Serial In-network Processing for Large Stationary Wireless Sensor Networks

Mohammed Amine Merzoug
Department of Computer Science
Faculty of Exact Sciences
University of Bejaia
06000 Bejaia, Algeria
amine.merzoug@univ-batna2.dz

Azzedine Boukerche
PARADISE Research Lab.
University of Ottawa
Ottawa, Canada
boukerch@site.uottawa.ca

Ahmed Mostefaoui*
FEMTO-ST Institute, DISC Dept.,
Univ. of Burgundy-Franche-Comte
Belfort 90000, France
ahmed.mostefaoui@univ-fcomte.fr

ABSTRACT

In wireless sensor networks, a serial processing algorithm browses nodes one by one and can perform different tasks such as: creating a schedule among nodes, querying or gathering data from nodes, supplying nodes with data, etc. Apart from the fact that serial algorithms totally avoid collisions, numerous recent works have confirmed that these algorithms reduce communications and considerably save energy and time in large-dense networks. Yet, due to the path construction complexity, the proposed algorithms are not optimal and their performances can be further enhanced. To do so, in the present paper, we propose a new serial processing algorithm that, in most of the cases, approximates the optimal number of hops (i.e., it requires $n - 1$ communications to traverse a network of n nodes). The extensive OMNeT++ simulations confirm the outperformance and efficiency of the proposal in terms of scalability and energy/time consumption.

CCS CONCEPTS

• **Mathematics of computing → Paths and connectivity problems**; • **Networks → In-network processing**; **Sensor networks**;

KEYWORDS

In-network data aggregation; sensor query processing; serial data fusion; wireless sensor networks

1 INTRODUCTION

Usually in wireless sensor networks, the mission of the randomly deployed sensor nodes is to respond as quickly and efficiently as possible to the queries of the sink node (e.g., maximum sensed value, alive nodes count, etc.). To meet this goal and ensure communication efficiency, numerous recent research works have proposed a more effective alternative to the in-network structure-based approaches [5, 8, 15], namely the serial localized algorithms [3, 9, 11, 13, 14].

*Dr. A. Mostefaoui is a visiting Professor at PARADISE Research Lab.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSWiM '17, November 21–25, 2017, Miami, FL, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5162-1/17/11...$15.00
https://doi.org/10.1145/3127540.3127568

The three main features that differentiate serial algorithms from their structure-based counterparts are:

- Compact operation: in fact, one of the main reasons behind the bad performance of structure-based approaches is their mode of operation. Actually, these approaches operate in three separate phases: structure construction, query dissemination, and data processing. For example, at first, a spanning tree must be created. Once the whole network is covered, a query can be spread throughout the tree ordering nodes to perform a certain processing on data. After query dissemination, data processing starts from leaf nodes and goes up towards the root/sink. Performing these three phases separately not only increases energy consumption but also delays the response time. With regard to serial algorithms, energy and time are considerably saved through the combination of the three previous phases into one step. While the path is being gradually laid out throughout the network, at the same time, the query is disseminated and data is processed.
- Collision-free: in serial algorithms, the desired task is executed sequentially by each node while the network is gradually traversed. Hence, serial algorithms are inherently collision-free and no elaborated MAC layer is needed in these algorithms. As regards tree-based approaches, the network in this case is traversed in a parallel fashion. So, from a theoretical point of view, we can say that this feature would give an advantage to tree-based approaches and enhances their response time. In reality however, as many recent works have shown [3, 11], the parallel traversal creates a lot of collisions, which considerably wastes time and dissipates energy especially in large-dense networks. In fact, even the tree construction process is deeply affected by collisions.
- Structure-free: unlike structure-based approaches, serial algorithms do not rely on any pre-established structure (no path is built in advance). Instead, each time a query is issued, a new path will be gradually built by each traversed node. This characteristic makes serial algorithms more resistible to topology changes and links/nodes failures. On the contrary, the main concern in structure-based approaches is topology changes because rebuilding or fixing a structure that covers a large dense network, is a very time and energy-consuming task.

1.1 Motivation

The major challenge faced when designing a serial algorithm is ensuring the traversal of the entire network while reducing time and

energy consumption. Recently, several localized serial algorithms have been proposed in the literature [3, 11, 13]. Despite their interesting features and outperformance compared to structure-based approaches, these serial algorithms require an extra overhead to construct the path. So, our primary objective is to develop a scalable serial algorithm that shortens the traversal path and reduces communications to the maximum extent possible. We aim the optimal number of communications (i.e., $n-1$ packets to traverse a network of n nodes). No extra overhead or control packets must be required. Second, the proposed algorithm has to be able to traverse any connected network and ensure the visit of all nodes regardless of the topology (hole topology, regular or irregular topology, sparse or large-scale topology, etc.).

1.2 Contribution

We present in this paper a new serial processing algorithm, called GSS (for Geometric Serial Search). In this algorithm, no control packets are required, in fact, just one data packet that can be issued by any node, moves from node to node and traverses the entire network. As confirmed by the obtained OMNeT++ simulation results, in most of the cases, GSS approximates the optimal traversal path, which means that GSS scales well in large networks and significantly saves energy and time. For example, for a network of 500 nodes, GSS requires approximately 510 packets to visit each and every node. As the present paper shows, we have also compared GSS with other existing approaches. The obtained results confirm the efficiency and superiority of our proposal.

The remainder of this paper is organized as follows. In the next section, we specify the problem. Section 3 details the proposed solution. Section 4 presents and discusses the simulation results. Finally, the paper is concluded by Section 5.

2 PROBLEM SPECIFICATION

We consider a finite set of n stationary wireless sensor nodes, and we make the following assumptions. We suppose that the network is connected. Further, we assume that all nodes are aware of their locations through GPS or any other localization technique [4]. Finally, each node is aware of its one-hop neighbors and their corresponding locations.

Our objective is to start from any node and be able to traverse the entire network using one single packet. This latter must jump sequentially from node to node and browse the network while reducing communications to the extent possible. That is, minimizing or avoiding the visit of any node more than once. Also, in order to reduce communications, the next hop of the packet must be determined locally by each traversed node using only its local pre-collected one-hop neighbors table (no extra communications or collaboration between nodes should be required). In simple words, the problem that we are trying to solve can be boiled down to a distributed graph traversal. We recall that perfectly, a network with n connected nodes, should be traversed using exactly $n-1$ communications. Thus, theoretically, the path must cross every node precisely once. But, realistically, not every graph or network contains such optimal Hamiltonian path [14], and even if it does, determining that path constitutes an NP-complete problem [6].

3 PROPOSED SOLUTION

This section describes the proposed algorithm and details its behavior through illustrative examples.

3.1 Traversal tool

The whole operation of our proposal is based on a geometric form called *rolling-ball* (or *rolling-disc*) [10]. To summarize, a rolling-ball is a virtual circle that is hinged at a node and must be empty of any other node. Figure 1 shows a rolling-ball hinged at node N_1 with $c_1 \in \mathbb{R}^2$ as its center and $r/2$ as its radius (where r is the communication range of nodes [2]).

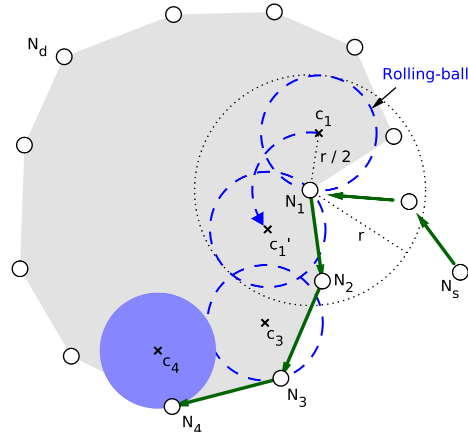

Figure 1: Rolling-ball.

Originally the rolling-ball has been used as a solution for the void problem encountered in geographic routing [10]. For instance, as demonstrated in Figure 1, in order to reach the destination N_d, node N_1 which is a local minimum creates a rolling-ball and spins it counterclockwise. The first touched neighbor, i.e., node N_2, in its turn, spins the received rolling-ball and determines the next hop. The process is repeated at each visited node until the greedy routing is resumed or until the whole boundary is traversed.

In this paper we make use of the rolling-ball as a network traversal tool. The reason behind this option resides in the fact that the rolling-ball is localized and other than the one-hop neighbors table of its owner, it does not require any other information to move to the next node. The second reason behind this choice is to guarantee the visit of all nodes in the network.

3.2 Initialization of the algorithm

In our algorithm, the node that launches the serial processing, which we refer to as the *trigger node*, can be any node in the network. Saying that the trigger node can be anywhere and knowing that the rolling-ball must be all the time empty of nodes, signifies that two cases are possible: the trigger node can be external or internal. The definition of external and internal nodes is given as follows:

Definition 3.1. External and internal Node
Given a set of wireless nodes with a communication range r, a node N_i is said to be external (resp. internal), iff at least one rolling-ball (resp. no rolling-ball) with radius $r/2$ can be hinged at this node.

Definition 3.1 simply means that an internal node cannot hold a rolling-ball while an external one can. Therefore, as Algorithm 1 shows, an external trigger node is considered as the starting point of the traversal, whereas an internal one is not. Actually, in the case of an internal trigger node, a starting point must be determined. In order to efficiently find a starting point (i.e., a random external node in the network), several techniques can be utilized. The goal is to find an external node as quickly as possible, because minimizing communications reduces the overall required time and energy.

Algorithm 1 Initialization

```
1: if (current_node.isExternal()) then
2:   call Algorithm 2;
3: else
4:   // current_node is internal
5:   Calculate boundary_point coordinates;
6:   Find nearest neighbor to boundary_point;
7:   Send init_packet to this neighbor;
8:   Upon receiving init_packet: current_node calls
     Algorithm 1;
9: end if
```

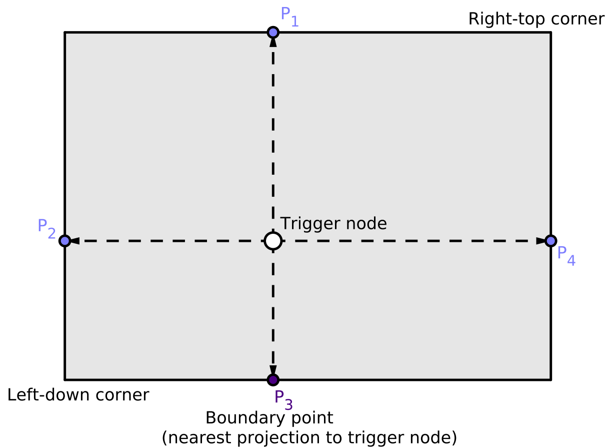

Figure 2: Boundary point determination by internal trigger node.

The technique used to find an external node may differ according to the considered scenario. To illustrate this, let us assume that the field in which nodes are deployed is a rectangle. And let us assume that this rectangle is defined by two points: *left-down* and *right-top* corners. In this situation, the best way to find an external node is to aim the external boundary of the network. To do so, as Figure 2 shows, the internal trigger node calculates its perpendicular projection on each side of the rectangle. Once calculated, the internal trigger node picks the closest perpendicular projection and determines the nearest neighbor to this point. Finally, as shown in Algorithm 1, an initialization packet has to be sent to this neighbor. Upon receiving the initialization packet, the destination node resumes the execution of Algorithm 1 (i.e., it checks if it is external or internal, and acts accordingly).

Note that the initialization packet does not have always to go all the way to the external boundary of the network. For example, if a hole is encountered halfway (i.e., an external node has been found), this condition is sufficient to stop the execution of Algorithm 1.

3.3 Algorithm's key idea

The idea of our traversal algorithm is to launch a rolling-ball (processing packet) and let it visit the network node by node (Figure 3). In order for this technique to work, initially, all nodes must be *unmarked* and each time a node is traversed, it has to be *marked.* In addition to that, in order to correctly determine the next hop, each node must keep track of its unmarked neighbors. In fact, thanks to the broadcast communication model used in wireless networks, this process does not require any additional communications or overhead other than the processing packet. Because, when a node forwards the processing packet to the next hop, all its neighbors can receive it and hence can update their neighbors table (locally mark the source of the received packet).

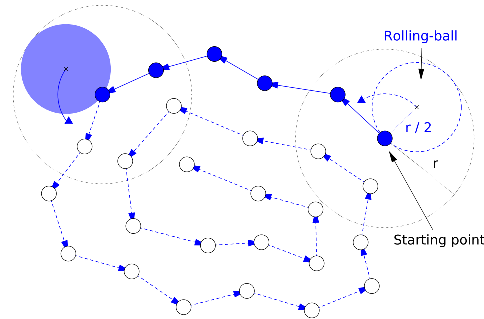

Figure 3: Traversal process using the rolling-ball.

3.4 Connectivity issue

The main issue faced by our distributed algorithm is ensuring the connectivity of the unmarked nodes while saving energy and time. In general, the problem arises in the case where a node that is essential for the unmarked nodes connectivity, marks itself. For example, as Figure 4 shows, if node N_6 and the subsequent nodes in the path mark themselves, data processing will end at node N_{15}, while nodes N_{16}, N_{17} and N_{18} have not been visited yet. We underline that processing termination is detected at a node when the neighbors of this latter have all been marked.

To solve the connectivity problem, we have introduced two new states for nodes. Thus, each node will have four possible states: unmarked, potential-cut, actual-cut, and marked. The unmarked, potential-cut, and actual-cut nodes are considered as **alive**, while the marked ones are seen as dead (they do not participate in the traversal process).

To keep things clear, we are going to start by introducing the concept of actual-cut nodes. As we go along, we will explain why the concept of potential-cut nodes has been introduced.

Definition 3.2. Actual-cut node
An actual-cut node in a connected network is a node whose removal (marking) disconnects the set of alive nodes into two or more sub-sets.

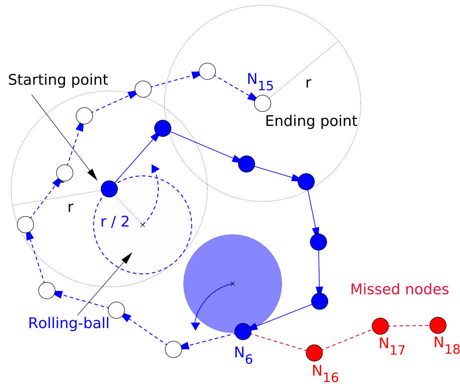

Figure 4: Connectivity issue

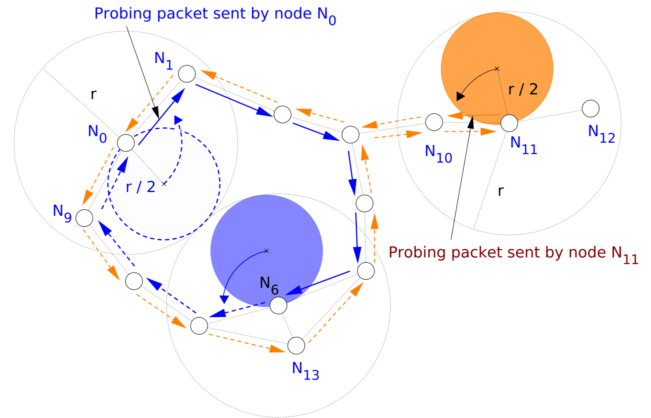

Figure 5: Concept of actual and potential-cuts

The goal behind determining the actual-cuts is to avoid network partitioning and hence ensuring the traversal of all nodes. For example, in Figure 4, node N_6 and the subsequent nodes in the path (except node N_{15}) are all actual-cuts and must remain involved in the traversal process. Once an actual-cut is no longer needed (to ensure the connectivity), it passes to the *marked* state.

In fact, having a network topology overview, it is easy to correctly determine the actual-cuts. For example, as Figure 5 shows, having an overview, it is easy to say that node N_{11} is an actual-cut while N_0 is not. As a matter of fact, being able to correctly determine the actual-cuts, the steps described above are sufficient to traverse any connected network while considerably saving both time and energy. Nevertheless, the assumption of having a network topology overview, holds only in two cases: (1) the algorithm is executed by nodes that have each a global knowledge about the network topology or (2) the algorithm is executed in a centralized fashion by one entity possessing a global knowledge. Despite their advantages or disadvantages, these two cases are out of scope of this paper. In the remainder, we treat only the distributed version of the algorithm in which each node must rely only on its immediate one-hop neighbors. For instance, as shown in Figure 5, all that node N_0 and node N_{11} are aware of, is that they have two neighbors that cannot communicate with each other without their help. That is to say, both nodes see the same thing.

With that being said, we conclude that a node cannot locally determine if it is an actual-cut. In other words, Definition 3.2 cannot be fulfilled locally and necessitates communications and collaboration between nodes. A first intuitive solution for determining the actual-cuts consists of using probing (or control) packets. Before explaining this solution, let us first define what is a potential-cut.

Definition 3.3. Potential-cut node
A potential-cut node in a connected network is a node whose removal (marking) disconnects the set of its one-hop alive neighbors into two or more sub-sets.

For example, in Figure 5, nodes N_0 and N_{11} are potential-cuts while node N_6 is not (i.e., it can be marked). We underline that unlike the actual-cuts case, a node can locally (i.e., using its one-hop neighbors table) determine if it is a potential-cut. Note also that an actual-cut is necessarily a potential-cut but the opposite is not true.

For a potential-cut to determine if it is an actual-cut, this node can issue a rolling-ball probing packet that due to its nature will make a tour and come back to it. When the probing packet gets back to its sender, this latter based on which side the packet has come back from, can properly decide if it is an actual-cut. As a simple illustrative example, let us consider Figure 5. In order for node N_0 and node N_{11} to decide if they are actual-cuts, these two nodes send a probing packet and wait to receive it back. Once this is done, N_{11} concludes that it is an actual-cut because the probing packet got back from the same side (there is no other path connecting its neighbors). Whereas, N_0 concludes that there is another path connecting its neighbors and hence it can mark itself and pass the processing packet to the next hop.

Without entering into details, we can say that from one side, the solution of probing packets solves the connectivity problem but from the other side, it violates the localized design of our distributed algorithm and forces it to spend an extra overhead (i.e., extra time and energy). We recall that our objective is to propose a distributed algorithm that excludes any use of control packets and relies only on the processing packet and the local one-hop neighbors table of each node.

The solution that we propose to determine the actual-cuts makes use of the processing packet itself. This latter besides its ordinary role, plays the role of a probing packet, which considerably reduces communications and saves both energy and time. The solution can be explained briefly as follows. If a node detects (locally) that it is a potential-cut then instead of issuing a probing packet, this node changes its status to *potential-cut* and forwards the processing packet. Given its nature, the processing packet will certainly get back to its sender. Once revisted, the node decides whether (1) it becomes actual-cut, (2) remains as a potential-cut, or (3) marks itself (no longer needed for the traversal). Our serial processing technique is summarized in Algorithm 2, which can be divided into three big steps:

(1) **Processing packet received for the first time:** (*statements 3 to 9*). When a node receives the processing packet for the

Algorithm 2 Serial data processing

```
1:  The starting point (which must be an external node):
      change its state to potential-cut or marked;
      forwardProcessingPacket();
2:  Upon receiving the processing packet, the destination node
    executes the following steps:
3:  if (current_node.state == unmarked) then
4:     // Processing packet received for the first time.
5:    process_data();
6:    change_state(); // to potential-cut, actual-cut or marked.
7:    forwardProcessingPacket();
8:    return;
9:  end if
10:
11:  // Processing packet has already been received.
12: if (not current_node.isPotentialCut()) then
13:   // current_node is no longer a potential-cut
14:   current_node.state = marked;
15:   forwardProcessingPacket();
16: else
17:   if (current_node.state == actual_cut) then
18:     call Algorithm 4;
19:   else
20:     // current_node.state == potential_cut
21:     call Algorithm 3;
22:   end if
23: end if
```

Algorithm 3 Code executed by potential-cuts

```
1:  if (current_node.canBecomeActualCut()) then
2:    current_node.state = actual_cut;
3:    forwardProcessingPacket();
4:    return;
5:  end if
6:  switch (previous_hop.state){
7:    case marked: {
8:      if (processing_pkt came back from the same set
9:      and this set still contain alive nodes) {
10:         current_node.state = actual_cut;
11:     }
12:     break;
13:   }
14:   case potential_cut: {
15:     if (processing_pkt came back from the same set) {
16:         current_node.state = actual_cut;
17:     }
18:     else {
19:         Cut link with previous_hop;
20:         Change state to marked or potential-cut;
21:         sendCycleBreakPacket(previous_hop);
22:         return;
23:     }
24:   }
25: }
26: forwardProcessingPacket();
```

first time, it executes the required task (query), changes its state to *potential-cut*, *actual-cut* or *marked*, and finally forwards the processing packet to the next hop. We mention that a node can become actual-cut if it is potential-cut just to ensure the connectivity of other actual-cuts. In other terms, a potential-cut can become actual-cut, if without considering its actual-cut neighbors, it is not a potential-cut.

(2) **Processing packet has already been received and node is no longer a potential-cut:** (*statements 12 to 15*). Being no longer a potential-cut means also that the node cannot be either an actual-cut. In such a case, the current node marks itself and forwards the processing packet to the next hop. We underline that `isPotentialCut()` method applies Definition 3.3.

(3) **Processing packet has already been received and node is still potential-cut:** (*statements 16 to 23*). Being a potential-cut according to `isPotentialCut()` method, means that the current node can be also an actual-cut. In this situation, the node refers to its local stored state and executes the corresponding code. The steps executed by potential-cuts and the ones executed by actual-cuts are described respectively in Algorithm 3 and Algorithm 4.

As mentioned earlier, when a potential-cut receives the processing packet back, it can decide to whether become actual-cut, remain potential-cut or mark itself. In fact, a potential-cut determines its next state according to two factors: the state of its immediate neighbors, and the side (set) from which the processing packet has come back. The code executed by a potential-cut (Algorithm 3) can be divided into three main steps:

(1) **Current node can become actual-cut:** (*statements 1 to 5*). As mentioned earlier, a node can become actual-cut if it is potential-cut only to ensure the connectivity of other actual-cuts. In other words, a potential-cut can become actual-cut, if without considering its actual-cut neighbors, it is not a potential-cut.
If the potential-cut can become actual-cut, it changes its state accordingly, forwards the processing packet, and stops the execution of Algorithm 3. In the opposite case (i.e., node is not an actual-cut), the execution of Algorithm 3 continues according to the previous-hop state (i.e., marked or potential-cut).

(2) **Current node cannot become actual-cut and previous-hop is a marked node:** (*statements 7 to 13*). To determine its state, besides the state of the previous-hop, in this case, the current node relies also on which set the processing packet has come back from. If the processing packet came back from the same set it was sent to, and this set still contain alive neighbors, then the current node is ensuring the connectivity of this set. So, it changes its state to actual-cut (statement 10) and forwards the processing packet (statement 26). The opposite case means that, (1) the processing

packet came back from the same set, but this set does not contain any alive neighbors, or (2) processing packet came back from a different set. In both cases, the current node remains as a potential-cut and forwards the processing packet to the next hop (statement 26).

(3) **Current node cannot become actual-cut and previous-hop is a potential-cut:** (*statements 14 to 24*). Here also, to determine its state, besides the state of the previous-hop, the current node relies also on which set the processing packet has come back from. Actually, if the processing packet came back from the same set (to which it was sent), then there is no need to check if this set still contain alive neighbors because the previous-hop is alive. In such a case, the current node changes its state to actual-cut (statement 16) and forwards the processing packet (statement 26). The opposite case (i.e., the processing packet came back from a different set) means that there is a cycle that must be broken.
To well illustrate the cycle break process, let us consider the example depicted in Figure 6(a). In this example, node N_0 which is the trigger node changes its state to potential-cut and forwards the processing packet. This latter makes a tour and comes back to N_0 from N_7 which is also a potential-cut. In this case, to break the cycle, as Figure 6(b) shows, node N_0 executes the following steps:
- It cuts its link with node N_7
- After cutting the link with the previous-hop, N_0 cannot be marked, so it remains as a potential-cut.
- The last step consists of sending a cycle break packet to the previous-hop. This packet, as Figure 6(b) shows, orders the previous-hop and other nodes to cut the proper links.

Once the cycle has been broken, the previous-hop, i.e., node N_7 continues the traversal by forwarding the processing packet.

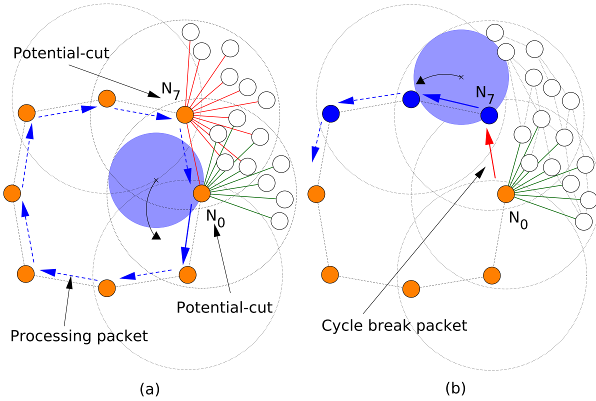

Figure 6: Example of cycle break.

In the previous paragraphs, we have treated the cases where the previous-hop can be marked or potential-cut, but we did not talk about the case of a previous actual-cut. We mention that even if the previous-hop is an actual-cut then this does not mean that the current node can become actual-cut. In fact, if the previous-hop is an actual-cut and the current node cannot become actual-cut, the execution of Algorithm 3 will jump to statement 26 (i.e., current node remains potential-cut and forwards the processing packet).

The fact that a node is an actual-cut does not mean that it will remain in this state. Similarly as a potential-cut, when an actual-cut receives the processing packet back, it decides to whether remain actual-cut, become again a potential-cut or mark itself. The next state of an actual-cut is determined by the state of its immediate neighbors and the set from which the processing packet has come back. The steps followed by an actual-cut when it receives back the processing packet, are summarized in Algorithm 4.

Algorithm 4 Code executed by actual-cuts

```
1:  switch (previous_hop.state){
2:    case marked: {
3:      if (set for which current_node is actual-cut
4:      contains no alive nodes){
5:          current_node.state = potential_cut;
6:      }
7:      break;
8:    }
9:
10:   case potential_cut: {
11:     if (processing_packet came back from a different
12:     set than the one it was sent to){
13:         Cut link with previous_hop;
14:         Change state to marked or potential-cut;
15:         sendCycleBreakPacket(previous_hop);
16:         return;
17:     }
18:   }
19:
20:   case actual_cut: {
21:     if (processing_packet came back from a different
22:     set than the one it was sent to){
23:         current_node.state = potential_cut;
24:     }
25:     break;
26:   }
27: }
28:
29: forwardProcessingPacket();
```

4 SOLUTION EVALUATION

Given the fact that numerous research works have already shown the superiority of serial algorithms over structure-based ones [3, 11], in this paper, we consider only serial algorithms. More exactly, we chose to compare our solution with three other serial algorithms, namely: the Peeling Algorithm (PA) [11], the Greedy and Boundary Traversal algorithm (GBT) [3], and the well-known depth-first search algorithm (DFS).

In summary, the Peeling Algorithm uses a *curved-stick* [12] as a network traversal tool and must start from the external boundary of the network. The term peeling came from the fact that each time, the visited node is removed from the external boundary. As

Table 1: Simulation parameters

Parameter	Value(s)
Number of nodes	100, 150, 200, ..., 500
Deployment field	1000 x 1000 m^2
Nodes deployment	Uniform
Location of the trigger node	Random
Transmission range of nodes	150 m
Processing packet size	50 Bytes

regards GBT, this algorithm operates in two alternative distinct modes: greedy forwarding and boundary traversal. At first, the path is extented towards the unvisited nodes as long as possible, and when there is no more left unvisited neighbors, the boundary traversal will be launched. During boundary traversal, if a non-visited node is encountered, the greedy mode will be resumed. This way, GBT switches between the two modes until visiting all nodes. DFS extends the path as far as possible towards the unmarked nodes, and when it gets stuck at some node (all neighbors have been marked), it goes back to the parent of that node and so forth.

4.1 Evaluation metrics and simulation settings

To evaluate the performance of all four algorithms, we have opted for OMNeT++ along with Castalia [1]. The first considered evaluation metric is the total number of packets required to process data. In fact, given the serial nature of the evaluated algorithms, this metric is the most important because it affects the two other metrics, which are the time and energy necessary for data processing.

By processing time, we mean the time elapsed between the instant when the trigger node launches data processing and the moment when it receives the processed data. In the chosen simulation scenario, while being traversed one by one, nodes have been queried to compute the *average sensed value* in the network.

As regards processing energy, this metric represents the total energy dissipated by all nodes to process data. The energy consumed by the radio of each node has been estimated using the model proposed by Heinzelman [7]. In this well-known energy consumption model, in order to send a k-bit packet a distance d, the radio consumes $E_{TX}(k,d) = E_{\text{elec}} * k + \epsilon_{\text{amp}} * k * d^2$, and it consumes $E_{RX}(k) = E_{\text{elec}} * k$ to receive this packet. Where:

- E_{elec} = 50 nJ/bit: energy for running the transmitter/receiver circuitry.
- ϵ_{amp} = 100 pJ/bit/m^2: energy for running the transmitter amplifier.

The parameters used in the simulations are summarized in Table 1.

4.2 Evaluation results

The number of packets spent by GSS (proposed algorithm), PA (Peeling Algorithm) [11], GBT (Greedy and Boundary Traversal) [3] and DFS (Depth-First Search) to process data are shown in Figure 7. This figure shows also the number of packets that an optimal algorithm (if it existed) would have spent. As a matter of fact, the number of packets used by both, the optimal theoretical algorithm and DFS, have been depicted to serve as a reference to measure the effectiveness of the evaluated serial algorithms in terms of communications. We recall that an optimal algorithm traverses a network of n connected nodes using $n-1$ packets, whereas DFS, due to its backtracking behavior, requires $2*(n-1)$ packets.

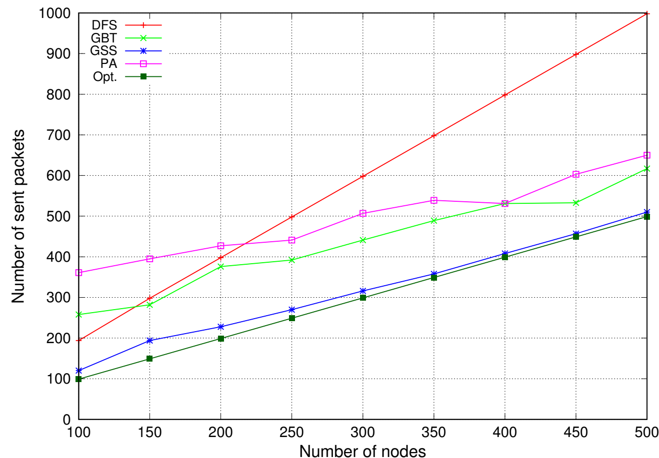

Figure 7: Required communications (sent data and control packets).

As Figure 7 demonstrates, our proposal is clearly superior. Actually, GSS outperforms all the other algorithms and spends a number of packets that is very close to the optimal algorithm's number. We can say that the denser the network, the better the performance of GSS will be. In low density networks, GSS is slightly different than the optimal algorithm because in such networks, cycles have a high occurrence probability. In such scenarios, due to the limited knowledge of nodes, the processing packet has to make a tour in order to be able to detect the existence of a cycle. In spite of that, the performance of GSS is not drastically affected as it is the case for PA and GBT. In fact, as demonstrated in Figure 7, in sparse networks, even DFS outperforms PA and GBT.

The good performance of GSS comes from the idea used to solve the connectivity issue and overcome the limited knowledge of nodes. This solution attributes a twofold role to the processing packet: at the same time it serves as a network traversal tool and as a probing packet.

The time and energy consumed by the four algorithms to process data are depicted respectively in Figure 8 and Figure 9. Given its sequential behavior, the more a serial algorithm requires packets, the longer it will take to finish and the more energy it will consume. As Figure 8 and Figure 9 show, since GSS requires less communications, it outperforms the other algorithms it in terms of processing time and energy.

5 CONCLUSION AND FUTURE WORK

Lately, serial in-network processing has proven its efficiency in large-dense networks. However, due to the complexity of path building process, the proposed solutions are not optimal and require an extra overhead. To overcome this drawback, we have proposed in this paper a new scalable serial processing algorithm that reduces

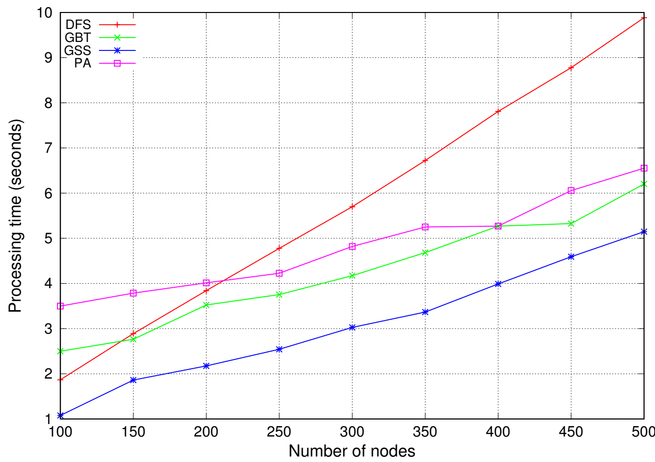

Figure 8: Processing time.

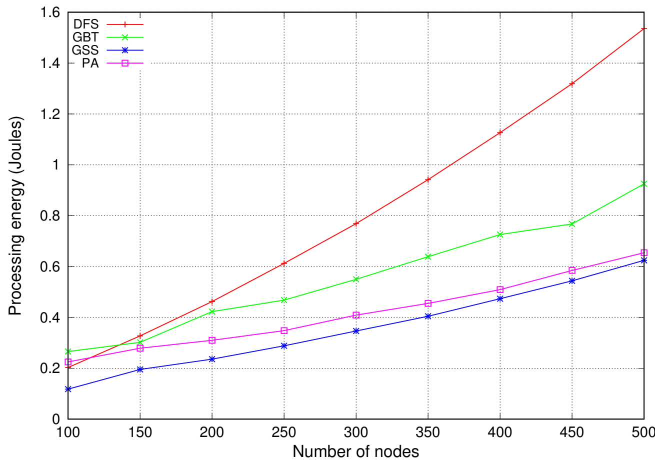

Figure 9: Processing energy.

further the processing time and energy. The extensive simulations have demonstrated that the proposed solution approximates the optimal number of hops. The obtained results have confirmed also that the proposed algorithm traverses all nodes in the network and does not loop.

As a future work, we plan to formally prove the correction of the proposed algorithm. More specifically, prove that (1) it is free of looping, and (2) it visits all connected nodes. First, in order to prove that GSS terminates and does not loop indefinitely, we base on the fact that all cycles (which can be generated due to the use of potential and actual-cuts) are detected and properly removed. Second, in order to prove that GSS visits all nodes, we base on the fact that the network is initially connected and its connectivity is maintained throughout the traversal.

REFERENCES

[1] OMNeT++ : Simulation Environment. http://www.omnetpp.org/. (????).

[2] Azzedine Boukerche, Xin Fei, and Regina B Araujo. 2007. An optimal coverage-preserving scheme for wireless sensor networks based on local information exchange. *Computer Communications* 30, 14 (2007), 2708–2720.

[3] A Boukerche, A Mostefaoui, and M Melkemi. 2016. Efficient and robust serial query processing approach for large-scale wireless sensor networks. *Ad Hoc Networks* 47 (2016), 82–98.

[4] Azzedine Boukerche, Horacio Oliveira, Eduardo F. Nakamura, and Antonio A. F. Loureiro. 2007. Localization systems for wireless sensor networks. *IEEE Wireless Communications* 14, 6 (2007), 6–12.

[5] Elena Fasolo, Michele Rossi, Jorg Widmer, and Michele Zorzi. 2007. In-network aggregation techniques for wireless sensor networks: a survey. *IEEE Wireless Communications* 14, 2 (2007).

[6] Michael R. Garey and David S. Johnson. 1983. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman, New York.

[7] Wendi B. Heinzelman, Anantha P. Chandrakasan, and Hari Balakrishnan. 2002. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications* 1, 4 (2002), 660–670.

[8] Mo Li, Yajun Wang, and Yu Wang. 2011. Complexity of data collection, aggregation, and selection for wireless sensor networks. *IEEE Trans. Comput.* 60, 3 (2011), 386–399.

[9] Stephanie Lindsey and Cauligi S Raghavendra. 2002. PEGASIS: Power-efficient gathering in sensor information systems. In *Aerospace conference proceedings, 2002. IEEE*, Vol. 3. IEEE, 3–3.

[10] Wen-Jiunn Liu and Kai-Ten Feng. 2009. Greedy routing with anti-void traversal for wireless sensor networks. *IEEE Transactions on Mobile Computing* 8, 7 (2009), 910–922.

[11] Ahmed Mostefaoui, Azzedine Boukerche, Mohammed Amine Merzoug, and Mahmoud Melkemi. 2015. A scalable approach for serial data fusion in Wireless Sensor Networks. *Computer Networks* 79 (2015), 103–119.

[12] Ahmed Mostefaoui, Mahmoud Melkemi, and Azzedine Boukerche. 2012. Routing through holes in wireless sensor networks. In *Proceedings of the 15th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems.* ACM, 395–402.

[13] Swapnil Patil, Samir R Das, and Asis Nasipuri. 2004. Serial data fusion using space-filling curves in wireless sensor networks. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on.* IEEE, 182–190.

[14] Michael G. Rabbat and Robert D. Nowak. 2005. Quantized incremental algorithms for distributed optimization. *IEEE Journal on Selected Areas in Communications* 23, 4 (2005), 798–808.

[15] Ramesh Rajagopalan and Pramod K. Varshney. 2006. Data Aggregation Techniques in Sensor Networks: A Survey. *IEEE Comm. Surveys & Tutorials* 8 (2006), 48–63.