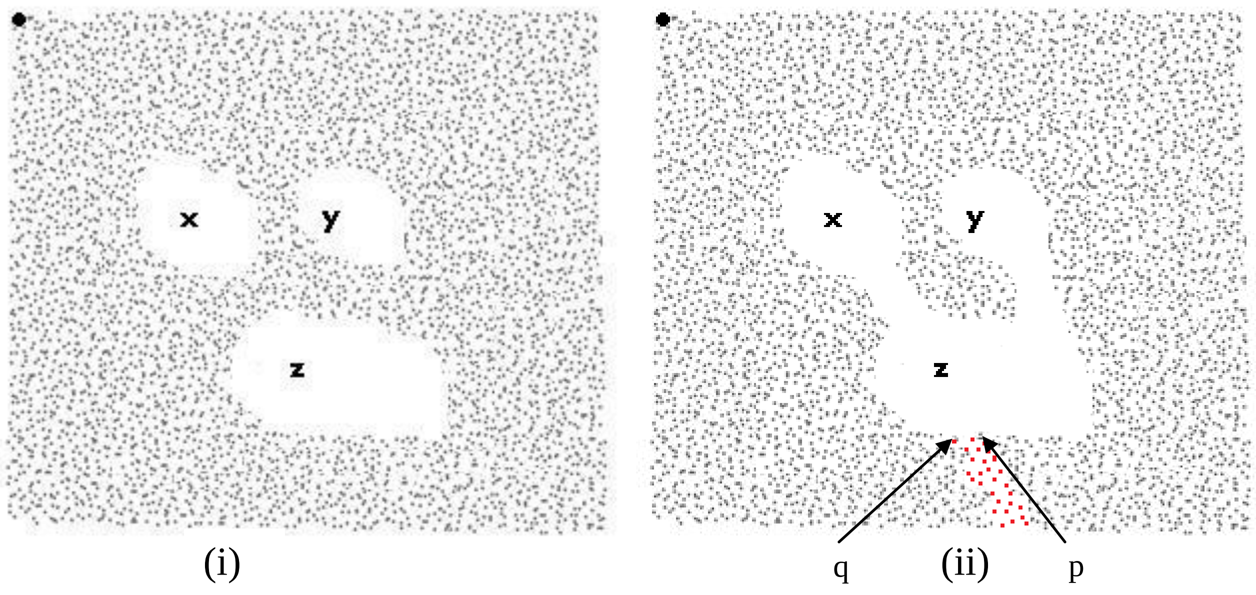


Figure 7: (i) and (ii) display the composite hole *C* composed out of inner holes *x*,*y* and *z* before and after the removal of cut pairs respectively.

3.1.5 Find the extremal nodes

An extremal node is a node whose minimum distance to nodes in the coarse inner boundary R is locally maximal. To discover extremal nodes, we have the nodes on R synchronize among themselves and start flooding the network at roughly the same time. We can see how to synchronize distributed nodes in Elson [3] and Ganeriwal et al [4] where the complexity is bounded by O(n). Each node in the network records the minimum RSSI to nodes in the coarse inner boundary R. At the end of the run, each node in the network holds the minimum distance to its closest node on boundary R. This is as the nodes merge in R to a dummy root σ, and build a shortest path tree T(σ), rooted at σ for the whole network. The extremal nodes are the ones with **locally maximum** distance to R. Each extremal node can detect itself by checking its direct neighbors. Intuitively, the extremal nodes are on the outer boundary or are the ridges on the real inner boundary of a concave hole as can be seen in Figure 8.

The message complexity in this process is upper bounded by O(n + k), where n is the number of nodes and k is the connectivity degree. This is due to the fact that the propagation is done locally, meaning, node does not forward a message with a greater RSSI that it already encountered with. Therefore, each node sends up to a constant number of messages, total of O(n) messages, plus the verification each node commits at the end of the propagation process with all his neighbors in order to discover local maximum, that is O(k) messages.

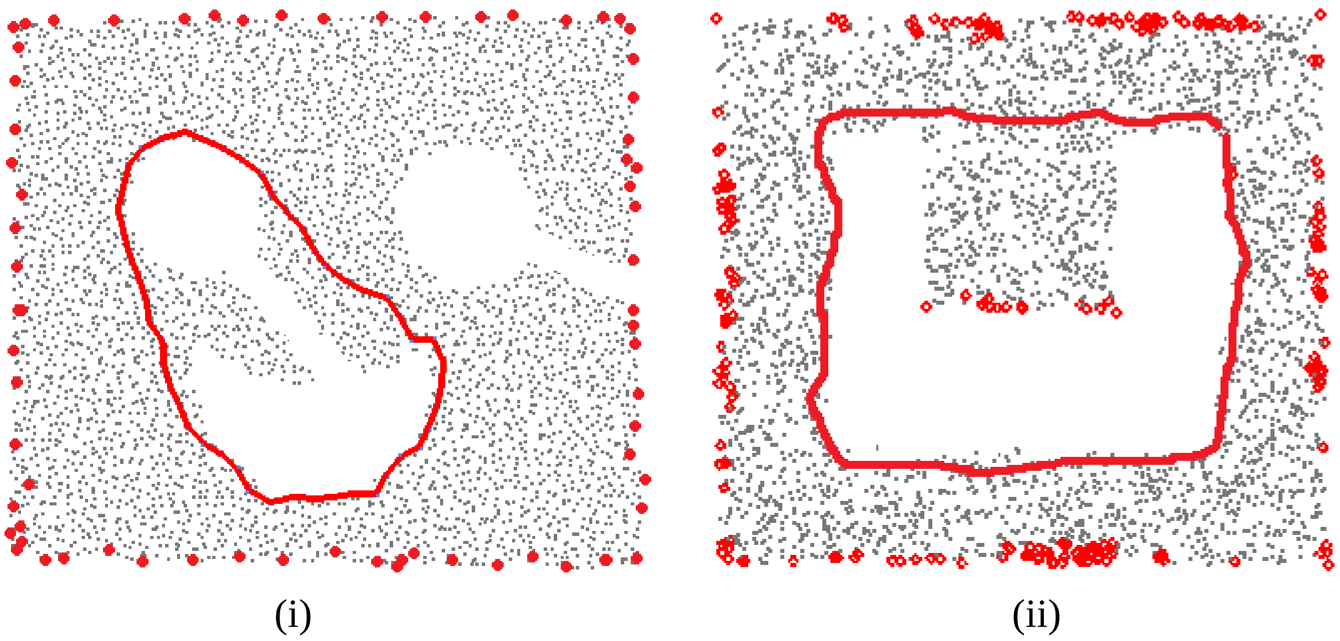


Figure 8: (i) The coarse inner boundary R of multi-hole scenario and the extremal nodes. (ii) Boundary R for one concave hole scenario and the extremal nodes (red) with locally maximum distance from the coarse boundary.

3.1.6 Find the outer boundary and refine the coarse inner boundary

In this step an attempt is made to connect the extremal nodes into a consistent cycle. Notice that here we only consider the case with one inner hole, since we have removed the cuts to connect multiple holes into a composite hole. If this composite hole is a convex, as in the multi-hole case, then all extremal nodes belong to the outer boundary (see Figure 8(i)). If the composite hole is concave (see Figure 8(ii)), there will be two types of extremal nodes, those on the outer boundary and those on the interior of R. We use the following rule to differentiate extremal nodes on different sides of R. The removal of R, together with neighbors of within distance Δ from R, partitions the network into several connected components Ci, i = 1, . . . ,w. We select Δ, as a small fraction of the network diameter *d*.

An attempt will be made to connect the extremal points into extremal paths in each connected component, which will then be further connected into boundary cycles. Focus will first be on a particular connected component and describe how an extremal path is constructed. Specifically, all of the nodes that are closest to Δ distance from R on the upper bound will connect themselves through local flooding, this results in a path (or a cycle) Pj along R. Then path Pj is refined so as to force it to go through the extremal nodes in this connected component. Notice that each extremal node has shortest path from the closest node on R. Next the extremal nodes will be connected by the shortest path between two closest extremal nodes. To find the shortest path between two extremal nodes, one either uses any shortest path algorithm or a greedy approach. For the latter, there is a natural path between any two extremal nodes u, v, composed by the shortest path from u to its closest point on Pj, u’ , the shortest path from v to its closest point on Pj, v’, and a segment of Pj between u’ and v’. Then, this path can be greedily refined and a shorter path between u and v can be found. By the above procedure, the extremal nodes are connected into a path Pj, in each connected component. The step is bounded by O(*n*) messages.

Lemma 4: The shortest path connecting the outer extremal nodes is a cycle, and is called the outer boundary.

Proof: Due to assumption 1, it can be concluded that each outer extremal node has only two other extremal nodes as its neighbors, one on each side. Connecting each outer extremal node to its two neighbors will necessarily create one cycle due to assumption 2, which promises that the edge connecting the two neighbors lie on the same boundary side, not allowing any crossovers between two boundary sides that will result with more than one cycle at the outer boundary.

It becomes necessary to check whether the path Pj produces a cycle for the inner boundary. This is done in the following distributed way: Each node on the path checks whether it has two adjacent neighbors, if so, then the path is actually a cycle, which is called the inner boundary cycle. Otherwise, the nodes on the two path’s edges, the ones who discovered they have only one neighbor, will connect themselves to the coarse boundary R. We do so by flooding a message connection request locally till we reach the closest node on R and branch to it. This can be done in the following way: when a connecting message reaches a node on the boundary R, it returns on the same path to the source edge, accumulating the RSSI. The source edge then, selects the shortest path to the boundary R. The last thing left is to disconnect the original segment of R that is delimited by the two nodes on R that the path Pj branched to. This will eventually come back and close a cycle, called the inner boundary.

Lemma 5: connecting the two edges of the path Pj to the boundary R after removing the redundant segment in R, will result with the inner boundary.

Proof: R is a cycle from definition, therefore, removing a segment delimited by two nodes on the cycle and connecting a new path (line) between these two nodes, will also result in a cycle. The inner boundary has to go through the inner extremal nodes and generate a cycle. R is the tightest inner boundary surrounding the inner hole, by definition above. Hence, connecting the path Pj to the boundary R by the shortest path between them will guarantee the tightest inner boundary. Since, if exists a closer node to the inner hole in the network, it would already be on the boundary R, except for the removed segment on R, where in that case, there is no tighter boundary than the shortest path between Pj to R.

So far, we classify extremal nodes and connect them into two cycles, corresponding to the inner and outer boundary. Figure 9 shows the results of this stage for the two examples.

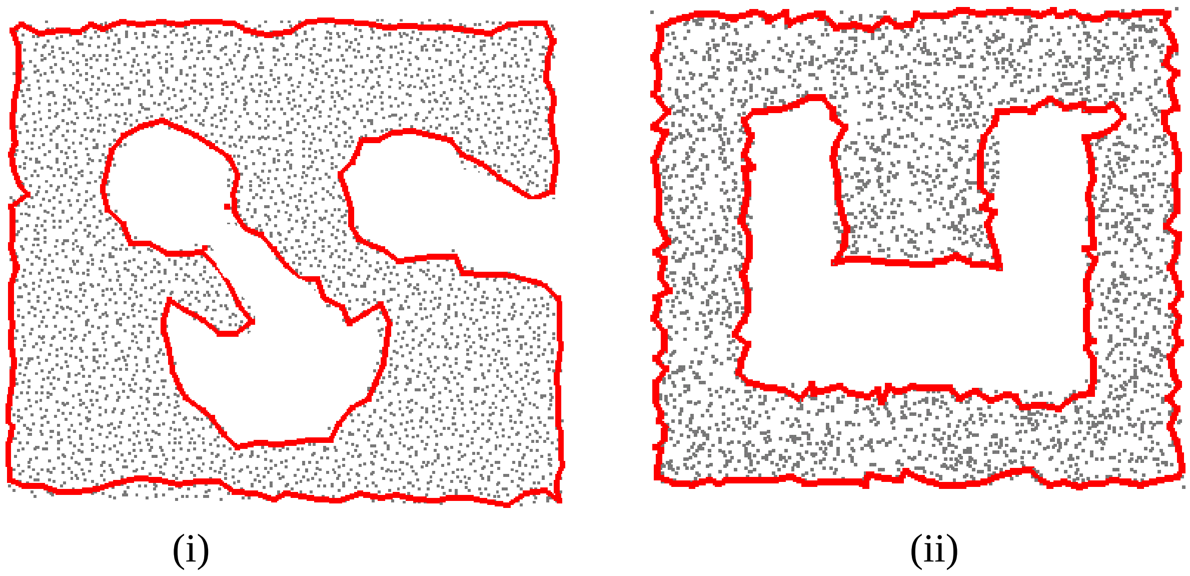


Figure 9: The outer boundary and the refined inner boundary in: (i) multi-hole example (with cut nodes). (ii) one concave hole example.

3.1.7 Restore deleted nodes and refine inner boundary for multiple holes

The final step is to recover the inner holes in the sensor field and refine their boundaries. The cut nodes that were removed earlier by the dead/alive bit will be restored, as can be seen Figure 10(i). For each cut branch there is a leading cut pair (p,q) that is the closest to the network root, as mentioned earlier. The two nodes in this cut pair (p,q) will be connected. If p and q are not on the refined inner boundary R, one traverses upwards in the shortest path tree T, and connect the edges, until a parent that is on the inner boundary R is reached. As a result, the inner boundary R is partitioned into two boundary cycles, as can be seen in (Figure 10(ii).

The two new boundary cycles will share nodes p, q. Then cycles are shrunk locally to make them tight. Here, the shrinking procedure has a restriction that the shrunk path still has to pass through the extremal nodes. Thus, the refined inner boundary is partitioned into a number of cycles, each representing the boundary of an inner hole. The step is bounded by O(*n*). Figure 11 demonstrate the final result.

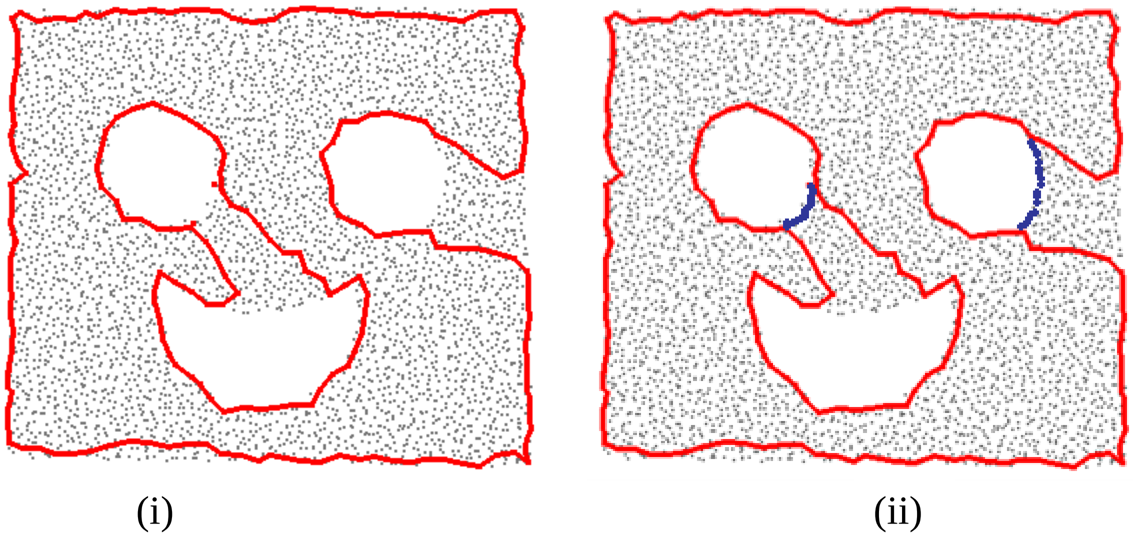


Figure 10: multi-hole example: (i) after recovering the cut nodes. (ii) after connecting the “leader” cut branch pair to inner boundary and partition R into two cycles .

Lemma 6: traversing upward from a cut pair (p,q) in the shortest path tree T will finally reach a node on the interior boundary R.

Proof: The coarse boundary R was created after holes were detected and merged by connecting nodes on tree T to get to the LCA. The refined R was created by shrinking the coarse boundary every time we found a smaller shortest path between two nodes. Even after refining the boundary, the “root” of boundary R remains the LCA. The cut pair (p,q) are decedent nodes of the same LCA, from the definition of a cut pair. Therefore, when one traverse upwards from p and q towards the LCA, he knows for sure that he will encounter nodes on R in his way, since in the worst case, one will encounter the LCA itself, which is on the boundary R.

Lemma 7: Connecting the edge pq (where p and q are a cut pair) to the boundary node R will result in portioning R into two boundary cycles.

Proof: Since R is the refined boundary rooted at the LCA after the shrinking process tightened the boundary to the hole, connecting the edge pq to the boundary nodes p and q encounter on R, will either (i) close a tighter cycle around the hole, that could not have been found thus far since p and q were deleted or (ii) transect the cycle of R and conform a loosen path to the LCA. In either case the R would be split. Now, the reason this split will close a cycle is because p and q will connect R from two different sides. This is because p and q are neighbors with different ancestors that never meet, until the intersection with R is reached in two different places creating inner cycle or the LCA is reached and even then they will connect to the LCA from both sides conforming a cycle.

Note:

In the continuous case all of the boundary points are identified as extremal points. However, in a discrete network, the shortest path is computed on a combinatorial graph. Thus, not all of the boundary nodes are identified as extremal nodes. The boundary refinement can be performed in an iterative fashion such that we commit message flooding from the current boundary cycle and identify more extremal points until the boundary cycle is sufficiently tight.

Wang et al [1] proved rigorously that the algorithm will correctly detect all the inner and outer boundaries in a continuous domain with polygonal boundaries, the essence of the proof can be found in section 4.2 below.

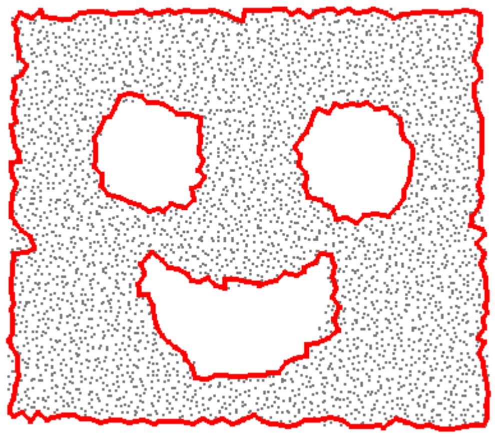


Figure 11: The final boundary cycles created for multi-hole example.

3.1.8 "no inner holes" scenario

This section will prove the correctness of the algorithm in the scenario where there are no inner holes. It was mentioned earlier the two approaches to solve that case, decreasing δ_1 or choosing an arbitrary node as inner boundary

.

If one chooses to decrease the value of δ_1, until he finds artificial holes, then the algorithm continues with no change and the general proof applies here as well. If he chooses an arbitrary node or the network root *r* as the inner boundary, then he will find extremal nodes and create an outer boundary surrounding r. Since, the extremal nodes are the ones with locally maximum distance to R, where R represents the nodes in the inner boundary. This is as if one merge the nodes in R to a dummy root σ, and build a shortest path tree T(σ), rooted at σ for the whole network. σ, is exactly what the network root *r* represent. After it has been proven that the extremal nodes selection process functions the same, the rest of the algorithm is straight forward, since the outer boundary is created by connecting the extremal nodes into a consistent cycle. In this scenario, all extremal nodes belong to the outer boundary, therefore, a simple shortest path tree between the extremal nodes will provide the refined cycle of the outer boundary. The algorithm can halt at that, as the next two steps: restore deleted nodes and refine inner boundary are not relevant when the inner boundary consists of one node and there are no nodes that where cut as can be seen in Figure 12.

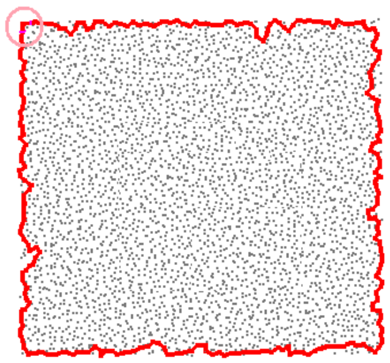


Figure 12: The outer boundary cycle created for no–hole scenario.

3.2 Proof of correctness in the continuous case

In this section meticulously demonstrated that the algorithm will correctly detect all the inner and outer boundaries in a continuous domain with polygonal boundaries. This proof is taken entirely from Wang et al [1]. The results apply also to non-polygonal domains, since they are approximated by polygons. Let F be a closed polygonal domain in the plane, with k simple polygonal obstacles inside; F is a simple polygon P, minus k disjoint polygonal holes. F is referred to as the free space and its complement, O, as the obstacle space. O consists of k open, bounded simple polygonal holes and an unbounded complement of the simple polygon P. Denote by V the set of all vertices of F.
For any two points p, q ∈ F, we denote by g(p, q) the geodesic shortest path in F between p, q. The Euclidean length of g(p, q) is denoted by d(p, q). The shortest path g(p, q) is not necessarily unique. In fact, our algorithm aims to detect points with one or more shorter paths to the root. Rigorously, given a root r ∈ F, the shortest path tree at r is the collection

of shortest paths from each point in F to r. A geodesic shortest path is a polygonal path with turn points at vertices of F [13]. We say that a vertex s ∈ V is a parent of a point p ∈ F if for some geodesic shortest path g(r, p), s is the last vertex along the path g(r, p) \{p} at which g(r, p) turns. If the geodesic path is a straight line from r to p, then r is the parent of p. The free space F can be partitioned to maximal regions, called cells, such that all the points in the same cell have the same parent or set of parents. This partition is called the shortest path map, SPM(r) (see Figure 13). The bisector C(vi, vj) is defined of two vertices vi, vj as the set of points in F with the set of parents {vi, vj}. A point in F with three or more parents is called an SPM-vertex. The union of all bisectors and SPM-vertices, B, is often called the cut locus. The boundary detection algorithm makes use of the properties of the shortest path map. List below are the useful properties that are proved in [14].

Lemma 8([14]): Given a closed polygonal region F in the plane, with k simple polygonal obstacles inside. The shortest path map at an arbitrary root r ∈ F has the following properties.

1. Each bisector is the union of a finite set of closed sub-arcs of a common hyperbola (a straight line is a degenerate case of a hyperbola).
2. The collection of bisectors and SPM-vertices, denoted by B, forms a forest.
3. There is at least one bisector point on the boundary of each obstacle.
4. F \ B is simply connected.

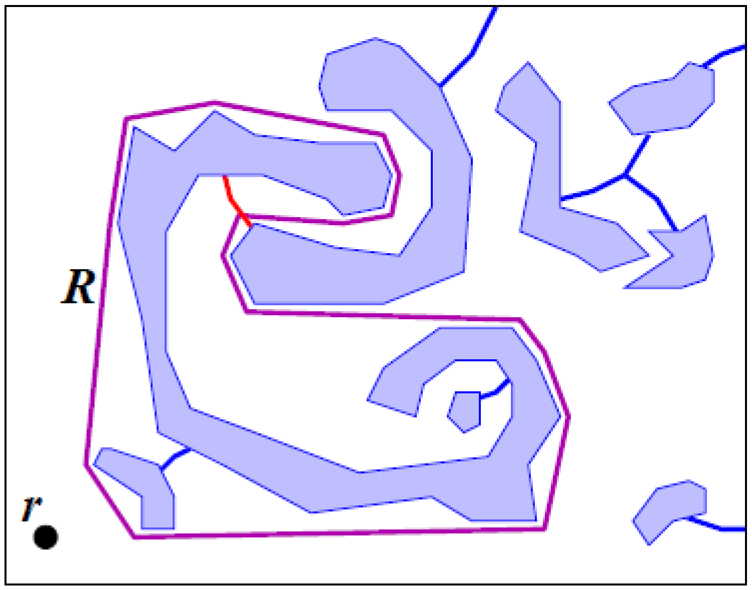


Figure 13: A shortest path map SPM(r) for F with k = 8 obstacles (blue), showing the bisector set B (red), which has one SPM-vertex, 7 connected components, and 9 arcs. R is the shortest cycle corresponding to the red bisector arc. Other bisector arcs are blue, having become part of the obstacle set for F′.

Lemma 9: For a region F with k polygonal holes inside and m SPM-vertices, there are exactly k + m bisector arcs.

Proof: This follows from the fact that a tree of v vertices has v−1 edges, where, here, v = k+1+m since the bisector arcs connect the k obstacle boundaries, the m SPM-vertices, and the 1 outer boundary. In fact, our boundary detection algorithm finds the bisector arcs. For completeness, it is re-state the outline of the boundary detection algorithm for the continuous case. The focus is on the correctness of the algorithm so it is explained in a centralized setting. The boundary detection algorithm for F works as follows.

1. Find the bisectors and SPM-vertices; each of them has at least two geodesic shortest paths to the root r.
2. Delete bisector arcs until there is only one bisector arc left. Correspondingly, connecting k − 1 holes into one composite hole. This results in a new domain F′ with only one interior hole O′ and one bisector arc.

3. Find the shortest cycle R in F′ enclosing the inner hole O′.
4. Refine R to find both the inner boundary and outer boundary of F′. In particular, the following algorithm was used for boundary refinement.
    a. Compute the Voronoi diagram of R in F′. Find the extremal points, defined as the points that do not stay on the interior of any shortest paths from points of F′ to its closest point on R, denoted by E. Furthermore, it was found for each extremal point the closest point(s) on R.
    b. Order the extremal points by the sequence of their closest point(s) on R. The extremal points are naturally connected into paths or cycles. Tour the coarse boundary R and replace the segments of R by their corresponding extremal paths. Touring the boundary twice will come up with two cycles that correspond to the inner and outer boundaries.
5. Undelete the removed bisector arcs. Restore the boundaries of interior holes and the outer boundary.

Next it will be proven that this algorithm correctly finds all the boundaries of F. The proof consists of a number of lemmas.

Lemma 10: The domain F′ has one interior polygonal hole.

Proof: This is due to the fact in Lemma 8 that F \ B is simply connected. Thus, by removing all but one bisector arc and all SPM-vertices, a polygonal region F′ is obtained with one interior hole.
The focus becomes F′ now and it is argued that we indeed find the correct outer and inner boundaries of F′ by the iterative refinement. The proof is only for the outer boundary R^+, the correctness of the inner boundary R^- can be proved in the same way. By the refinement algorithm, it is possible to find the Voronoi diagram of R by a wavefront propagation algorithm (e.g., [15]). The extremal points are the points that do not stay on the interior of any shortest paths from points of F′ to its closest point on R. Due to the properties of wavefront propagation, the extremal points must be either on the boundary or on the medial axis, where wavefronts collide [14, 16]. However, since cycle R is a

shortest path cycle surrounding the hole, we argue that the extremal points on the exterior of R must stay on the boundary of F′.

Lemma 11: The extremal points are exactly the boundary points of F′.

Proof: First it is argued that all the inward concave vertices of R must be on the outer boundary of F′. By the properties of geodesic shortest paths [17], all the vertices of R must be vertices of F′. If an inward concave vertex is a vertex of the inner hole, then one can move the concave vertex outward and shrink the cycle R, as shown in Figure 14(i) (the vertex of the inner hole w cannot be on R). This is a contradiction.
Thus, the removal of cycle R partitions the space F′ into disjoint components {Cj}. Correspondingly, it is denoted by Rj the segment of R that bounds the component Cj. By the above argument, each boundary segment Rj is inward concave. The wavefront propagation of R is actually composed of Voronoi wavefront propagation of each segment of Rj to its bounded region Cj. Since each segment is concave and the Cj's are disjoint, the wavefront propagation does not collide. All the extremal points can only happen at wavefront-boundary collision. In fact, all the boundary points collide with the wavefront propagation and thus are considered as extremal points.

At this point it is possible to consider R to be a coarse approximation to the outer (and inner) boundary. By the refinement algorithm one can improve the approximation and obtain the correct outer boundary. Specifically, the removal of R from F′ partitions the space F′ into disjoint components {Cj}, j = 1, . . . ,w. For each connected component Cj , the extremal points connect themselves into either a path (with open endpoints) or a cycle Pj .

- If w = 1, the extremal points form a cycle. R is already the inner boundary. All the extremal points are on the outer boundary.
- If $w \geq 2$ and one of Pj is a cycle, this cycle is the outer boundary. The rest of the extremal points refine R to the inner boundary. Specifically, we replace the segments on R by their extremal paths.

- Otherwise, one will tour R and discover the inner and outer boundaries. Specifically, as one travels along R and always prefer to branch on an extremal path. When the cycle is closed, a boundary cycle R_1 is obtained. Then we visit R again, starting at an extremal point that is not on R_1. Again it is prefered to branch on extremal paths. This will provide a different cycle. The two cycles are inner and outer boundaries. For an example, see Figure 14 (ii).

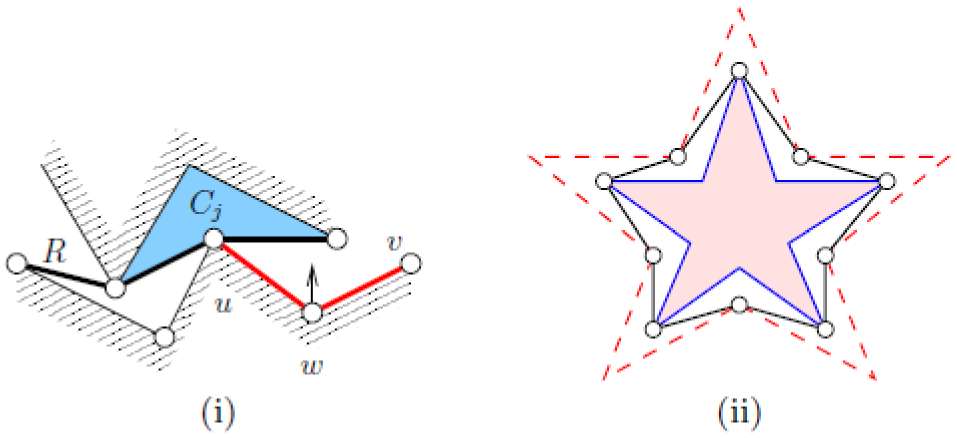


Figure 14: (i) All the inward concave vertices of R must be on the outer boundary of F′. (ii) The dark cycle is the coarse inner boundary R. The cycles in blue and (dashed) red are inner and outer boundaries.

With the correct inner and outer boundaries of F′, we are now ready to recover the real boundaries of F. One simply puts back the deleted bisectors. Indeed, the boundaries of F′ are the boundaries of F plus all the bisectors and SPM- vertices. Thus, removing the bisectors from the boundaries of F′ and obtain k + 1 cycles that are the real boundaries.

Theorem 1: Given a closed polygonal domain F in the plane, with k simple polygonal obstacles inside, the boundary detection algorithm will find the boundary of the region correctly.

3.3 Boundary nodes numbering

In order to find a finer and improved medial axis, the boundary nodes will be numbered sequentially. Boundary nodes are defined as all set of nodes that sit on inner or outer network boundaries.

3.3.1 Boundary leader election

To enable the numbering of boundary nodes a leader should be elected at each boundary cycle. Since, each boundary cycle is a ring and all nodes have a unique ID, we will choose the node with the highest ID (for example) as the leader. In order to do so, consider the following algorithm:

3.3.1.1 Leader election in a Ring Algorithm

1. Each node v does the following:
2. Initially all nodes are active. {all nodes may still become leaders}
3. Whenever a node v sees a message w with $w > v$, then v decides to not be a leader and becomes passive.
4. Active nodes search in an exponentially growing neighborhood (clockwise and counterclockwise) for nodes with higher identifiers, by sending out probe messages. A probe message includes the ID of the original sender, a bit whether the sender can still become a leader, and a time-to-live number (TTL). The first probe message sent by node v includes a TTL of 1.
5. Nodes (active or passive) receiving a probe message decrement the TTL and forward the message to the next neighbor; if the TTL is zero, probe messages are returned to sender using a reply message. The reply message contains the ID of the receiver (the original sender of the probe message) and the leader-bit. Reply messages are forwarded by all nodes until they reach the receiver.

6. Upon receiving the reply message: If there was no active node with higher ID in the search area (indicated by the bit in the reply message), the TTL is doubled and two new probe messages are sent (again to the two neighbors). If there was a better candidate in the search area, then the node becomes passive.
7. If a node *v* receives its own probe message (not a reply) *v* decides to be the leader.

Proof of correctness: Let node *z* be the node with the maximum identifier. Node *z* sends its identifier in clockwise direction, and since no node can turn off the leader-bit, at each TTL expiration, the reply message directed to *z* will have the leader-bit "on". Since the message TTL grows (exponentially) every round, eventually, *z* will receive its own probe message and declares itself to be the leader. Every other node will declare non-leader at the latest when forwarding message *z*.

3.3.1.2 Leader election in a Ring complexity

The algorithm time is bounded by O(*n*) and message complexity is bounded by O(*n*log(*n*)).

Proof: The time complexity is O(n) since the node with maximum identifier *z* sends messages with round-trip times 2, 4, 8, 16, . . . , $2 \cdot 2^k$ with $k \leq \log(n + 1)$. (Even if we include the additional wake-up overhead, the time complexity stays linear.)
The message complexity is O(nlog(*n*)), since, if a node *v* manages to survive round *r*, no other node in distance 2^r (or less) survives round *r*. That is, node *v* is the only node in its 2^r neighborhood that remains active in round *r* + 1. Since this is the same for every node, less than $n/2^r$ nodes are active in round *r*+1. Being active in round *r* costs $2 \cdot 2 \cdot 2^r$ messages. Therefore, round *r* costs at most $2 \cdot 2 \cdot 2^r \cdot \frac{n}{2^{r-1}} = 8n$ messages, bounded by O(*n*). Since there are only logarithmic (log(*n*)) many possible rounds, the message complexity is thus O(*n*log(*n*)).

3.3.2 The node numbering process

The leader of each boundary cycle starts the numbering process by initializing its local ID in the cycle to one and sends a message to its closest boundary node (e.g. clockwise) with its local ID. Each node receiving the message increases the counter, stores the counter as its local ID and forwards the message to the next node on the boundary. When the message reaches the boundary leader, it issues a new message containing two values: the number of nodes in the cycle (for latter normalization) and the boundary cycle ID (CID) which can be any arbitrary unique number, e.g. the boundary leader ID. At the end of the run, each node holds three values: local ID, number of nodes on the boundary and the CID.

Lemma 12: The time and message complexity of the numbering process is O(m) for each boundary cycle, where m is the number of nodes in the cycle. And the total message complexity of all numbering processes in all the boundary cycles in the network is upper bounded by O(n).

Proof: Since each boundary is a cycle, a message issued by one of the nodes will take m message transmissions, where m is the number of nodes in the cycle until it reaches the issuer. Therefore, the numbering process for a single boundary takes 2m time and messages. Since boundaries cannot collide (from definition), each node in the network is either a simple node or belongs to one of the boundaries. Hence, the numbering process for all the boundaries in the network cannot exceed O(n).

3.4 Find the medial axis of the sensor field

Once aware of the network boundaries and boundary nodes, one can find the medial axis.

Definition 4: The medial axis is defined as the set of nodes with at least *two closest boundary nodes*. That means, one is looking for nodes that their distance to the closest boundary nodes is equal up to some ε.

In the literature, many articles where written regarding the calculation of the medial axis, as in [2,5,6,7,8]. A new method was chosen that gives better results under the sensor filed assumptions (see assumptions 1 & 2) and the prior knowledge regarding the network boundaries that was acquired.

Intuitively, the medial axis can be thought of as the Voronoi cell boundaries of the Voronoi diagram. Therefore, the detection of nodes on the medial axis can be done locally, in the distributed following way: Starting from all boundary nodes and flood the network simultaneously. Each node thus records its closest boundary. All of the nodes with the same closest boundary are naturally classified to be in the same cell of the quasi-Voronoi diagram. Further, the nodes at which the frontiers collide, meaning they are in the same distance within ε from at least two boundary nodes, belong to the medial axis. The medial axis is 'cleaned up' from nodes having their closest points on the boundary in a consecutive interval.
The steps mentioned above will be elaborated on, in the following sections.

3.4.1 Build the Voronoi diagram

The Voronoi diagram on the boundary nodes is computed in the following distributed way. The boundary nodes will be marked as the Voronoi sites (see Figure 15). Essentially all the boundary nodes flood the network simultaneously and each node records the closest boundary node/s. A node *p* will not forward the message if it carries a RSSI greater than the smallest RSSI *p* has already received. Thus the propagation of messages from a boundary node *b* is confined within *b'*s Voronoi cell. All the nodes with the same closest boundary node are naturally classified to be in the same cell of the Voronoi diagram. Nodes with more than one closest boundary node, defined by equal RSSI within ε, to at least two different boundary nodes, stay on Voronoi edges or vertices.

Message suppression was used to reduce the communication cost. Since messages are forwarded only within the cells, the number of messages transmitted by each node is within a small constant, allowing the total message complexity to be bounded by O(*n*).

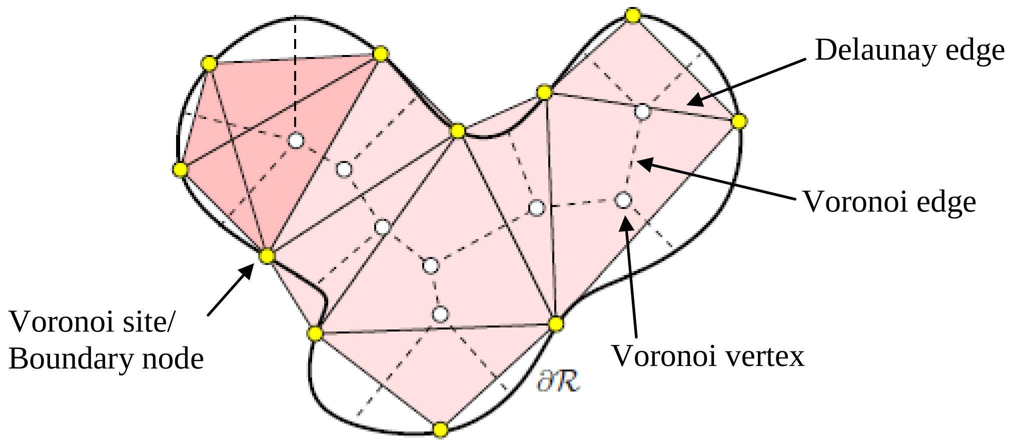


Figure 15: The Voronoi graph illustration (shown in dashed lines) and the Delaunay complex for a set of boundary nodes.

In the next five lemmas, proof will be offered regarding the construction of Voronoi diagram correctness in a network that contains holes, since the classic case of Voronoi diagram is proven in the literature, as in [9, 10].

Lemma 13: Given a disk B containing at least two points on ∂R, for each connected component of B ∩ R, either it contains a point on the inner medial axis, or its intersection with ∂R is connected.

Proof (taken from [11]): One can take one connected component C of B ∩R and assume that it does not contain a point on the inner medial axis and intersects ∂R in two or more connected pieces. Now take point *u* in C that is not on ∂R. Now take *u*'s closest point on C ∩ ∂R. If the closest point is not unique, then *u* is on the inner medial axis and we have a contradiction. Now the closet point *p* stays on one connected piece of C ∩ ∂R. Take *u*'s closest point on a different piece of C ∩ ∂R, denoted as *q*. See Figure 16. Now, a point *x* is moved from *u* to *q* along the geodesic path between *u* and *q*, *x*'s closest point on C ∩ ∂R starts with *p* and eventually becomes *q*, so at some point *x* the closest point changes. That point *x* is on the inner medial axis. This leads to a contradiction, and hence the claim is true.

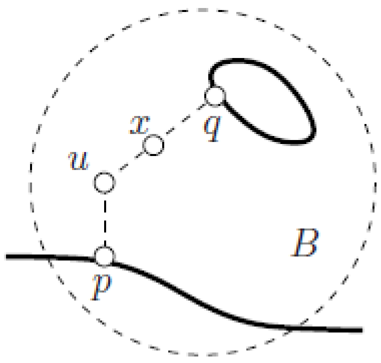


Figure 16: Each connected component of B ∩R either contains a point on the inner medial axis or its intersection with ∂R is connected.

Lemma 14: For any two adjacent boundary nodes *u, v* on the same boundary cycle, there must be a Voronoi vertex inside R whose closest boundary nodes include *u,v.*

Proof (taken from [11]): Take two adjacent boundary nodes *u, v* and consider the set of points in R with equal distance from *u, v*. The mid-point on the geodesic path connecting *u, v*, denoted by *x*, is at an equal distance from *u, v*. Take a disk through *u, v* centered at *x* and move the disk while keeping it through *u, v*. Its center will trace a curve called C(*u, v*) with all the points on C(*u, v*) having equal distances from *u, v*. C(*u, v*) has two endpoints *p, q* with *q* on the boundary segment in between *u, v* and *p* also on the boundary. Take $r = d(p, u) = d(p, v)$. See Figure 17. Here it is claimed that there must be a Voronoi vertex on C(*u, v*) that involves *u, v* and it is proven by contradiction. Otherwise, *p*'s two closest boundary nodes are *u, v* : the ball $B_r(p)$ centered at *p* with radius *r* contains no other landmark inside. When one takes $r^- = r - \varepsilon$ with $\varepsilon \to 0$. Thus $B_{r-}(p)$ contains no boundary nodes. Now it can be seen that this will violate the sampling condition if one can show that there is a point on the inner medial axis inside $B_{r-}(p)$ (meaning that $r^- \geq \text{ILFS}(p)$). The connected component of $B_{r-}(p) \cap R$ that contains the curve C(*u, v*), denoted by F is taken. By Lemma 13, if F does not contain a point on the inner medial axis, then its intersection with the boundary ∂R is connected. Now, one does a case analysis depending on how the boundary curve goes through *u* and *v*. In Figure 17 (i) & (ii), the ε-neighborhood of the boundary at *u, v* also intersects $B_{r-}(p) \cap R$. In (i), $F \cap \partial R$ has two connected pieces, thus leading to a contradiction. In (ii), the boundary between *u, v* through *p* is completely inside $B_{r-}(p)$, which has no other boundary node inside. In this case there are only 2 boundary nodes, namely *u*, *v*, on the boundary cycle containing *p*. This contradicts the sampling condition. If the boundary at *v* (or *u*, or both) is only tangent to $B_{r-}(p) \cap R$ (meaning that $B_{r-}(p)$ does not contain any ε-neighborhood of *v*, see Figure 17 (iii) & (iv)), we argue that F contains a point on the inner medial axis. To see that, we take the ball B_r(p) tangent at *v* with *v*'s ε-neighborhood outside the ball. Now we shrink it while keeping it tangent to *v* until it is tangent to two points on the boundary of F. Now the center of the small ball B′ is on the inner medial axis, which is inside $B_{r-}(p)$. Thus we have the contradiction. The claim is true.

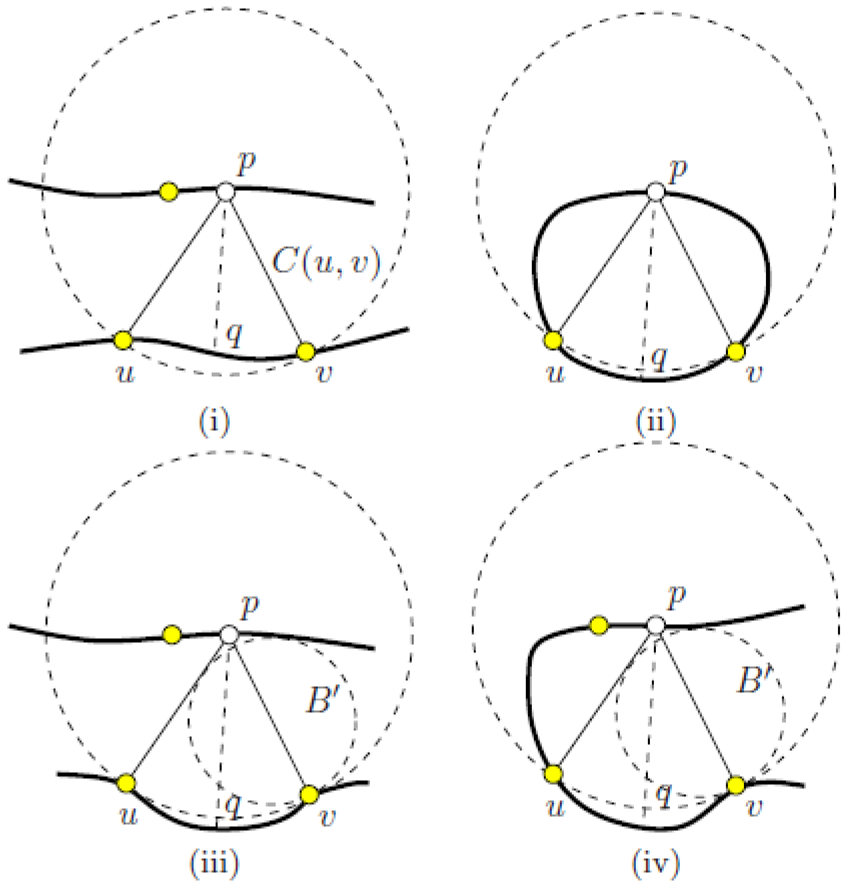


Figure 17: *u, v* are two adjacent boundary nodes. The point *p* on the boundary has its closest boundary nodes as *u, v*. (i)-(iv) four possible cases.

Lemma 15: If there is a continuous curve C that connects two points on the boundary ∂R such that C does not contain any point on Voronoi edges, then C cuts off a topological 1-disk of ∂R with at most one boundary node inside.

Proof (taken from [11]): Without loss of generality it is assumed that C has no other boundary points in its interior. Assume C connects two points p, q on the boundary. Since C does not cut any Voronoi edges, C must stay completely inside the Voronoi cell of one boundary node say *u*. Without loss of generality assume that *u* is to the right of boundary point *q*. See Figure 18(i). Now the boundary of Voronoi cell of *u* is partitioned by the curve C, with one part completely to the left of C. Consider one of the intersections between the Voronoi cell boundary of *u* with the region boundary ∂R, say *p′*. Consider the ball $B_r(p')$ with $r = d(p', u)$. The point *p′* has two closest boundary nodes, with one of them as *u* and the other to the left of C, denoted as *w*. Now, this ball cannot contain any other boundary nodes besides *u*,*w*. It is argued by Lemma 13 that the component of $B_r(p')$ ∩R containing *p′* intersects ∂R in a connected piece. Otherwise $B_r(p')$ contains a point on the inner medial axis, which means $r >$ ILFS(p'). Thus by the sampling condition there must be a boundary node inside $B_r(p')$. Now, since the component of $B_r(p')$ ∩R containing *p′* intersects ∂R in a connected piece, this intersection is a continuous segment between *u* and *w* on ∂R, completely inside $B_r(p')$, by using the same argument as in the previous lemma; see Figure 18 (ii). In this case, the curve C cuts off a segment of ∂R with at most one boundary node inside. The claim is true.

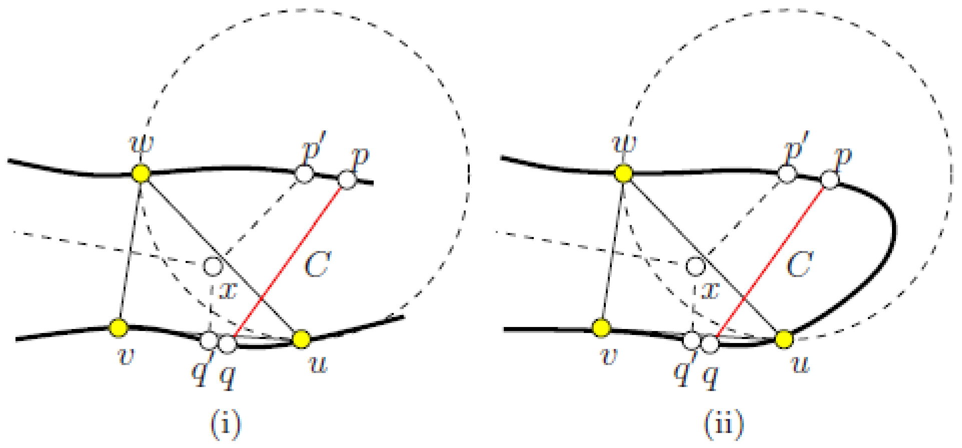


Figure 18: (i) C is inside the Voronoi cell of boundary node *u* to the right of C. (ii) the curve C cuts off a segment of ∂R with no other boundary node inside.

Lemma 16: The Voronoi graph V(S) is connected.

Proof : That is a corollary from lemmas 14 & 15. This can be proven in negation, as follows. If one assumes the V(S) is not connected, hence there is at least one possible curve C that cuts R into two pieces without intersecting with the Voronoi diagram, specifically, not cutting any Voronoi edges. In each cut section then, we can find some boundary nodes. This contradicts lemma 15, which says if curve C did not intersect any Voronoi edge, then exists maximum one boundary node in the form cut. Thus, the Voronoi graph is connected.

Lemma 17: lemma 16 is correct even in a network containing holes.

Proof: Let us examine the different components that the Voronoi diagram consists of, we will see that the boundary nodes (Voronoi sites) cannot be in any hole, since they are only at the networks boundaries (from the boundary nodes definition). The Voronoi vertices cannot be in the hole from definition, since they are nodes in R that are at equal distance from at least two boundary nodes. The only components that can be set across holes are the Delaunay edges (between two boundary nodes) or the Voronoi edges (between two Voronoi vertices or network boundary). We will distinguish between two cases. If the two nodes at the end of the edge, can communicate directly, than the edge going across the hole represents real edge and the RSSI represents the real distance between those two nodes.

Otherwise, since the Voronoi graph is connected (lemma 16), exists a shortest path between those two nodes. Hence the edge is a virtual edge between the nodes and its (RSSI) value represents the energy or the distance that a message has to go through between the two nodes in reality.

Either way, the edge going across a hole is reachable and represents the best approximation of the distance between nodes/vertices at the two ends.

3.4.2 Medial axis 'Clean up'

The nodes sitting on the Voronoi edges, where the cells frontiers collide are marked as part of the medial axis. Since, there are as many cells as there are many boundary nodes, the medial axis is too 'noisy', as in Figure 19(i). We wish to clean up the medial axis from nodes that their closest boundary nodes are on the same boundary cycle and are close to each other. We will define 'close to each other' by γ, where γ is a fraction of m_i, the number of nodes in the boundary cycle *i*. We will choose $\gamma = 0.1\, m_i$. All boundary nodes know from previous steps, their logical ID (LID) in the boundary cycle, the boundary cycle ID (CID) and the number of nodes in their cycle m_i.

Therefore, each node on the medial axis will check whether its closest boundary nodes lie on the same boundary cycle and are within distance γ from each other. If the answer to those two questions is true, the node will remove himself then from the medial axis.

Since, the node with the highest LID is adjacent to the node with the lowest LID in the boundary cycle, we will normalize the numbers so the check would be carried out in the following way:

$$m_i - \gamma \geq | LID_x - LID_y | \geq \gamma$$

Where, m_i is the number of nodes in the boundary cycle *i*, LID_x and LID_y are the logical IDs of the two closest boundary nodes x and y and $\gamma = 0.1\, m_i$.
Since all medial axis nodes know from previous steps their closest boundary nodes' logical ID (LID) in the boundary cycle, the boundary cycle ID (CID) and the number of nodes in their cycle m_i. it takes them O(1) to compute whether they should remove themselves or not and no message transmission is required. See Figure 19(ii) for results after the clean up process.

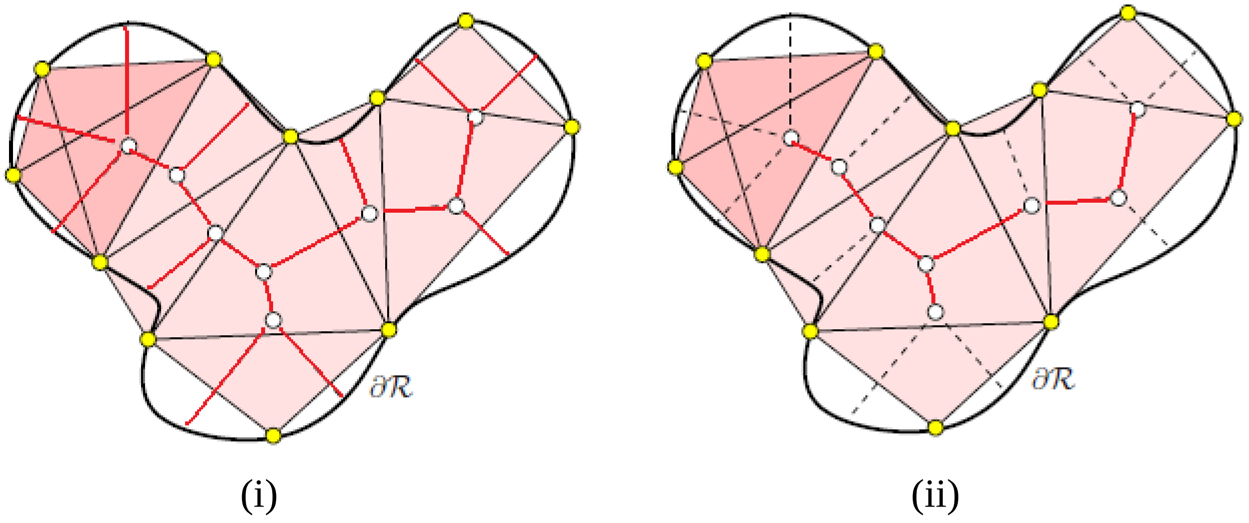


Figure 19: The medial axis marked in red. (i) Before the 'clean up' process. (ii) After the 'clean up' process with γ=1.

Lemma 18: The cleaning method described results with refined medial axis.

Proof: In order to 'clean up' the medial axis we wish to leave nodes that are on Voronoi edges between two cells originated at two boundary nodes (Voronoi sites):

1. From different boundary cycles.
2. From two different sides of the same boundary cycle.

One can immediately see that we fulfill the first condition, since we remove only nodes that are on the same boundary cycle. The dentition of 'different sides' is a little vague, since in a circle for instance, there are no clear sides. But it can be said that, the thing that characterizes nodes from different sides is that they are well apart from each other. And this is what γ gives us. We remove nodes that their boundary nodes are adjacent up to γ distance from each other. That way, we handle Voronoi edges between adjacent boundary nodes and we are not influenced by curves in the sensor field.

The complexity of the cleaning process is O(*c*). That is, because all nodes commit the check at approximately the same time, and no messages need to be passed, since the values of the CID, LID and m_i are known to all boundary nodes and can be passed as parameters in the Voronoi cells construction, where all boundary nodes propagate a message.

See Figure 20 to observe the results of the refined medial axis.

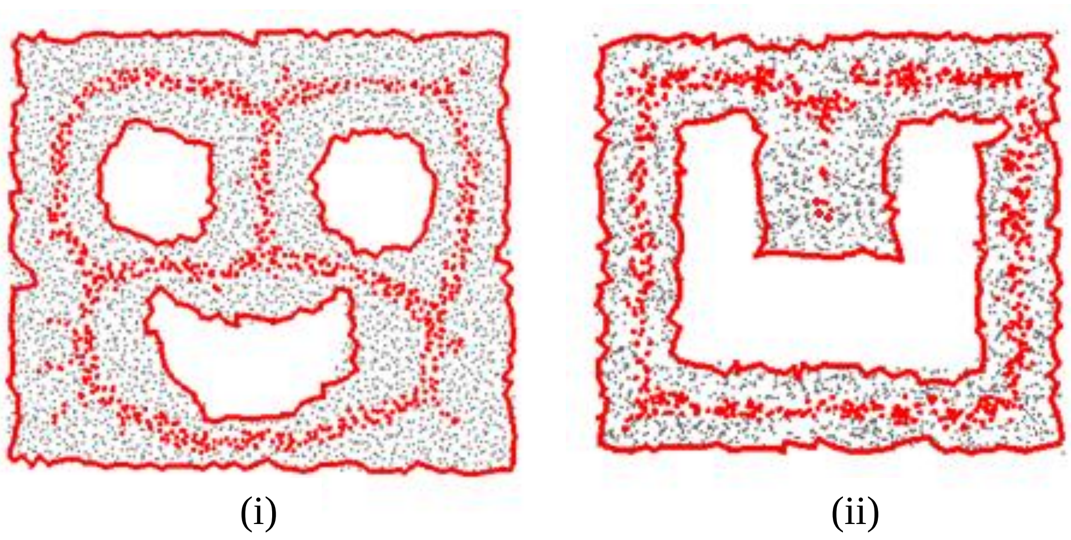


Figure 20: The refined medial axis diagram for the (i) multi hole example. (ii) concave hole example

3.5 Network bottlenecks discovery methods

There are several methods proposed in this section to identify bottlenecks once given the medial axis. It was decided to give few methods and not to stick with one, since one method might be more suitable than the other, based on the specific use case. In all the methods represented below, we are using the medial axis, since the medial axis gives us indication of the gap or distance between two network boundaries. Intuitively, the smaller the gap is, the narrower the network become and lowers the entire network throughput, causing the nodes in that area to become a bottleneck. A reminder, every node on the medial axis knows its distance to the closest boundary nodes, since that is what defined nodes on the medial axis.

3.5.1 Constant threshold

One can choose a constant threshold regardless of the network size, representing the minimum distance between boundaries. Every node on the medial axis having a distance to the boundaries that is less than the specified threshold is classified as a bottleneck. The time and message complexity is O(c), since the distance to the nearest boundary nodes in known to all nodes on the medial axis, leaving for them only to compare the distance to the constant threshold individually and with no message transmission.

This method can be useful, when prior simulations indicated the minimal network width necessary for an optimal throughput.

3.5.2 Fraction of the widest part in the network as threshold

One can choose the threshold as some fraction of the widest part in the network. This can be carried out by informing the nodes in the medial axis of the distance to the boundaries of all other nodes in the medial axis. Then, after taking the greatest distance, meaning the widest part, each node can compute individually whether he is above the threshold, which is some fraction of that distance. This can be implemented by flooding the values between the medial axis nodes and forwarding a message only when a node encounters a value greater than any other value he had seen thus far. The time complexity is bounded by O(m), where m represent the number of nodes on the medial axis and $m<n$, since the time it takes for a packet to travel between the two furthest nodes is maximum m. The message complexity is O(m^2), since there are m messages transmitted in every interval (every node transmits one message) and there are m intervals.

This method can be useful, when it is important not to reduce the highest achieved throughput (the widest part) in the network in more than a constant fraction of it.

3.5.3 Variation of the average or median

One can choose a threshold as a variation of the average or the median, e.g. all nodes that their distance to the boundaries is under the average distance or the median distance are defined as bottlenecks. This can be implemented by message flooding (containing the boundary nodes distance) between all medial axis nodes, where each node forwards to all its neighbors every packet it receives from another medial axis node. Then each node computes the average or median of the network individually. The time complexity is bounded by O(m) and the message complexity is bounded by O(m^2) as explained in the above paragraph.

This method can be useful when one wants to take all the gaps under consideration and hence look at the overall picture of the network. In this method one is also immune from deviations in the network. Since, the average or the median would not be influenced much if exists a small area in the network that is several orders of magnitude larger than the rest.

3.5.4 Lowest percentile

One can choose the threshold as the lowest percentile of gaps between network boundaries. This can be implemented by flooding messages from all nodes containing the nodes distance to the boundaries, same as above method only here each node has to store all distances it receives in buckets or some other means in order to find out after receiving the distances from all the nodes, whether he is the lowest percentile. The time complexity is bounded by O(m) and the message complexity is bounded by O(m^2) as explained in the above paragraph.

This method is useful like the average and median threshold, when one wishes to look at the overall picture of the network and be immune from extreme deviations. But also very applicable in the real world, when we have a limited amount of new nodes we can contribute to the system and we want to know where the network pain is. Since we are aware of the amount of nodes in the network, we can deduce from that what is the percentage of the nodes we are about to contribute, and add them to the area where the lowest percentile found.

4 Conclusion

This paper addressed the network bottleneck detection problem in WSNs. A brief overview of the problem and motive as well as some research that was carried out in this field was provided. A new distributed algorithm that addresses the problem and results with the detection of the bottlenecked nodes in the network was provided. After the algorithm execution is completed, the network inner and outer boundaries, as well as the network medial axis are received as byproducts. The total run time complexity of the algorithm is bounded by O(*n*) and the total message transmission complexity is bounded by max(O(*nlog(n)*) , O($m^{2`}$)), where *m* represents the number of nodes constructing the medial axis. A detailed breakdown can be seen in Table 1.

One can find at Wang et al [1], extensive simulations in various scenarios as for the network inner and outer boundaries detection algorithm.

Future work might include committing simulations and experiments in the real world with a large number of sensors, in order to examine the accuracy of the medial axis discovery and the bottleneck detection suggested in this article. Furthermore, new mathematical models can be suggested and investigated upon given the medial axis and network boundaries to better address the problem.

Step	**Time complexity**	**Message complexity**
Leader election	n	$nlog(n)$
Shortest path tree construction	d	n
LCA discovery	n	n
Network holes discovery	n	n
Coarse inner boundary detection	d	n
Extremal nodes detection	n	n
Outer boundary discovery	n	n
Inner boundary refinement	n	n
Ring leader election	n	$nlog(n)$
Boundary nodes Numbering	d	n
Voronoi diagram construction	n	n
Medial axis clean up	c	-
Bottlenecks detection with constant threshold	c	c
Bottlenecks detection – all other methods	m	m^2

Table 1: The algorithm complexity breakdown, where *n* represents the number of nodes in the network, *d* represents the network diameter, *m* represents the number of nodes constructing the medial axis and *c* stands for constant number.

5 Acknowledgements

I would like to thank *Prof. Danny Dolev* for advising and supervising throughout the presented work. *Danny Dolev* helped me by routing my ideas into a complete product and in addition, offering valuable input and constructive criticism along the way.

I would like also to thank *Bracha Hod* that introduced the world of sensor networks to me during my Master’s degree and demonstrated infinite patience and optimism in our collaborated work implementing communication protocols on the sensors in the DANSS lab.

Last but not least, to my family for supporting and taking interest in my thesis and especially for my father *Prof .Dennis Cogan* for proof reading my work.

6 References

[1] Wang, Y., Gao, J., and Mitchell, J. S. B. (2006). Boundary recognition in sensor networks by topological methods. In Proc. of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom). 122–133.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In Proc. 14th ACM Sympos. Parallel Algorithms and Architectures, pp. 258–264, 2002.

[3] J. Elson. Time Synchronization in Wireless Sensor Networks.PhD thesis, University of California, Los Angeles, May 2003.

[4] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In Proc. 1st Internat. Conf. on Embedded Networked Sensor Systems, pp. 138–149, 2003.

[5] H. I. Choi, S. W. Choi, and H. P. Moon. Mathematical theory of medial axis transform. Pacific Journal of Mathematics, 181(1):57–88, 1997.

[6] J. Bruck, J. Gao, and A. Jiang. MAP: Medial axis based geometric routing in sensor networks. In Proc. ACM/IEEE Internat. Conf. on Mobile Computing and Networking (MobiCom), pp. 88–102, 2005.

[7] D. Attali, J.-D. Boissonnat, and H. Edelsbrunner. Stability and computation of the medial axis - a state-of-the-art report. In Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration. Springer-Verlag, 2004.

[8] O. Aichholzer, W. Aigner, F. Aurenhammer, T. Hackl, B. J¨uttler, and M. Rabl: Medial axis computation for planar free-form shapes. Computer-Aided Design, Vol. 40, Elsevier, 2008

[9] F. Aurenhammer, “Voronoi Diagrams—A Survey of a Fundamental Geometric Data Structure,” ACM Computing Surveys, vol. 23, pp. 345-405, 1991.

[10] S. Fortune. Voronoi diagrams and Delaunay triangulations. In Computing in Euclidean Geometry, F.K. Hwang and D.Z. Du, eds., World Scienti_c, 1992.

[11] S. Lederer, Y. Wang, and J. Gao. Connectivity-based localization of large scale sensor networks with complex shape. In Proc. of the 27th Annual IEEE Conference on Computer Communications (INFOCOM’08), pages 789–821, May 2008.

[12] I. Cidon and O. Mokryn, “Propagation and Leader Election in Multihop Broadcast Environment”, 12th International Symposium on DIStributed Computing (DISC98), September 1998, Greece. pp.104–119.

[13] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry: Algorithms and Applications. Springer-Verlag, Berlin, 1997.

[14] J. S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. Ann. Math. Artif. Intell., 3:83–106, 1991.

[15] M. Held. Voronoi diagrams and offset curves of curvilinear polygons. Computer Aided Design, 30(4):287–300, 1998.

[16] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. SIAM Journal on Computing, 28(6):2215–2256, 1999.

[17] J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack and J.Urrutia, eds., Handbook of Computational Geometry, pp. 633–701. Elsevier, 2000.

[18] Daganzo, C. F. (1997), Fundamentals of Transportation and Traffic Operations, Elsevier, New York, 133–35, 259.

[19] Zhang, L. & Levinson, D. (2004), Some properties of flows at freeway bottlenecks, Journal of the Transportation Research Board, 1883, 122–31.

[20] A. Akella, S. Seshan, and A. Shaikh, "An Empirical Evaluation of Wide-Area Internet Bottlenecks," in IMC, 2003.

[21] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating internet bottlenecks: Algorithms, measurements, and implications. In Proc. ACM SIGCOMM, August 2004.

[22] Mechthild Stoer, Frank Wagner, "A Simple Min-Cut Algorithm," Journal of the ACM, Vol. 44, No. 4, July 1997, pp. 585–591.

[23] Haosong Gou, Younghwan Yoo, "Distributed Bottleneck Node Detection in Wireless Sensor Network". 10th IEEE International Conference on Computer and Information Technology (CIT 2010). Busan, Korea, 2010.

[24] Haosong Gou, Gang Li and Younghwan Yoo, "A Partion-Based Centrilized LEACH Algorithm for Wireless Sensor Network Using solar Energy", in Proceedings of the International conference on Convergence & Hybrid Information Technology 2009 (ICHIT 2009).

[25] P. Coy and N. Gross, "21 ideas for the 21st century," Business Week, pp. 78–167, Aug. 1999

האוניברסיטה העברית בירושלים
The Hebrew University of Jerusalem

מציאת צווארי בקבוק תקשורתיים ברשתות סנסורים מבוזרות

עבודה זו מוגשת במסגרת תואר מוסמך במדעי המחשב

על ידי

דוד קרפף – כוגן

בית להנדסה ומדעי המחשב ע"ש רחל וסלים בנין
האוניבריסטה העברית

העבודה נעשתה תחת פיקוחו של
פרופ' דני דולב

תשרי התשע"א

תקציר

בעולם בו העיסוק ברשתות סנסורים אלחוטיות מבוזרות (Distributed Wireless Sensor Networks) גובר עם השנים וחולש על מגוון רחב של תחומים, כגון: בתעשייה, צבא, רפואה ובמגזר הפרטי. ישנו עיסוק מתמיד בניסיון להפיק תועלת מרובה יותר מרשתות אלו ומאמצים רבים נעשים על מנת לשפר את אורך חיי הרשת. גילוי מוקדם של צווארי בקבוק תקשורתיים ברשתות הסנסורים וטיפול הולם הם אבן דרך משמעותית בדרך להשגת מטרה זאת. צווארי הבקבוק גורמים לעלייה במספר ההודעות הנזרקות ואי לכך לעלייה במספר השליחות החוזרות ברשת, כמו כן צווארי הבקבוק גורמים לירידה בתפוקה (*throughput*) ולעלייה בזמן התגובה (*latency*). בנוסף לכך, במידה וסנסור השייך לצוואר הבקבוק מושבת, יכול להיווצר מצב שחלקים מהרשת ייהפכו לבלתי נגישים, מה שיגרום לחלוקה של הרשת למספר קטעים.

ברשתות סנסורים מבוזרות, אשר מאופיינת על ידי פיזור רנדומלי של החיישנים המרכיבים אותן, בעלי אספקת חשמל נמוכה ומוגבלת המועדים לכישלון, גילוי צווארי הבקבוק הוא צורך קריטי לשמירת פעילותה התקינה של הרשת.

הכלים הקיימים כיום לגילוי צווארי בקבוק באינטרנט ואף הכלים החדשים יותר לגילוי צווארי בקבוק ברשתות סנסורים אלחוטיות שאני נתקלתי בהם, אינם שמים דגש על מספר ההודעות הנשלחות בתהליך. למרות שזו תכונה חשובה ברשתות אשר אספקת החשמל שלהם מוגבלת, לאור העובדה שתשדורת ברשת אלחוטית היא פעולה יקרה במונחים של צריכת אנרגיה. לדוגמא, שידור בית אחד בחיישן מסוג MSP430 לטווח 125 מטר, שוות ערך לכ- 4,000 פעולות מעבד (CPU), ולטווח של עד 300 מטר שוות ערך לכ- 32,000 פעולות מעבד.

עבודה זו מתבססת על האלגוריתם למציאת גבולות הרשת שפרסם Wang [1] עם התאמות מסוימות. לאחר מכן, היא מציעה דרך חדשה לחישוב הציר התיכון (*medial axis*) של הרשת. לבסוף, היא מציגה מספר שיטות למציאת צווארי הבקבוק ברשת המתבססות על ממצאים אלו.

זמן הריצה של האלגוריתם חסום ב- $O(n)$ וכמות ההודעות הנשלחות בתהליך חסום על ידי $\max(O(n\log(n)), O(m^2))$, כאשר m מייצג את מספר האיברים המרכיבים את ציר התיכון.