THESIS

COORDINATE REPAIR AND MEDIAL AXIS DETECTION IN VIRTUAL COORDINATE BASED SENSOR NETWORKS

Submitted by

Gunjan S. Mahindre

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2014

Master's Committee:

Advisor: Anura Jayasumana

J. Rockey Luo
Yashwant Malaiya

ABSTRACT

COORDINATE REPAIR AND MEDIAL AXIS DETECTION IN VIRTUAL COORDINATE BASED SENSOR NETWORKS

Wireless Sensor Networks (WSNs) perform several operations like routing, topology extraction, data storage and data processing that depend on the efficiency of the localization scheme deployed in the network. Thus, WSNs need to be equipped with a good localization scheme as the addressing scheme affects the performance of the system as a whole. There are geographical as well as Virtual Coordinate Systems (VCS) for WSN localization. Although Virtual Coordinate (VC) based algorithms work well after system establishment, they are hampered by events such as node failure and link failure which are unpredictable and inevitable in WSNs where sensor nodes can have only a limited amount of energy to be used. This degrades the performance of algorithms and reduces the overall life of the network. WSNs, today, need a method to recover from such node failures at its foundation level and maintain its performance of various functions despite node failure events. The main focus of this thesis is preserving performance of virtual coordinate based algorithms in the presence of node failure.

WSNs are subject to changes even during their operation. This implies that topology of the sensor networks can change dynamically throughout its life time. Knowing the shape, size and variations in the network topology helps to repair the algorithm better. Being centrally located in the network, medial nodes of a network provides us with information such as width of the network at a particular cross-section and distance of network nodes from boundary nodes. This information can be used as a foundation for applications such as network segmentation, VC

system implementation, routing scheme implementation, topology extraction and efficient data storage and recovery. We propose a new approach for medial axis extraction in sensor networks. This distributed algorithm is very flexible with respect to the network shape and size. The main advantage of the algorithm is that, unlike existing algorithms, it works for networks with low node degrees.

An algorithm for repairing VCS when network nodes fail is presented that eliminates the need for VC regeneration. This helps maintain efficient performance for all network sizes. The system performance degrades at higher node failure percentages with respect to the network size but the degradation is not abrupt and the system maintains a graceful degradation despite sudden node failure patterns. A hierarchical virtual coordinate system is proposed and evaluated for its response to network events like routing and node failures. We were also able to extract medial axis for various networks with the presented medial axis detection scheme. The networks used for testing fall under a range of shapes and an average node degree from 3 to 8. Discussions over the VC repair algorithm and the novel medial axis extraction scheme provide an insight into the nature of proposed schemes. We evaluate the scope and limitations for VCS repair algorithm and medial axis detection scheme. Performance of the VC repair algorithm in a WSN is evaluated over various conditions simulated to represent a practical node failure events to gauge the system response through routing percentage and average hop count over the network. We compare the results obtained through our medial axis detection scheme with existing state-of-the-art algorithm. The results show that this scheme overcomes the shortcomings of the medial axis detection schemes. The proposed medial axis detection technique enables us to extract the information held by a medial axis of a sensor network. The VC repair algorithm and the new

medial axis extraction scheme perform very efficiently to make a WSN tolerant of node failure events.

ACKNOWLEDGMENTS

This successful completion of my thesis is a symbol of the excellence of people who supported me and helped me enormously. The most important person behind this successful attempt is my adviser, Prof. Anura Jayasumana. He gave me the opportunity of working with him. He believed in my abilities and guided me so that I put my best efforts. He has worked with me for my best upbringing as a father and as a teacher. He encouraged me during the tough times of this journey. During my work with him, I learnt to be an independent and efficient researcher. Thank you Prof. Anura Jayasumana, for believing in my abilities and giving me your valuable guidance. I cannot thank Prof. Jayasumana enough for the lessons I have learnt from him in my academic career and my life.

I would like to gratefully and sincerely thank, other committee members: Prof. Rockey Luo and Prof. Yashwant Malaiya. Thank you very much for giving me your time. I would also like to thank Prof. Liuqing Yang, Prof. Edwin Chong, Prof. Ronald Sega, and Prof. Anthony Maciejewski. I would like to specially thank Prof. Michael Kirby and Prof. Kimiharo Noguchi for their guidance on the data processing skills I learned with them and changing my view towards Mathematics and Statistics.

My dearest gratitude to my colleagues: Vidarshana Bandara, Negar Musharaff, Ali Bound and Chepchumba Limo, in the CNRL lab for sharing their time and guidance on my work. Thank you for being there as my friends and my critiques. I would also like to thank Dulanjalie Dhanapala, Pritam Shah and Yi Jiyang who helped me time to time with my academic doubts.

There are several other important people that I would like to thank who have helped me indirectly in my work with their constant support and encouragement they gave me. My undergraduate professors, Prof. Sachin Takale and Prof. Surendra Badwaik, who have molded me during the beginning of my career, believed in my abilities and guided me during the most important years of my life. I am very thankful to them for their teachings and life lessons. My heartiest gratitude to Abhishek Wadurkar, my dearest friend, for being there whenever I needed a true friend and supporter. Thank you for motivating me to give my best. I would like to thank my friends who have helped me in countless ways and have always been there for me in my good and bad times, it would have been much more difficult without you all.

Finally, and most importantly, I would like to thank my parents, and my beloved brother Aniket for their love and support which always strengthen me and encourages me to achieve the best I can. I dedicate my work to them. Thank you very much for being there!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

A Wireless Sensor Network (WSN) is a mesh of small, wirelessly interconnected sensor nodes. They monitor environmental or physical conditions in their surroundings. The sensing devices are interconnected to each other in large numbers in order to sense events, collect, exchange and process data in a distributed fashion and make collaborative decisions.

The sensor nodes are deployed either in a random fashion or with uniform density. Moreover, being dispersed over a wide area in the network, identifying the location of the node becomes crucial so as to determine the source of the data or the destination location where the data is to be routed. Localization is one of the most important research topics as the location information is typically useful for coverage, deployment, routing, location service, target tracking, and rescue.

While the Global Positioning System (GPS) is one of the most popular positioning technologies which is widely accessible, the drawbacks such as it being of high cost and energy consuming makes it difficult to install in every node. To reduce the energy consumption and cost, only a few of the nodes which are called beacon nodes may contain the GPS modules [1]. The rest of the nodes could obtain their locations through localization method. The process of nodes estimating the unknown positions within the network is referred to as node self-localization. However, this goal of self-localization can also be obtained by using Virtual Coordinates (VCs) for sensor node localization. Virtual coordinates provide a very efficient way of localizing the sensors instead of using Geographical Coordinates (GC) which are much heavier from the computation point of view. In VC based addressing, a set of anchor nodes are

selected as landmarks and relative distances from these set of anchor nodes, in terms of shortest hop counts, are used to define a node's virtual coordinates. Other than localization; operations such as routing, boundary detection, topology estimation, data analysis and distributed decision making can also be efficiently performed in a system with VCs. It allows the WSN to act as an independent system placed in the physical surroundings. Therefore, VC based localization schemes play a vital role in WSNs.

1.1 Motivation and Problem Statement

Sensor nodes, in general, have a limited battery life. They are aided to be alive until their energy drains. This - discharge of energy - can be termed as node failure. WSNs are often deployed in inhospitable environments. So node failure can also be caused by destruction of the nodes due to external events. In addition to that, few links in the network can be failure prone leading to dynamic changes in the network topology [2]. Links can behave as faulty links due to erroneous communication or blockage of signal. The failure of nodes can cause improper connectivity or even loss of connectivity for some nodes. As GCs can be obtained from devices such as GPS, they automatically update themselves with any changes in the location of system components. VCs, however, are sensitive to network connectivity and therefore have to be updated by the network autonomously. Thus we need energy-efficient and fault tolerant algorithms for network operation in such circumstances.

Consider a WSN deployed in a forest to monitor physical parameters such as humidity, temperature and wind speed. The system has a set of anchor nodes and the localization is based on the Virtual Coordinate schemes as discussed above.

Figure (1.1.a) shows an example of WSN deployed in a forest. The blue dots represent sensors on the trees while the red dots represent sensors damaged by forest fire. Figure (1.1.b) shows the actual remains of the connected sensor network.

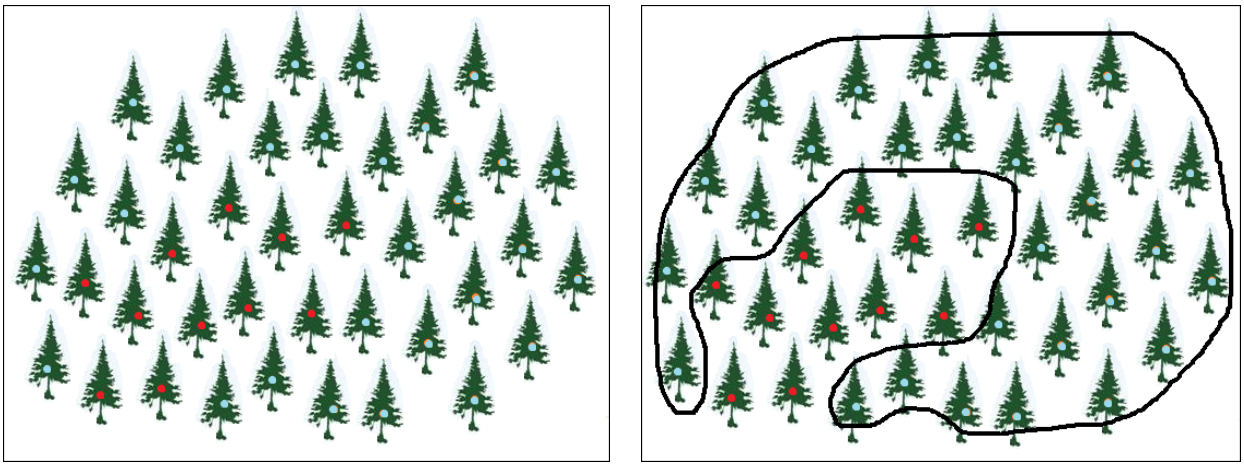


Figure 1.1: WSN deployed in a forest; (a) WSN deployed in a forest setting. Red nodes are damaged nodes, (b) Effective connected WSN due to node failure

A forest fire may destroy few of the sensor nodes from the WSN causing node failure. These failed or dead nodes can be the only nodes constituting the shortest path for few other nodes in the network. This implies that the dependent nodes lost their shortest routes to the anchors due to node failure rendering their VCs no longer valid.

We need to update VCs when such failure event(s) occur. If outdated and wrong VCs are used even after node failure, it might cause problems such as

- **Routing failure**: Loss of connection to a particular part of the network can cause permanent blockage for routing in that area. Wrong VCs can also cause false local minima and prevent routing beyond that point. Due to outdated VCs, now we might not have the shortest path to the destination causing inefficient routing within the network.

Thus, instead of taking a backtracking to reach the destination, the message might get stuck at a node which previously, was supposed to lead to the destination.

- **Excessive energy consumption**: As discussed above, few nodes can act as false sinks and the packets may get stuck at such nodes. Usually, routing mechanisms attempt to reroute the packet for a certain number of times or hops before calling the routing unsuccessful. These attempts, however, waste a lot of energy. Thus, inefficient routing can lead to unnecessary energy depletion of the sensor nodes causing quick failure of such nodes.
- **Wrong topology prediction**: There has been work done to derive the topology maps from VCs [3], [4], [5]. Affected VCs might lead to wrong topology results giving wrong information about node neighborhood and connectivity.
- **False data location**: When GCs are not available, VCs can be used to estimate tentative locations of wireless sensor nodes so that we can relate the data with the location. This is useful for applications such as indoor navigation. Lack of reliable VCs will make us correlate data with improper locations causing misinterpretation of available information.

On the other hand, a small modification in the network establishment algorithm for VC fault detection and recovery can make and keep the network as efficient as possible instead of rendering it useless. This motivated us to work on methods to update Virtual Coordinates in a WSN system affected by node failure.

Node failure and presence of faulty links in the network requires system repair for graceful degradation of the network. The more we know about the network, the better quipped we are to repair it in case of failure events. Medial axis of a network provides a perfect foundation to extract maximum information about the network. Medial axis of a network can

also be termed as the skeleton of the network. Figure (1.2) gives an example of network skeleton – medial axis of the network.

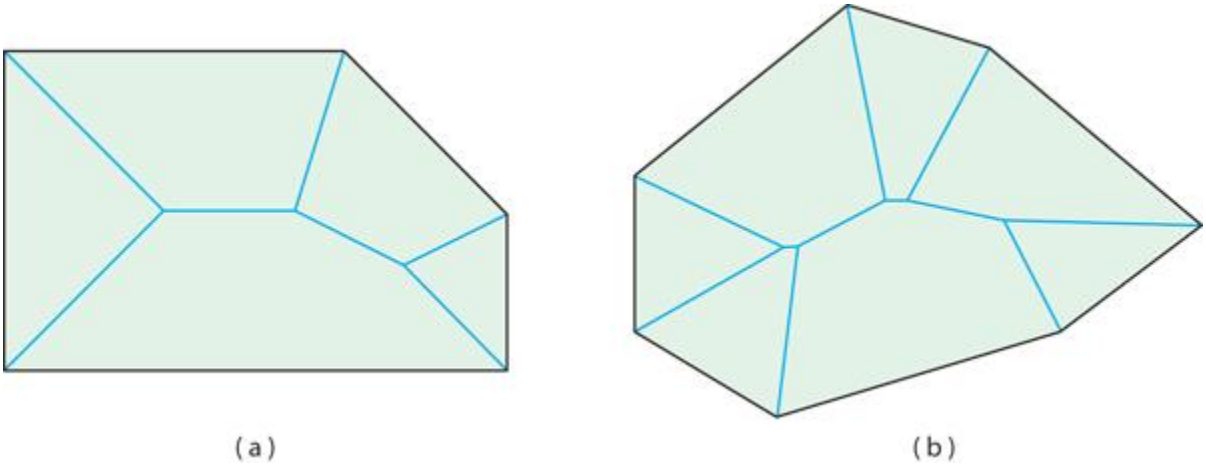


Figure 1.2: Illustrations of medial axes for various polygons; (a) Medial axis of polygon with 5 corners; (b) Medial axis of polygon with 7 corners

Knowledge of the medial axis of a network can prove very useful for other network functions that are carried out on WSNs. Some examples are network segmentation, routing, establishing VC system, topology extraction and data storage.

1.2 Contribution

This thesis contributes in two areas; VC repair post failure events and medial axis extraction.

In order to have a robust WSN, the WSN should be able to tolerate the failure of single or multiple nodes and adapt to or recover from such events on its own in an efficient way such that the VCs are updated to smallest correct values possible with minimum transfer of messages and thus minimum investment of energy for the required amendments in the system. We present a simple method to detect the nodes with incorrect VCs as well as repair the VCs of the affected

nodes in a network due to unpredicted node failure events. The algorithm is distributed and works locally. The algorithm has been explained through a systematic pseudocode, illustration on a sample WSN. We also study the effect of node failure on the system through a discussion. Performance of the system has been meticulously tested on various systems for different node failure percentages to test the relative communication efficiency and percentage routability of the system before node failure, after node failure and after system recovery. The results show that our VC repair algorithm works effectively to improve the system performance after node failure. Although it is not always as good as the system before node failure, the algorithm helps the system to degrade gracefully.

As compared to working towards developing an algorithm that tries to improve a single application over the network; repairing the system so that it will recover the foundation of all the applications helps to improve performance of all the system functions. This is what we had aimed to achieve through VC repair algorithm. It has been tested and proved that VC repair helps improve routability and communication efficiency; similarly the algorithm can be tested for other applications.

This thesis also presents a novel definition and thus a novel way to detect medial axis nodes in a WSN. Using only the connectivity information of the network, we are able to extract a connected medial axis for the network in not more than two rounds of algorithm. We propose a distributed algorithm that works locally with very low memory requirements and low computation cost. Medial Axis Detection algorithm for Low Degree networks (MALD) works for low degrees as well as higher degrees. The computation cost lowers as the node degree increases as shorter paths are available. MALD works for degrees as low as 3 ~ 4. It performs better than other medial axis detection algorithms proposed till date, in aspects such as

communication cost, scope of the network shape and algorithm complexity. Being able to detect medial nodes of a low degree network makes it possible to utilize these nodes in several applications such as data storage, routing, topology generation and network segmentation; which have further applications of them.

The focus of this thesis is on node failure recovery of the VCs for better system performance. The algorithm provides significant improvement over the routability of the network after node failure. It also helps to maintain graceful degradation of the network post node failure. The algorithm has also been proved to be computationally efficient as compared to the conventional network amendment method. Properties of VC repair algorithm such as low communication cost, energy efficiency and low memory requirements contribute to increased lifetime of the WSN whilst providing a foundation for better system performance.

1.3 Outline

Rest of the thesis is organized as follows. Chapter 2 reviews the related work in the area of node failure amendment techniques and in the field of medial axis detection in sensor networks. Chapter 3 describes the algorithm for VC repair and analyses the performance of the proposed algorithm under several node failure conditions. Then in Chapter 4, we discuss the novel medial axis detection scheme and test the performance of the scheme for different network shapes and sizes. Finally, Chapter 5 summarizes our work and lists the conclusions followed by our future interests.

Chapter 2

RELATED WORK

WSNs rely on VCs for numerous functions. This is so because of the low energy requirement and high information extraction capability of VCs. VCs, in a way, help to lengthen the life time of sensor networks. Functions such as routing and localization are mainly dependent on the information available from the VCs of the network nodes. However, network events such as node failure and link failure hamper the performance of these function based protocols. Hence, there is enormous amount of research work being carried out on protocols that are robust towards node failure. WSNs are a great source of absorbing information about the desired environment. But it is also important to know about the deployed network in order to maintain and repair the network if required. Since this thesis is based on VC repair algorithm and medial axis detection for a WSN, we will look at the work on node failure and medial axis extraction in sensor networks. We will also briefly study their advantages and their shortcomings.

2.1 On Node Failure and Virtual Coordinate Repair

Node failure events can lead to two types of faults in the sensor network system. After node failure, the network can get partitioned into two or more networks or the network might still be connected through few of its links maintaining a single connected component in the network. We concentrate on the cases where nodes unpredictably fail but the network does not lose its connectivity as a whole. We refer to the definition of ‘fault tolerance’ as the ability of a system to deliver a desired level of functionality in the presence of faults [8].

We have to detect and then repair the faults and prevent it from affecting the system performance as much as possible. Basically there are two types of fault detection techniques: self-detection and cooperative detection. The faults that can be detected by sensor nodes themselves by sensing the battery level or lack of response on a particular link, can be termed as self-detection of faults. On the other hand, if a large portion of the WSN is affected, it takes more than one sensor node to work together to detect and diagnose the problem [9]. Usually, the former type of fault detection is needed in WSNs.

A significant amount of work has been done in the field of WSNs in order to improve the fault tolerance of the system and extend the overall life of the WSN. Fault recovery in 1-hop WSNs is studied in [10]. The basic idea is to partition the sensor memory as data memory and redundant memory. The data memory is used to store sensed data and data recovered from failures of other sensor nodes. The redundant memory is used to store redundant data for future recovery. When a node failure is sensed, the node with its data backup covers for the location of the failed node and the data in the redundant memory is put into use as recovered data. The recovered data is then distributed among the other nodes of the network. The technique works with an overhead of (n+1)/n where there are n nodes in the network. A routing scheme that takes failure of a single sensor node from the sensor network into consideration is presented in [9]. The algorithm also works towards detecting and recovering from the loss by replacing the failed node. The replacement sensor facilitates fault tolerance by performing the functions of a failed node and reconnecting the lost sensors. However, this technique works only when the sensor failure is sparse and not in cluster.

The routing algorithm discussed in [11] watches out for the radio activity to detect failures when they occur. It then takes actions at the point of failure and works towards rerouting

the data through a different node if the second broadcast is not heard. It repairs the routing path without starting over the routing from the source node. Node failure is simulated by considering that each node has a 3% chance of suffering a failure and becoming unresponsive for a short time. Thus, the failure is unpredicted but temporary.

Work which focuses on repairing the node failure by moving the nodes also exists. Algorithms such as [12] and [13] provide an autonomous and distributed ways to recover from node failure. The scheme presented in [12] provides a solution to reinstall connectivity among the isolated patches of WSNs caused due to bulk node failure by populating the least number of nodes possible at the location of failure event while [13] provides a repair mechanism of damaged WSN topologies in the event of multiple node failure by moving one or few nodes towards the failure location to restore the connectivity of the network. However, not all the sensor nodes are equipped with the ability to move. Thus, these solutions are not applicable when the sensor nodes are stationary.

A centralized solution for recovery from node failure is presented in [14] and [15]. A fault management technique is proposed in [14] where a central node analyzes the network status at regular intervals to detect faults if any. Although the approach can be useful in certain applications, due to centralized nature of the mechanism, it consists of high messaging overhead causing low energy efficiency. MANNA-a fault diagnosis using management architecture proposed in [15], is similar to the central fault management scheme.

Work presented in [16], [17] and [18] is also directed towards connectivity restoration. In the algorithm presented by us, however, we concentrate on detecting and repairing the node failure in the system while the network is still a single connected component.

There are mechanisms that detect the node failure but do not work for the recovery. A Cash fault detection scheme in [19] where faulty nodes are detected by gathering neighbor information. It does not perform recovery. A distributed fault detection algorithm for WSNs is proposed in [20]. Faulty nodes are identified based on comparisons between neighboring nodes. A localized fault detection algorithm is evaluated in [21], it is used to identify faulty nodes. The false positive rate is low but not zero.

There has also been work done towards improving the fault tolerance of event detection algorithms. Event region recognition algorithm from [22] tries to prevent node failure from affecting the sensor measurements and presents that the faults in decision making are reduced by 85 – 95% for node failure rates up to 10%. It assumes that the node failure is uncorrelated and takes only random failure into consideration.

2.2On Medial Axis Detection

The medial axis formation starts with the basic step of boundary detection. There are several algorithms implemented to detect boundary nodes in a WSN. The boundary recognition technique [31] uses a threshold on the node degree to separate inner nodes from boundary nodes. Whereas, [32] is a very simple algorithm that uses triangle area to determine whether a node is located on the outskirts or not w.r.t. its neighbors. These algorithms work very effectively for networks with constant density. There are other complex boundary recognition methods such as [33] that achieve to detect inner and outer boundaries in 4 steps. The message complexity is high and the algorithm is based on the assumption that each WSN has one or more inner boundaries. However, this may not be true in many cases. The algorithm works for node degrees as high as 35, 25, and 16. The algorithm also performs for node degrees as low as 6 but the boundary

detection is not as accurate as it is for higher node degrees. Medial axis is a byproduct of this algorithm but the computed axis has a lot of branches with nodes sparsely located on them i.e. it is not definite.

MAP –Medial Axis based naming and routing Protocol - [34]defines medial axis as a set of points which has two or more (instead or one) closest points in set F. where F is the set of all the boundary nodes for which the Medial axis has to be found. The paper also mentions that this type of medial axis is very sensitive to boundary noise and works towards eliminating it by disregarding the unstable medial nodes whose closest boundary nodes are on the same boundary and are within a small distance. However, the scheme only works for networks with at least one inner boundary. For networks with no inner boundary, the second scheme can be applied but the distance can vary with the shape of network. If the network has varying thickness, the 'small distance' can be larger for wider areas and shorter for the sleeker ones in the same network. Thus the protocol works only either for the networks with at least one inner boundary or for the networks with uniform thickness. The lowest average node degree of the communication graph for which the protocol gives significant results is 5.4067. However, the paper does not promise a connected medial axis component.

CASE – Connectivity Based Skeleton Extraction - [35] presents a distributed skeleton extraction algorithm (CASE) to compute a skeleton graph that is robust to boundary noise and is accurate in preserving the original topology of the WSN. This is achieved in steps of 5 i.e. 1) partition the boundary using its corner points, 2) identify skeleton points, 3) generating skeleton arcs by connecting these points, 4) connecting these arcs and 5) refining the coarse skeleton graph. The skeleton nodes are those whose difference of the absolute distances from two nearest boundary segments is less than a threshold. If the group doesn't form a connected component

then we increase the threshold. CASE gives better results than MAP by reducing undesirable nodes. Where the algorithm gives good results, the performance depends on parameters H and threshold. Besides, the maximum hop count between two nodes in the network - a parameter on which further decisions are based - is computed intuitively. The message complexity for computing the skeleton graph increases significantly by boundary segmentation process. The boundary segmentation and coarse skeleton arc refinement process needs an end skeleton node. This does not work for networks with smooth outer boundaries. The algorithm works effectively towards skeleton extraction of networks with average node degree of 20.9. Distance Transformed-Based Skeleton Extraction (DIST) [39] extends the scope of the networks for which medial axis can be extracted. However, the extraction mechanism is very lengthy and computation cost is high.

Work has been done to trace the dynamic changes in the network in [36]. A Dynamic Medial Axis Model for Sensor Networks works towards abstracting geometry and topology of the network while it changes dynamically with the help of a dynamic medial axis. The medial axis is updated with any changes in the network. A mobile sensor is used to survey the network field for changes; the medial axis is modified accordingly. The paper shows efficient results for an 8 connected grid system.

There is a lot of work done that emphasizes on the applications of medial axis of a sensor network. Medial axis can be used to segment the network efficiently to partition at the critical areas that are relatively thinner than the neighborhood regions. Shape Segmentation and Applications in Sensor Networks [37], this paper utilizes Medial axis to segment the complex shaped network into simpler pieces. This is done via boundary detection using [33], then medial axis is formed via selecting the nodes that are equidistant from 2 or more boundary nodes.

Although segmentation is the main purpose of the paper, it relies mainly on proper medial axis formation. The algorithm gives results for average node degree around 7 ~ 8.

Medial axis can also be used to establish an efficient VC scheme in the network [34]. This can further be used to route packets within the network. Efficient data storage is another widely used application of medial nodes where medial nodes are treated as data centers and can be used as backup in case of node failure events [38].

We compare the scope of our algorithm with that of CASE, MALD and DIST using two examples. One is the Face shaped network with smooth inner and outer boundaries and another is an Odd shaped network with only one outer boundary with sharp corners. The comparison is between listed performance parameters of MALD and of CASE. We can also visually gauge the perfection of the medial axis formed for each algorithm.

2.3 Summary

There has been work done on improving the performance of specific network functions in WSNs but the effect of node failure on VCs and thus the WSN as a whole hasn't been studied yet. It is important to study the effect of node failure on VCs of the sensor nodes because when GCs are not available, VCs is the only set of information available for data extraction and processing. This being stated, it implies that for WSNs, where VCs are the foundation of any function that has to be performed on the system, any effect on VCs leads to changes in the performance of all the functions that are dependent on VCs of the sensor nodes.

Medial axis of a sensor network can provide us with a lot of information regarding the network. Research on medial axis extraction has to be extended in such a way that medial axis can prove to be useful to all types of networks. Even though today, we can extract a network

skeleton for networks with high node degrees, a method of medial axis detection is needed that can hold true for widely used networks with grid structure and average node degree around 3 ~ 7.

This thesis addresses the impact of node failure on the VCs of nodes in the WSN. The problem has been stated and a solution has been proposed for the problem. The performance of the proposed algorithm under various scenarios is tested in Chapter 3. Moreover, it proposes a novel method of determining medial axis nodes of a WSN, effective for low node degrees and overcomes conventional problems in medial axis detection. This will be discussed and explained in Chapter 4.

Chapter 3

PERFORMANCE OF VIRTUAL COORDINATE REPAIR POST NODE FAILURE

3.1 Introduction

Wireless Sensor Networks (WSNs) are expected to connect a large number of smart sensor and actuator devices, sense events, exchange data over wireless medium, make collaborative decisions and even interact with the environment. A lot of these network functions rely on Virtual Coordinates (VCs) of the sensor nodes when Geographical Coordinates (GCs) are not available. VCs prove to be very efficient because of their low energy requirement and usage in multiple applications.

However, VCs being based on the hop distances from anchor nodes, network failure events affect the validity of VCs; thus, impacting the performance of the functions performed on the system. A lot of work has been done in the areas such as routing, data storage and security in WSNs but we observe that the performance of such functions over a WSN is affected by node failure. Ref. [6] notes, for example, that Rumor Routing scheme maintains its routing performance for up to 90% for 5% node failures. This means that if about 5% nodes in the network fail, we can expect 90% of the queries that are delivered in absence of faults can still be delivered successfully. However, for node failure percentages over 20%, the performance degrades more severely. The protocols and algorithms may be designed to address the level of fault tolerance required by the sensor networks. Environments with little or no interference can allow the WSNs to have more relaxed protocols. For example, for a sensor network in a house deployed to keep track of humidity and temperature levels, the fault tolerance requirements can be low as the sensor nodes are not easily damaged or interfered by environmental noise.

However, sensor networks deployed in a battlefield for surveillance and detection need to have high fault tolerance levels as the data is critical and sensor nodes can be destroyed by hostile actions [7].

Thus the fault tolerance of the algorithms plays a vital role in the performance evaluation of such techniques. If we could add the repair algorithm to such WSNs it could drastically improve the performance of these techniques and the percentage of node failure that can be sustained by the techniques while maintaining their graceful degradation can be pushed by a large margin. Having fault tolerance increases reliability of the system and the overall life time of the WSN.

3.2 Algorithm

We consider the anchor-based VC system, in which the VCs of a node correspond to the shortest hop distances to a set of anchors. Anchors may be selected randomly, but a proper anchor selection scheme provides better performance with a very low number of anchors [29]. Coordinate system is generated by the set of anchors flooding the network. When node failure affects the shortest path from an anchor to a node, the corresponding virtual coordinate of the node becomes invalid. This may not be a concern to coordinate system such as Topology Coordinates [5] derived from VCs unless the fraction of nodes that failed is high. However, for algorithms that work purely using anchor based VCs [30], invalid VCs can result in dramatic failures. We present a Virtual Coordinate Recovery Algorithm (VCRA), which detects the node failures and responds by recovering the VCs using a distributed adaptive approach. The algorithm adapts naturally to recover VCs irrespective of the number of faults as long as the network remains connected. When VCRA was tested for node failures causing network

partitions, the algorithm successfully detected affected nodes and recovered the VCs w.r.t. the anchors accessible within the partition. VCRA is a distributed algorithm and works locally to correct the VCs of nodes affected due to node failure in the network with minimum exchange of messages.

This algorithm works for virtual coordinate recovery of nodes in a wireless sensor network in the case of single or multiple node failure where the network is still connected as a whole. It is a distributed algorithm and works locally to correct the VCs of nodes affected due to node failure in the network with minimum exchange of messages.

There is a lot of work done in the area of localization schemes which establish various VC systems [23], [24], [25], [26], [27], [28]. A scheme like VCRA can help not only to make these systems robust towards node failures and link failures in the network but also to maintain a graceful degradation of system performance after sudden node failures.

All the notations used in the text and the algorithm are listed in Table (3.1).

Table 3.1: Notations used in text

Notations	Description
N	Total number of nodes
n_i	Node i
M	Total number of anchors
$A_i, i = 1{:}\,M$	Set of all anchors
$h_{n_i n_j}$	Hop distance from n_i to n_j
$V(i) = [h_{n_i A_j}, \dots, h_{n_i A_M}]$	Node $i's$ VC
$r_{n_i A_j}$	Reliability variable of node i w.r.t. jth VC
$R(i) = \left[r_{n_i A_j}, \dots, r_{n_i A_M}\right];\ r_{n_i A_j} \in \{0,1\}$	Set of $n_i's$ reliability variables
$K(i)$	Set of nodes in $n_i's$ 1-hop neighborhood

We also list the definitions for terms used in the text.

- *Virtual coordinates*: the set of VCs $\mathbf{V(i)}$ for **node** $\boldsymbol{i}$
- *Ordinate*: a single VC of **node** $\boldsymbol{i}$
- *Reliable nodes*: A node $\boldsymbol{i}$ is reliable w.r.t. its ordinate If ∃ at least one **node** $\boldsymbol{p} \in \mathbf{K(i)}$s.t. for corresponding ordinate

$$\left\{\boldsymbol{h_{n_pA_j}} = \left[\boldsymbol{h_{n_iA_j}} - \mathbf{1}\right] \boldsymbol{and}\ \left(\boldsymbol{r_{n_pA_j}} = \mathbf{1}\right)\ \boldsymbol{and}\ \left(\boldsymbol{h_{n_iA_j}} \neq \mathbf{0}\right)\right\}; \boldsymbol{j} = \mathbf{1}\boldsymbol{:M};\ \boldsymbol{i} = \mathbf{1}\boldsymbol{:N} \qquad (3.1)$$

  Is satisfied, then $(h_{n_iA_j})$ is considered reliable and thus, $\left(r_{n_iA_j} = 1\right)$.

- *Affected nodes*: Node that cannot satisfy the reliability condition in Eq. (1), after node failure event, are referred as 'affected nodes' or 'unreliable nodes'.
- *Non-affected nodes*: Nodes that can satisfy the reliability condition in Eq. (1), even after node failure event, are referred as 'non-affected nodes'.

3.2.1 Assumptions

We make the following assumptions regarding the network.

1. The sensor nodes are stationary and cannot move.
2. Each node is aware of its neighbors' VCs. This can be easily achieved by each node exchanging the information with its neighbors and this is a part of normal VC based algorithms.
3. Reliability variable for each ordinate is 0, i.e. Unreliable or 1, i.e. Reliable. The default value of *R* is 1.
4. Failure of nodes does not partition the network.

Node failure causes neighbor change for its surrounding nodes. This triggers the VC repair algorithm. Thus, the system starts repairing itself as soon as the failure event occurs. This is irrespective of the location of the failure event in the network.

3.2.2 Pseudocode

Following pseudocode explains the flow of the algorithm in brief:

1. Each node contacts its neighbors
2. Nodes that sense neighbor change initiate reliability check for each of their ordinates
3. All the neighbors of nodes that have sensed neighbor change mandatorily conduct reliability check
4. Neighbors of affected nodes check for reliability
5. Repeat steps 3 and 4 until we find reliable node(s)
6. Reliable nodes inform their neighbors about its reliability.
7. Neighbors who need VC update request for $V(i)$of reliable node
8. When the reliable VCs are received, affected node broadcasts its VCs updates its unreliable ordinates and rechecks its reliability. Overwrite R=1
9. The recovered nodes act as reliable nodes and steps 6,7, and 8 are repeated until there are no unreliable nodes

Input: Neighbors of $n_i;\ n_j \in K(i)$

Output: $R(i), V(i)$

Step 1: Node n_i periodically contacts its neighbors.

Step 2: Node n_i checks the reliability of each of its VCs when number of neighbors decrease i.e. node failure occurs

A VC is reliable if and only if following condition in Eq. (1) is satisfied, then $(h_{n_i A_j})$ is considered reliable and thus, $\left(r_{n_i A_j} = 1\right)$

IF Reliability condition is not met

$$\left(r_{n_i A_j} = 0\right)$$

ELSE IF Reliability condition is met

Inform all the $n_j \in K(i)$that n_i is reliable

END

END

IF number of neighbors change i.e. node failure occurs

Inform all the neighbors to perform reliability check

END

Step 3: Node n_i takes actions as per the received message

IF it is informed to check reliability

Go to step 2

END

IF ∃ at least one $n_p \in K(i)$with reliable required ordinate $\left(r_{n_p A_j} = 1\right)$and $\sum r_{n_i A_j} > 0$

Request for $V(p)$

END

Step 4: When $V(p)$ is received, update $V(i)$ from the reliable node n_p

FOR $j = 1: M$

IF $r_{n_i A_j} = 0$

$$h_{n_i A_j} = h_{n_p A_j} + 1;$$

ELSE IF $(h_{n_p A_j} + 1) < h_{n_i A_j}$

$$h_{n_i A_j} = h_{n_p A_j} + 1;$$

END

END

END of algorithm

Figure 3.1: Algorithm for VC repair scheme

3.2.3 Example

The algorithm has been formally written as shown in Figure (3.1).An example explanation is presented to illustrate our algorithm with the help of a sample network. Figure (3.2.a) and Figure (3.2.b) show two sample wireless sensor network after node failure. The Figures also show the networks with node failure event and thus the nodes invalid with respect to their virtual coordinates due to node failure. Various nodes in the picture are explained as the algorithm progresses.

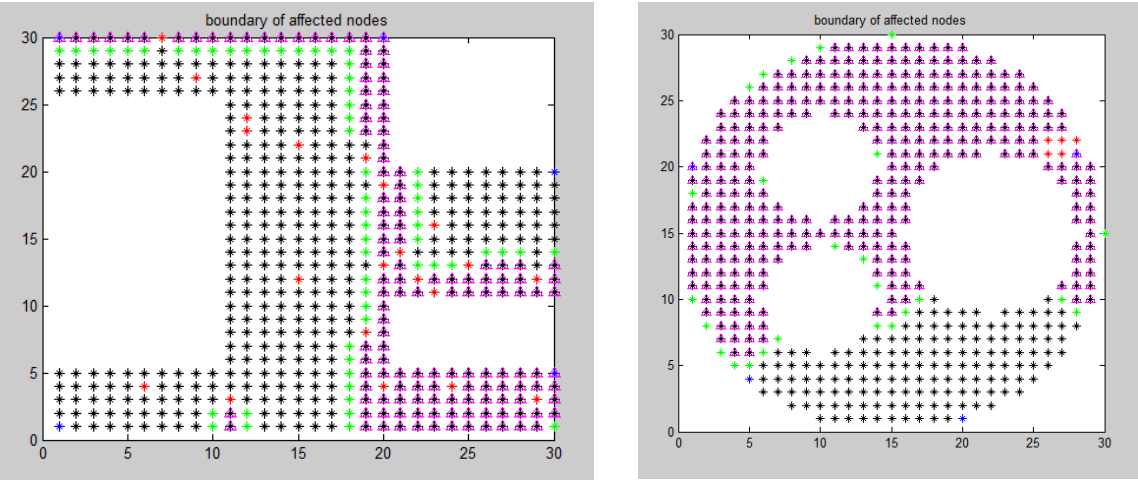


Figure 3.2: Sample network with (black) normal, (red) failed, (pink) affected and (green) reliable nodes to show an example illustration of algorithm flow; (a) Random failure in Odd shaped network (b) Clustered failure in Face shaped network

The black stars shown in the picture are normal nodes whereas the red stars are the failed sensor nodes that do not function anymore. Thus they are not connected to any of its neighbors. All the data related to these nodes are therefore, lost. However, these nodes are deleted after the network and its VCs were established. Thus, failure of these nodes will affect its surrounding nodes w.r.t. it's VCs in some way.

When the nodes beside these deleted nodes sense a neighbor change event, they trigger the reliability check. During the process of reliability check, a node X checks if it has at least one node in its neighborhood for its every VC with VC of one unit lower than that of itself. If the condition of reliability is met, then the node is declared reliable otherwise it is considered unreliable.

Neighbors of all the nodes, who have experienced neighbor change, mandatorily check for their reliability. Unreliable nodes inform their neighbors to have a reliability check and this wave of messages propagates until each message finds a reliable node. Considering, for example, that a network has five anchors, each node will have five VCs. If a node is affected with respect to its third VC, then it searches for a node which is reliable w.r.t. its third VC. If such node is found in the neighborhood, then the search stops there else the present node is declared unreliable and the search is handed over to the neighboring nodes. These nodes in turn search for nodes which are reliable w.r.t. their needs.

In Figure (3.2), the black stars surrounded by pink triangles are the affected nodes and the black stars with green triangles around are the reliable nodes from which the VC repair algorithm extracts the reliable VCs. These reliable nodes broadcast that they are dependable and thus the affected nodes in the neighborhood approach the nearby reliable nodes to derive their updated VCs by adding one unit to the VC of the green nodes. This one unit represents the one hop jump from the reliable node to the affected node. Now after update, the node again checks its reliability. When a node is updated from its nearby reliable node, it possesses all the correct VCs. Thus, this node can be termed as reliable after update. Therefore, this node itself can act as a reliable node for its neighbors. This process continues until any unreliable node in the network needs correction.

3.3 Discussion on node failure and its effect on the WSN system

Let us consider the following notations:

{D}: Set of deleted nodes

{N}: Set of all the nodes in the network whose shortest path to any one of the anchors passes through the deleted node(s)

{A}: Set of all the network anchors

If the deleted node {D} is part of the only shortest path for any other node {N} from at least one anchor from {A} in the network, failure of the node affects all the nodes in set {N}. This implies that neighbors of a node N_i will be affected by its failure if and only if their shortest path to any of the anchor $A_i \in \{A\}$ passes viaN_i.

This being stated, we can say that the nodes on concave turnings in a sensor network are the nodes that can affect its surrounding nodes if deleted, as the concave bending boundary serves as the shortest path to the nodes at the end of the curve. This also holds true for the nodes that are aligned in horizontal or vertical lines with $A_i \in \{A\}$. Whereas, nodes that are well inside the boundary or the nodes that are on convex boundaries do not affect their surrounding nodes as much if deleted.

As we reach outside the shortest paths from anchors to the affected nodes, we reach the reliable nodes. These are the nodes which are used as the reliable source of VCs and thus the affected nodes can derive their new correct VCs with the help of these nodes. After the node failure event, nodes around it initiate the reliability check and reach the reliable nodes if any. These reliable nodes broadcast that they possess the right VCs. Nodes in need request for the VCs and so the reliable nodes transmit their VCs. After repairing the VCs, the corrected nodes

now become new reliable sources for inner affected nodes. This process continues until all the affected nodes are corrected and are made reliable.

Thus, we can estimate that;

$$M_A = 3 * (B + U) \tag{3.1}$$

Where; M_A: Number of messages required to update VCs using this algorithm

U:Number of affected nodes

B: Number of reliable nodes around the boundary of affected nodes

Thus, we can see that the message complexity of the algorithm is only of the order$O(B + U)$i.e. the number of nodes affected and the reliable nodes. This makes the algorithm independent of network size.

3.4 Performance Analysis

We measured the system performance with the help of simulations, implemented in MATLAB 8.1. As we are looking for the algorithm to repair the dead nodes, we simulate the system for a range of node failure percentage. Starting from a low number as 1% we proceed in with multiples of 2 and check the system response for 2%, 4%, 8% and 16% node failure. We stop at 16% as beyond this amount, node failure mainly causes network partition which is not the present scope of this algorithm. System responses through 1% to 16% node failure have been listed and explained.

3.4.1 Testing conditions

As nodes can fail in any pattern, we try to cover these situations under three cases. The system has been simulated for random node failure, clustered node failure and mixed node

failure. Although random and clustered node failure are the patterns for which the WSNs are usually tested for in most of the related work that has been done; in reality nodes might not fail in a particular pattern. Thus, we study the third type i.e. mixed node failure.

The types of node failure patters are as follows:

- Random node failure:

  Nodes are randomly selected for failure by the algorithm. The deleted nodes then have no connectivity with its previous neighbors. After a node is deleted, the next selection is completely independent from the previously selected node.

- Clustered node failure:

  Nodes are deleted in a cluster. Initially, a random node is selected. Neighbors of the deleted node are stored and here after, nodes are selected for failure only from this pool of neighbors. This continues until we reach the desired number of failure percentage.

- Mixed node failure:

  Half of the nodes to be deleted are randomly deleted and half are deleted in cluster.

3.4.2 Performance parameters

The performance of the system has been observed at following instances:

1. Before node failure: (B)
2. After node failure but before VC update: (AB)
3. After node failure and after VC update: (AA)

Hence forth, we will refer to these instances using the initials listed in front of them.

Two shapes of WSNs have been tested for all the conditions listed above. The networks are shown in Figure (3.2). Although the chosen networks have been tested so as to observe the

performance of this algorithm for concave boundaries, dispersed anchor nodes and presence of physical voids in the network, other networks can also be tested. The parameters involved in the performance measurement process are listed in Table (3.2).

Table 3.2: Performance parameters measured for system evaluation

Total nodes in the network before failure	S
Total nodes in the network after failure	N
Total anchors in the network	M
# Nodes deleted	D
# Affected nodes	U
# Reliable nodes found after node failure	B
Messages required to update the VCs using VC repair algorithm	MSG
Messages required for conventional flooding to re-establish the system	C
Message overhead	(MSG/C)*100
Routing from all Non-affected to each Affected node	N-A
Routing from all Affected to each Non-affected node	A-N
Average percentage routability in one entire set (N-A or A-N)	Ravg%
Average hops needed for routing in one entire set (N-A or A-N)	Havg
Average of shortest hop count needed for routing in one entire set (N-A or A-N)	Hs
Normalized hop count	Havg/ Hs
NOTE: the highlighted parameters have been plotted	

Parameter 1: Efficiency (communication cost)

We measure the efficiency of the algorithm by measuring the communication cost of updating VCs. It indicates the ratio of cost of updating through VC repair algorithm to that through system re-establishment.

$$\boldsymbol{Communication\ Cost\ Ratio} = \left(\frac{\boldsymbol{MSG}}{\boldsymbol{C}}\right) * \mathbf{100}$$

(3.2)

Communication cast if defined in Eq. (3.2). Where; M: Number of transmissions occurred during the entire process of VC recovery and C: Total transmissions required to re-establish the system through flooding. For instance, if for a node failure event, the communication cost is 20%, it means that after the node failure event, if conventional re-flooding of the network takes 100 messages to re-establish the system, then the VC repair algorithm requires only 20 messages to repair the system.

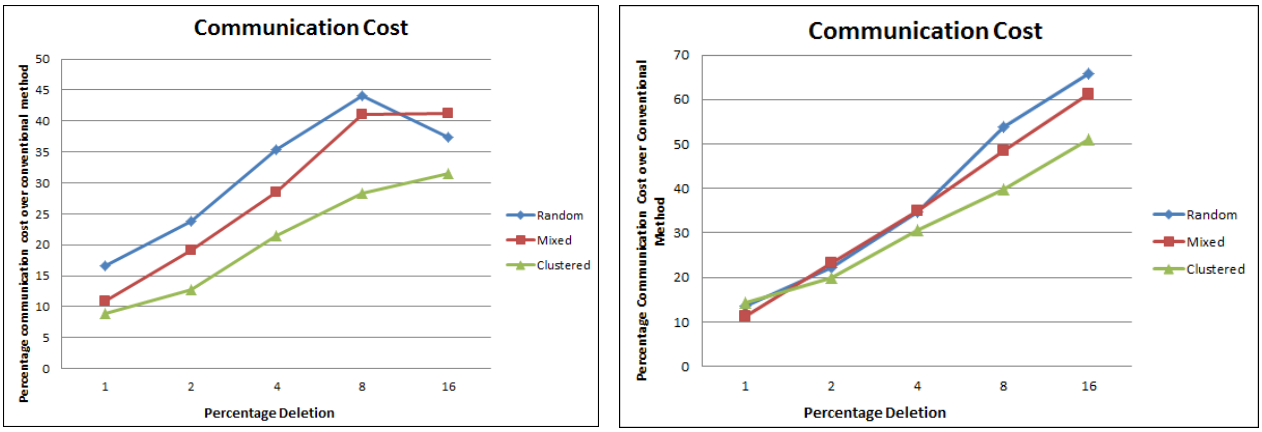


Figure 3.2: Communication cost for system repair measured as a percentage fraction of communication cost for system re-establishment; (a) Results for odd shaped network (b) Results for face shaped network

The chart shown in Figure (3.2.a) gives a comparison of communication cost through our algorithm as compared to the conventional complete network re-establishment cost for repairing the affected VCs of the sensor nodes in the network after node failure. The node failure range varies from 1% to 16% as mentioned earlier and the failure has been carried out in three patterns

i.e. clustered, random and mixed. We compute the percentage communication cost as the average of 100 iterations.

We can observe that the messages required to repair the system is highest in the case of random node failure. This occurs because the affected nodes are scattered. Thus, we have more nodes surrounding affected nodes, which implies more nodes conducting reliability check. Thus, the message complexity for VC repair is highest in case of random node failure events and least in case of clustered node failure event. The message complexity for mixed node failure event falls between these upper and lower limits which is self-explanatory.

We can also see that the maximum communication cost reached for the VC repair even in the clustered failure is only 45% of the number of messages required for conventional flooding to re-establish the system in case of the same node failure event. This shows that repairing the VCs in case of sensor failure is an efficient way of repairing the system as compared to re-establishing it.

Also, we verified the relationship estimated in the algorithm as mentioned in Eq. (3.1), that

$$M_A = 3 * (B + A)$$

Where; M_A: Number of messages required to update VCs using this algorithm

A:Number of affected nodes

B: Number of reliable boundary nodes around the affected nodes

The Figure (3.2.b) illustrates communication costs for Face network under the same circumstances and the results are found to comply with the previous observations.

Parameter 2: Routability

In order to know what effect does repairing the VCs of the nodes has on the overall system performance, we tested the routability of the system in three cases as discussed above i.e. A, AB and AA. H-A, H-AB and H-AA gives the corresponding normalized hop counts for the system which can be defines as stated in Eq. (3.2).

$$\boldsymbol{R}\% = \left(\frac{\boldsymbol{Total\ successfully\ routed\ packets}}{\boldsymbol{Total\ routing\ attempts\ in\ a\ set}}\right) * \mathbf{100} \tag{3.3}$$

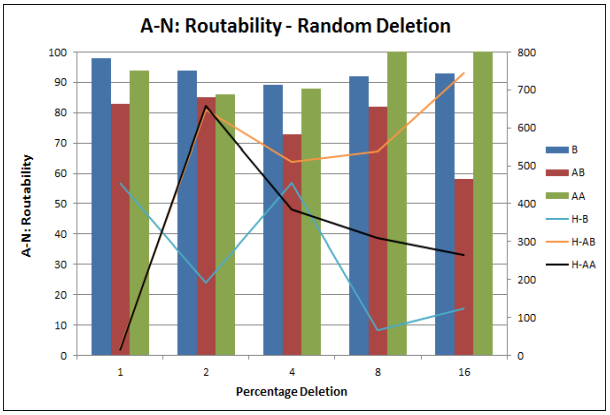
$$Normalized\ hop\ count = \left(\frac{H_{avg}}{H_s}\right) \tag{3.4}$$

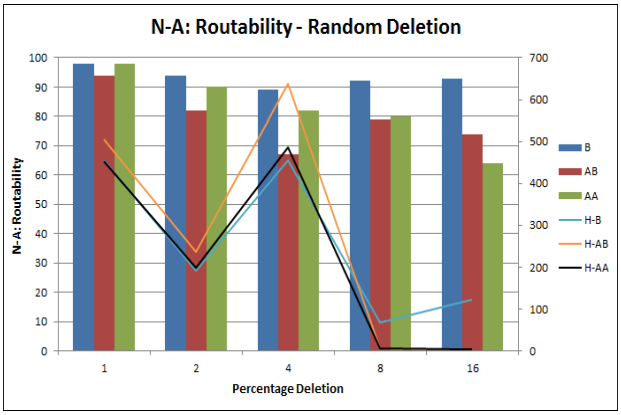
Where; H_{avg}: Average number of hops required to route a packet from non-affected nodes to affected nodes or affected nodes to non-affected nodes

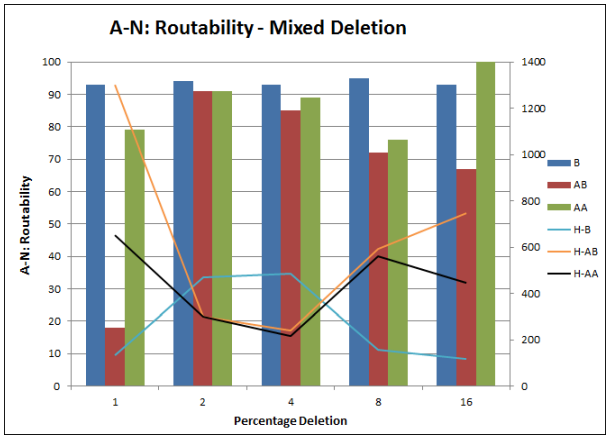
H_s: Average of shortest hop counts needed to route from one node to another

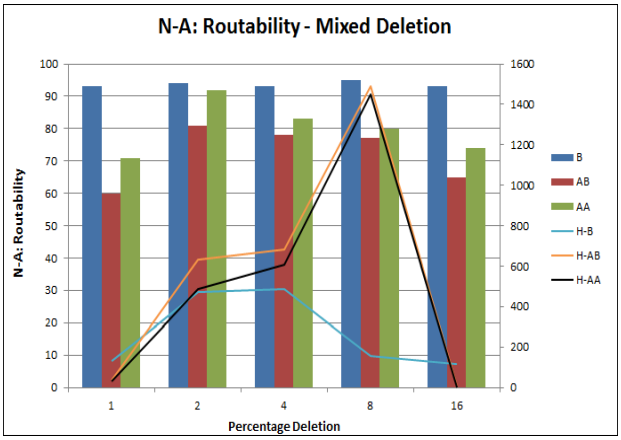
The vertical axis on the left side of the charts refers to the routability of the system whereas the vertical axis on the right side of the chart refers to the normalized hop count for the system.

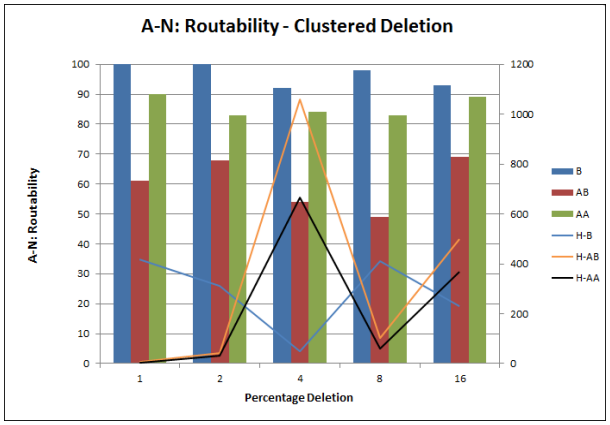
Let's take one chart for detailed explanation. Figure (3.3) gives routing performance of Odd shaped network. Chart in the Figure (3.3.c) gives us the percentage routabilty in case of clustered node failure event. The routing is done from the set of all affected nodes to each node in the set of non-affected nodes. Each bar represents the percentage routability. Blue bar is for system routability before node failure, red bar represents the system routability after node failure but before VC repair and green bar stands for system routability after node failure and after VC repair.


A-N: Routability - Random Deletion
A-N: Routability
Percentage Deletion
B
AB
AA
H-B
H-AB
H-AA


N-A: Routability - Random Deletion
N-A: Routability
Percentage Deletion
B
AB
AA
H-B
H-AB
H-AA


A-N: Routability - Mixed Deletion
A-N: Routability
Percentage Deletion
B
AB
AA
H-B
H-AB
H-AA


N-A: Routability - Mixed Deletion
N-A: Routability
Percentage Deletion
B
AB
AA
H-B
H-AB
H-AA


A-N: Routability - Clustered Deletion
A-N: Routability
Percentage Deletion
B
AB
AA
H-B
H-AB
H-AA

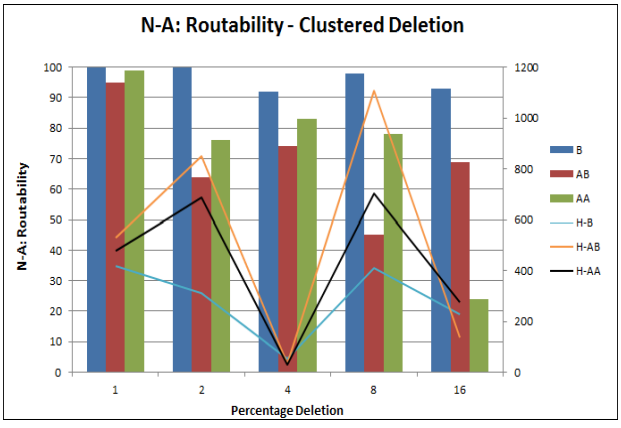

N-A: Routability - Clustered Deletion
N-A: Routability
Percentage Deletion
B
AB
AA
H-B
H-AB
H-AA

Figure 3.3: Percentage routability for odd shaped network, A-N: from a set of affected nodes to a set of non-affected nodes, N-A: from a set of non-affected nodes to a set of affected nodes; (a) A-N: Random failure, (b) A-N: Mixed failure, (c) A-N: Clustered failure, (d) N-A: Random failure, (e) N-A: Mixed failure, (f) N-A: Clustered failure

The line charts show the normalized hop counts for the corresponding situations. Blue line is for A, orange for AB and black for AA. The variation of each individual line is not predictable but we can see that the hop count lowers after we repair the system using VC repair algorithm. Results show that the routability increases after VC update in all the network structures tested. Also, in many cases, it is in fact better than the routability in the original network.

Similarly, other charts can be explained. It illustrates how repairing the VCs of the affected VCs makes a difference in the system performance of the system. We only checked the operation of routing on the system but other functions such as data storage, data collection, topology maps generation, node localization, etc. can also be tested.

We also tested the effects of VC repair on routability operation for the Face network. Figure (3.4) gives results for Face network. The results obtained are along the same nature as that of the results obtained for routability in Odd shaped network. However, one anomaly is observed. In Figure (3.4.f): N-A clustered failure, the normalized hop count is high because the algorithm has to search for reliable nodes across the network by travelling around the physical voids in the network. The lower the node failure, the more the algorithm has to travel in search of reliable nodes. This continues till 8% node failure but as we reach 16% node failure, the void created by deleted nodes actually puts affected nodes and reliable nodes comparatively closer to each other. Thus, reducing the normalized hop count in this case.

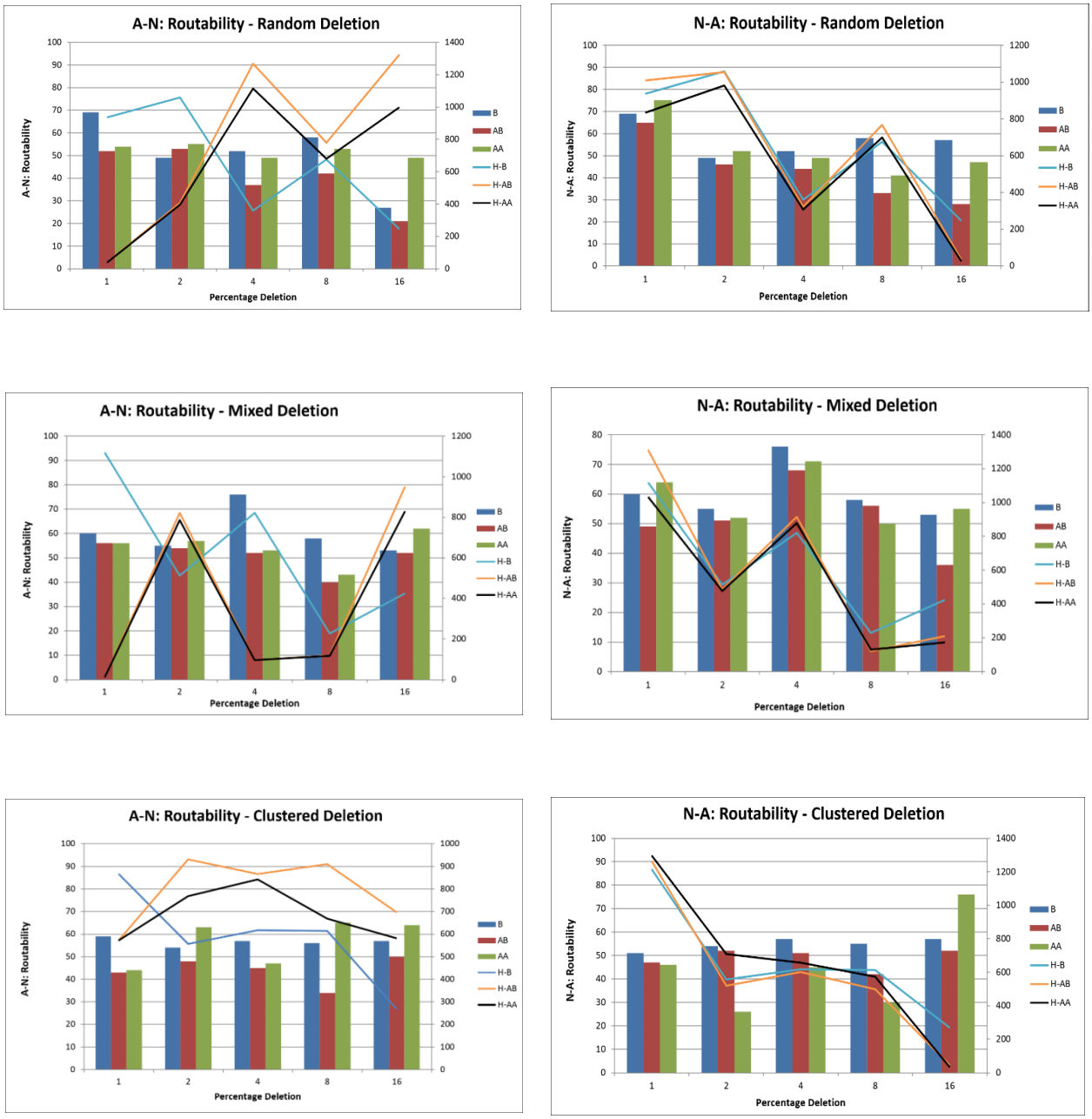


Figure 3.4: Percentage routability for face shaped network, A-N: from a set of affected nodes to a set of non-affected nodes, N-A: from a set of non-affected nodes to a set of affected nodes; (a) A-N: Random failure, (b) A-N: Mixed failure, (c) A-N: Clustered failure, (d) N-A: Random failure, (e) N-A: Mixed failure, (f) N-A: Clustered failure

Parameter 3: Fidelity

We define fidelity as the degree of exactness with which the virtual coordinates are reproduced for the affected nodes. Node failure causes change in number of neighbors for its surrounding nodes. This neighbor change event initiates VC repair algorithm. It starts with reliability check. We define a node to be reliable if it has at least one node in its neighborhood with a VC lower than its own by one unit for each of its VC. This shows that it has a valid path towards the corresponding anchor. After the reliability check, all the nodes that are declared reliable hold at least one node in their neighborhood which will route them to the corresponding anchor. Thus, these nodes are reliable. This implies that, we detect the faulty sensor nodes with 100% correctness. This also implies that the reliable nodes are correctly detected so as to derive the VCs from them. It implies that the algorithm gives a 0 (zero) false positive rate.

Also, as the VCs of the affected nodes are derived from the closest reliable nodes and the receiving node stores the smallest available value of VC received from the reliable neighbor, the new system gives the same results as that of flooding without the deleted nodes in the system. So the VCs after VC repair algorithm are same as it could have been if the system had been established without the dead nodes. Thus the fidelity of the system is maintained high with much lower communication cost.

Parameter 4: Scalability

Scalability of the algorithm is checked by simulating a range of sensor networks with a range of node failure events from 1% to 16% of total nodes failed of different network structures. During simulation, the node failure has been implemented by deleting the randomly selected nodes for failure as per the failure type.

The three types of node failure i.e. random, clustered and mixed; cover almost all the node failure patterns in a sensor network. Figure (3.5) demonstrate a sample node failure example for each type of node failure. Black nodes are normal nodes and red nodes are the failed nodes which are no longer functional.

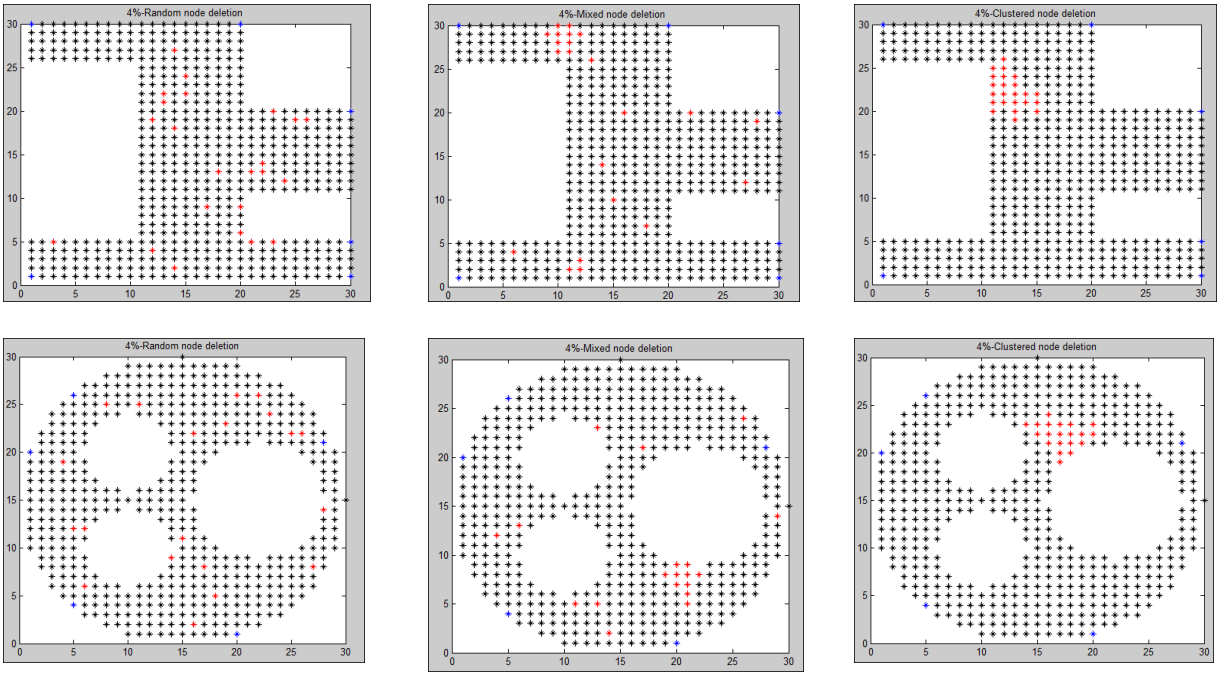


Figure 3.5: Sample node failure patterns for 4% node failure case; (a) Odd shaped network- Random failure, (b) Odd shaped network- Mixed failure, (c) Odd shaped network- Clustered failure, (d) Face shaped network- Random failure, (e) Face shaped network- Mixed failure, (f) Face shaped network- Clustered failure

Figure (3.5) gives an illustration of clustered, random and mixed node failure patterns. Simulations have been carried out on this Odd shaped network which has both, concave and convex network boundaries. The Face shaped network has been tested for observing the worst case results of the algorithm i.e. in the case of having physical voids in the network which makes

the algorithm to travel long paths in order to circumscribe the voids and reach for the reliable nodes.

From the results shown in Figure (3.2), Figure (3.3) and Figure (3.4); we can see that the algorithm performs efficiently throughout the 8% failure. The message overhead then starts being stagnant after 8%. Routability of the system lowers slowly beyond 10% node failure. It mainly affects the N-A routing.This is because we average the routability of one source to all of the nodes in the destination node set. i.e. from one non-affected node to all the affected nodes. Then we take the average of all such combinations. N-A routability is lower as compared to that for A-N because when we update the node VCs in the affected area, we actually create minima. This causes low routability. Thus when affected nodes are in the destination side, the routability lowers.

3.4.3 Strengths & Weaknesses

The algorithm guarantees a correct update of all the affected VCs belonging to the affected nodes as long as the network is connected after node failure. Thus 100% recovery is achieved through the update.

Node failure causes neighbor change for its surrounding nodes. This triggers the VC repair algorithm. Thus, the system starts repairing itself as soon as the node failure event occurs. The number of messages required to establish the VC system through flooding from the anchors has the complexity of$O(N)$. It is a constant for every time the same percentage node failure event takes place in the network. Also, it is a very large number. We can see that the computation cost for re-establishing the system does not depend on the number of nodes that are affected by the node failure event but only on the total number of nodes in the network. Thus, even if a small

number of sensor nodes fail and only few of its surrounding nodes are affected, the system has to invest a huge amount of energy to repair those few sensor nodes. However, the amount of energy consumed to recover the network from node failure through the VC repair algorithm is much lesser than flooding. On the contrary, number of messages required for complete update through the algorithm is independent of size of the network. It only depends on the number of nodes affected by the deleted nodes and the reliable nodes surrounding them. Thus the algorithm serves as a good substitute to the option of re-flooding the system after node failure to maintain the validity of VCs.

We observed that the routability after recovery is better for routing among set of affected nodes to non-affected nodes and vice versa. Thus, the algorithm helps in improving the performance of the system, post node failure event. Even though the improvement in the performance due to VC repair is not significant after a 10% to 15% of node failure, the system does not crash and the degradation of system performance is graceful.

As mentioned earlier, the algorithm is designed to repair the network that is a single connected component; but we also observed its response when subjected to network partition. Two types of partitioning have been studied.

1. *The secondary partition separates only the normal nodes from the rest of the network:*

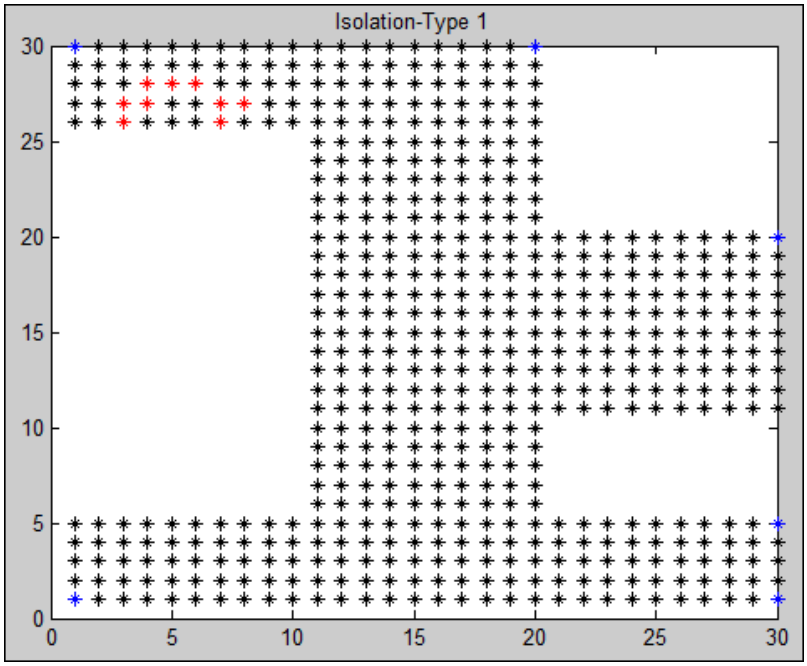


Figure 3.6: Sample network with Type-1 isolation

Figure (3.6) shows an example of secondary partition that isolates only the normal nodes from the rest of the system. I.e. No anchors are removed from the main network.

When the algorithm was subjected to this type of partition, as per the routine, we were able to conduct a reliability check, detect the affected nodes and try to find the reliable nodes. Now, as the primary partition contains all the anchors, affected nodes that fall under this region will be able to find reliable nodes to correct themselves from, but the nodes in the secondary partition cannot find a reliable node because they are not connected to any of the anchors in any way. Thus, only the primary partition of the network is corrected and is repaired for its affected VCs. The secondary partition now has all the affected nodes that cannot be corrected due to lack of reliable node.

If we find a way to make the system work without updating these uncorrected nodes in the secondary partition, e.g. if the nodes with smallest VCs are considered as anchors without changing the value of their VCs, the system will still function well.

2. *The partition isolates few anchors from rest of the anchors:*

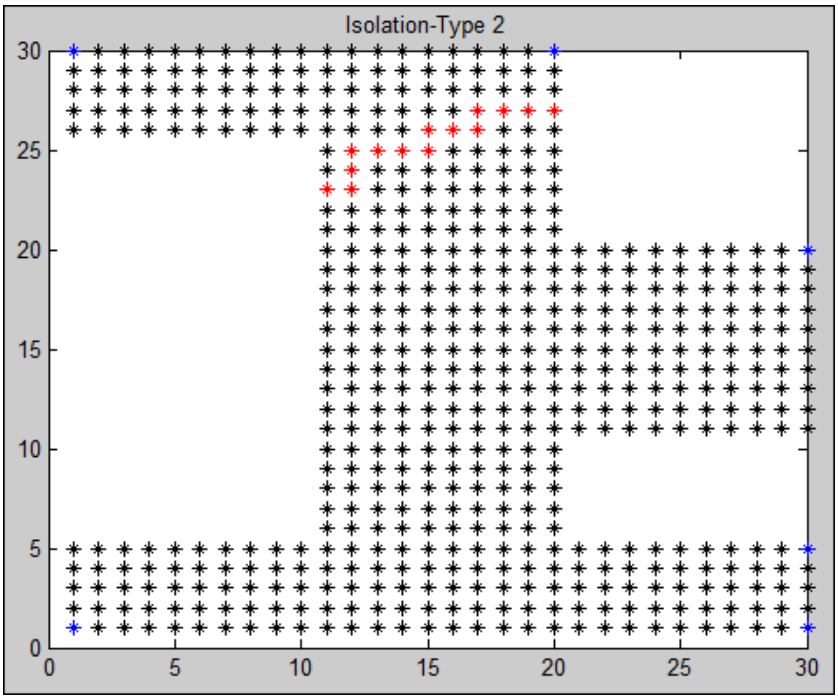


Figure 3.7: Sample network with Type-2 isolation

Figure (3.7) shows an example of a partition that isolates few anchors from the rest of the anchors.

When VC repair algorithm was subjected to this type of partition, as per the routine, we were able to conduct a reliability check, detect the affected nodes and try to find the reliable nodes. Now, as few of the anchors are missing from each partition, all the nodes are detected to as affected nodes and the affected nodes are considered unreliable. Hence, the algorithm fails to

find any reliable nodes when it tries to search. Therefore, the affected nodes cannot be corrected and the system remains unreliable as per our reliability definition.

On the other hand, if we try to make the system work without the VCs that are permanently damaged as the nodes are no longer connected to those anchors, and thus, only use the VCs for which the algorithm can find reliable nodes for and therefore can be corrected; then in that case, the system will work.

We studied how the system responds to the two types of network partitions. The algorithm does not work effectively towards detecting isolated nodes and its recovery. It has to be improvised on its reliability definition as per the situation. The algorithm is not able to detect isolation effectively with low message complexity.

As we saw, if the distances from all the nodes to one are considered, the anchors form local maxima and the VCs that are away from the anchors cause local minima in the network. Since update of VCs always causes increment in the hop count of the affected nodes from anchors, these newly updated nodes cause local minima in the network.

The algorithm has been tested on networks with significant thickness. When we consider networks such as a ring network or a backbone network, where failure of one node affects all the other nodes in the network, VC repair does not provide an efficient way of system repair. In fact, in cases of ring network, repairing the system might not be required at all. Redefining a 'reliable VC' will prove as a sufficient way to keep the system functioning.

3.5 Summary & Conclusions

The evolution of sensor network localization schemes has come a long way from classical geological coordinate system to VCap and now the various forms of virtual coordinate

systems. Non-geographical virtual coordinate systems work for following reasons. They are easy to implement and require much lower cost for implementation. Virtual coordinate systems consume less energy than the geographical coordinate systems and have minimal communication overhead. Despite all these advantages, they can provide much more precision as compared to commercial GPS equipment. VC systems also help to improve performance of network functions such as routing, topology extraction, localization and data extraction. Although the VC systems perform better as compared to the other classical systems we had before, the system performance is hampered with events of node failure. The system degrades with its performance as the sensor nodes fail. Most of these VCS are based on hop distances from anchor nodes and the node failure causes lack of a valid path to few or most of the sensor nodes; depending up on the effect of node failure. This results in lack of communication with some parts of the network and thus the network is paralyzed.

We present a simple yet effective algorithm that works towards detection and recovery of virtual coordinates of the sensor nodes affected by the node failure. We presented the need of such an algorithm with an example of forest fire. Then the algorithm has been explained through a systematic pseudocode, illustration on a sample WSN followed by a discussion on how and why node failure affects the nodes it does. Performance of the system has been meticulously tested on various systems for different node failure percentages to test the relative communication efficiency and percentage routability of the system before node failure, after node failure and after system recovery. The results show that the VC repair scheme works effectively to improve the system performance after node failure. Although it is not always as good as the system before node failure, the algorithm helps the system to degrade gracefully.

As compared to working towards developing an algorithm that tries to improve the performance of a single application over the network; repairing the system so that it will recover the foundation of all the applications helps to improve performance of all the system functions. This is what we had aimed to achieve through VC repair algorithm. It has been tested and proved that VC repair helps improve routability and communication efficiency; similarly the algorithm can be tested for other applications.

Chapter 4

MEDIAL AXIS DETECTION

4.1 Introduction

Wireless sensor networks are being widely used in several areas such as habitat monitoring, battlefield surveillance, etc. In all such cases, to relate the incoming information with the location of the sensor, knowledge of network topology is required. Other operations such as routing, localization, path planning, network segmentation, allocating data storage centers, establishing a virtual coordinate system are also carried out over WSNs. A medial axis holds a lot of information about the sensor network. It can convey the width of the network at any particular cross-section which can be used as basic information for carrying out all the operations mentioned above. Thus, a skeleton – medial axis - of the network plays a vital role as foundation of all such functions.

Medial axis of a network holds a lot of information about the network. Thus, knowing the medial nodes of a network would better equip one for any kind of network repair. However, medial axis detection schemes till date only focus on networks with network node degree as high as 8, 17 and above. Grid based sensor networks, which have average node degree between 3 ~ 7, have several applications such as environmental and habitat monitoring, healthcare monitoring of patients, weather monitoring and forecasting, military and homeland security surveillance, tracking of goods and manufacturing processes, safety monitoring of physical structures and construction sites, smart homes and offices and many other uses. Thus, to be able to fully utilize the information that can be extracted from a medial axis, in all fields of sensor networks, we need a method to detect the medial nodes of WSNs even with low node degrees.

Medial nodes are conventionally defined as follows:

Definition 1: Medial nodes of a network are conventionally defined as the nodes that are equidistant to two or more boundary nodes of the network.

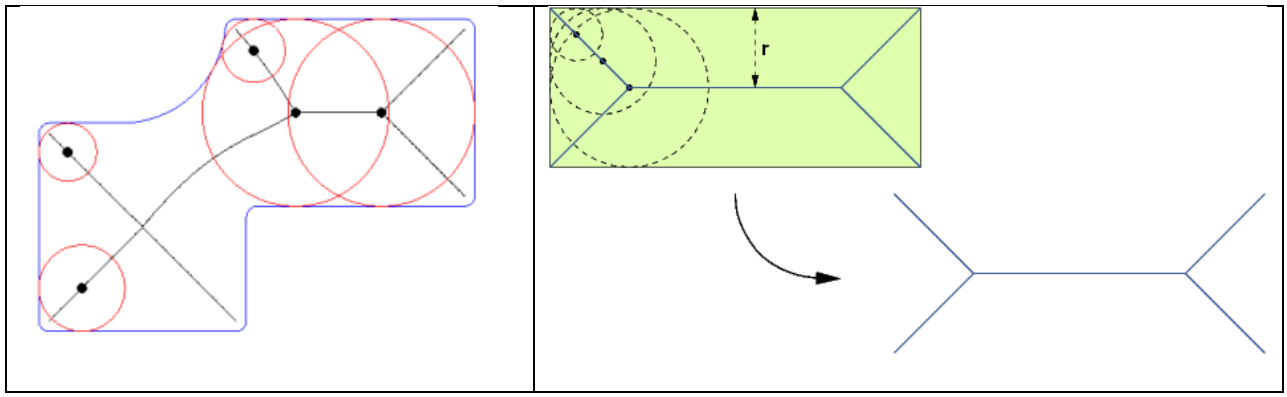


Figure 4.1: Medial Axis definition. (a) a network with its medial points shown equidistant from closest boundary set; (b) (left) Rectangular network with the medial axis extracted

Figure (4.1.a) illustrates this definition further. If we consider any point P on the medial axis, and draw a circle C with P as center; then the circle intersects at least two boundaries of the network in one point. The property of locus of medial points is shown for a polygon with sharp corners in Figure (4.1.b).

However, it is difficult to extract a skeleton – medial axis - for a low node degree network because these networks do not always have the shortest paths to the boundary nodes. The path has to travel through a corner node where it could have taken a diagonal course. Even though this seems to be a small difference, it affects the hop counts of the nodes after boundary flooding to a large extent. This causes several nodes to be equidistant from the boundary even

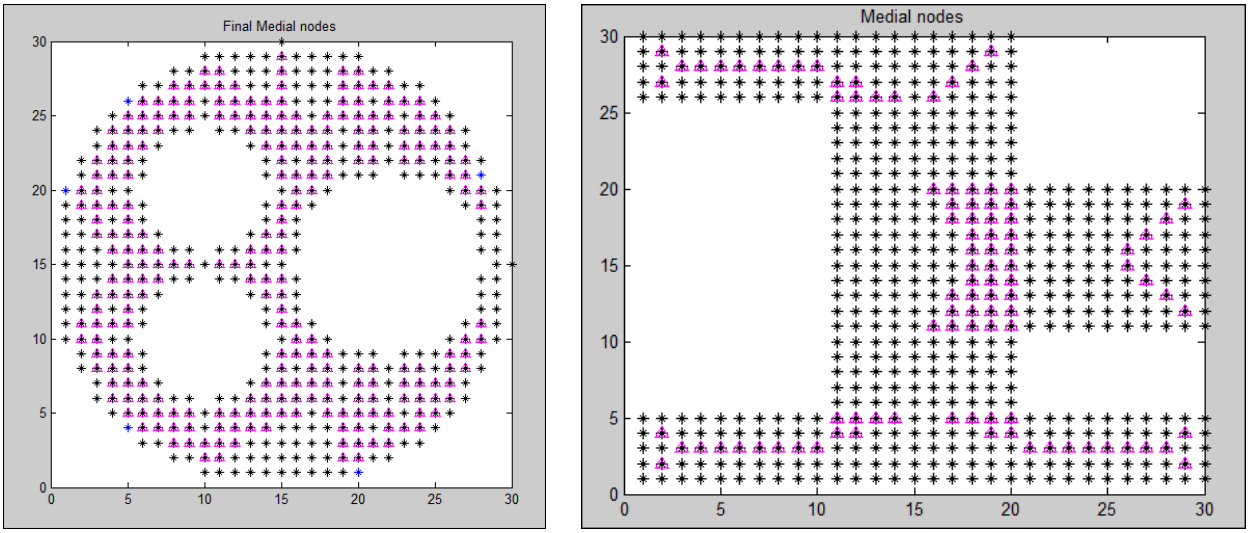


Figure 4.2: Medial Axis of the networks with 4 node connectivity with Definition 1; (a) FACE network with the noisy medial axis; (b) ODD network with boundary noise affected medial axis, even row nodes are not detected as medial nodes

Figure (4.2) illustrates this problem. All the medial axis detection methods are based on this definition. The medial nodes extracted in such a way are noisy. It implies that medial nodes extracted by this definition are sensitive to even a minor bump on the boundaries. This instability is more severe for a discrete network [34]. For example, all nodes that are one hop away from the boundary nodes are medial nodes by Definition 1, which are clearly undesirable. A proposed solution for this is a method called "boundary noise elimination". In particular, the medial nodes whose closest boundary nodes are on the same boundary or within a small distance from each other are disregarded. However, one has to be aware of the width of the smallest cross-section of the network in order to implement this. Also, if the smallest and largest width of the network varies significantly, then this method of boundary noise elimination fails. Furthermore, if the network consists of only one boundary, network such as oval or rectangular networks with no inner boundaries, then there will be no medial nodes detected according to this boundary noise

elimination method. Also, in Figure (4.2.b) we can see that in addition to the boundary noise, when the network has even rows in its branch, the nodes of the network fail to detect themselves as medial nodes. Thus, these issues make it difficult to extract the medial axis of a grid based network.

Sensor nodes are a great option to absorb information about the desired environment. But it is also important to know about the deployed network in terms of its shape, changes happening with time, if any, and the variations in the network itself; in order to maintain and repair the network if required. Knowing a network better makes one better prepared for repairing it in case of events of node failure or link failure. Thus, in order to make the network robust to a maximum possible extent towards such events, it is important to know about the network. Medial axis provides crucial information in several other such applications as network segmentation, routing, data storage, etc.

These functions themselves have several applications of their own. Network segmentation can be used to partition the network at the bottleneck areas so that each partition can be treated as a different network connected at the partition. This prevents the excess load on the boundary nodes while routing and increases the overall life time of the network. Knowing efficient node centers for data storage can improve the data storage for the network which can be used in case of future node failure events for data recovery. WSNs and hence, medial axis of the network have several uses both outdoors and indoors.

We propose a simple algorithm that works towards computation of medial axis of WSNs with any node degree. The algorithm also works for average node degrees as low as 3.5081. Performance of the system is evaluated using message complexity parameter for establishing the medial axis. The algorithm is presented and tested for different network shapes and sizes.

4.2 Algorithm

We consider the boundary detection scheme presented in [32] to detect the boundary nodes in the network. Other boundary detection algorithms can also be used such as detecting the nodes with node degree lower than the average within their h-hop neighborhood; where h is a variable parameter. Boundary detection is the basic step in medial axis extraction in this algorithm, hence, effective boundary detection can lead to better results for medial axis extraction for the network.

We present a distributed algorithm for medial axis detection that also holds true for low average node degrees, unlike work done till date. We detect the medial nodes using a novel definition:

Definition 2: Node that is a local maxima with respect to the hop counts from the boundary nodes; is detected as medial node.

It differs from Definition 1 in the sense that nodes that are locally distant from the boundary nodes as compared to their neighbors are selected as medial nodes. Advantages of this definition serves two intentions. It holds the meaning of medial nodes well as we are ultimately looking for the nodes that are most remote to the boundary nodes of the network. Also, the boundary noise created due to discreteness of the network is eliminated.

Table 4.1: List of notations used in text

Notations	Description
N	Total number of nodes
n_i	Node i
$B_{n_i} = \{0,1\}$	Flag at each node n_i to record if the node is a boundary node. Possible values are '0' i.e. not a boundary node, '1' i.e. boundary node. Default value is '0'.
$B = \{B_{n_1}, \dots\dots, B_{n_N}\}$	Set of all boundary node variables
H_{Bn_i}	Shortest hop distance from closest boundary node to node n_i
$M_{n_i} = \{0,1,2\}$	Flag at each node n_i to record if the node is a medial node. Possible values are '0' i.e. not a medial node, '1' i.e. Medial node and '2' i.e. formed in Round 2 of the algorithm. Default value is '0'.
$M = \{M_{n_1}, \dots\dots, M_{n_N}\}$	Medial flags for all the nodes in the network
$K(i)$	Set of nodes in $n_i's$ 1-hop neighborhood
$Path_{n_i}$	Flag at each node n_i to mark the nodes in the path
$Path_{source}$	Variable at each node to record the original source of the routing packet

Table (4.1) gives a list of notation used in text.

We also list the definitions for terms used in the text.

- *Boundary packets*: Packets sent by boundary nodes to flood the network. These packets contain information regarding the number of hops travelled by the packet. The packet is dropped by nodes with H_{Bn_i} value smaller than the hop count of the packet.
- *Medial search packets:* Packets that contain the ID of original source node and routing variable value is '0'; indicating that search is not finished.

- *Trace-route packets:* It is same as medial search packet except the routing variable value is '1'; indicating that search is finished and the packet is being routed to the original source node.
- *Path node:* Node $n_i$with $Path_{n_i} = 1$

We consider following assumptions while presenting the algorithm:

1) Boundary nodes are detected and they flood with information regarding hops the packet has travelled to reach that node.
2) One hop increments the hop count by one
3) All boundary nodes have $(B_{n_i} = 1)$
4) All the nodes generate a random but unique node Identification Detail (ID) number

Algorithm for Medial axis detection has been described is steps as follows:

1) Each node receives a flooding message from boundary nodes. The variable H_{Bn_i} saves the smallest hop count value from the boundary packets.

2) ROUND 1:

Each node contacts its neighbors to know their H_{Bn_j}; where Node n_iIf H_{Bn_j} for all the n_j is less than or equal to H_{Bn_i}, then n_i declares itself a medial node

3) ROUND 2 (part A):

i. If medial neighbors of n_i are less than 2 then n_i searches for medial nodes.

ii. n_i sends medial search packets with its ID to $n_j$with highest hop count where $n_j \in K(i)$.

iii. When a node receives medial search packet, it marks itself as a path node and stores the original source node ID.

iv. Steps (ii) and (iii) are repeated until medial node is found

v. When a medial node receives a medial search packet, it generates a trace-route message

vi. The trace-route message is transmitted to n_j with lowest hop count where $n_j \in K(i)$ and n_j is a marked path node.

vii. When a path node receives a trace-route message, $Path_{n_i}$ value is made '0' if $Path_{source}$ value matches the one in the trace-route packet. Also, $M_{n_i} = 2$.

viii. Steps (vi) and (vii) are repeated until the packet reaches original source node where it is dropped.

4) ROUND 2 (part B):

i. Each node searches for medial nodes in vicinity.

ii. A medial search packet is sent to neighbors which satisfy the following condition

$\{ (H_{Bn_j} < H_{Bn_i})$ and $(M_{n_j} = 0) \}$ OR $\{ (H_{Bn_j} = H_{Bn_i})$ and $(M_{n_j} = 1) \}$; $n_j \in K(i)$ ()

iii. Step (ii) is repeated until the medial search packet reaches a medial node

iv. When a medial node receives a medial search packet, it generates a trace-route message

v. The trace-route message is transmitted to n_j with lowest hop count where $n_j \in K(i)$ and n_j is a marked path node.

vi. When a path node receives a trace-route message, $Path_{n_i}$ value is made '0' if $Path_{source}$ value matches the one in the trace-route packet. Also, $M_{n_i} = 2$.

vii. Steps (v) and (vi) are repeated until the packet reaches original source node where it is dropped.

Input: Neighbors of n_i; $n_j \in K(i)$

Output: M

Step 1: Node $n_i$receives a flooding message from boundary nodes. The variable H_{Bn_i} saves the smallest hop count value from the boundary packets.

Step 2: ROUND 1:

Node $n_i$contacts its neighbors $n_j, n_j \in K(i)$ to receiveH_{Bn_j}.

IF for all n_j, $H_{Bn_j} \leq H_{Bn_i}$

$$M_{n_i} = 1$$

Informs all n_j

END

ROUND 2 (part A):

Step 1: IF medial neighbors of n_i are less than 2

n_i sends medial search packets with its ID to $n_j$with highestH_{Bn_j}; where $n_j \in K(i)$.

END

Step 2: IF medial search packet is received at n_i

$Path_{n_i}$=1

$Path_{source}$stores the original source node ID

Step 2

END

Step 3: IFn_i with $M_{n_i} > 0$ receives a medial search packet

generate a trace-route message

END

Step 4: Trace-route message is transmitted to $n_j$with lowest H_{Bn_j}; where $n_j \in K(i)$and $Path_{n_j}$=1

Step 5: IFn_i with $Path_{n_i}$=1receives a trace-route message

$Path_{n_i} = 0$ if $Path_{source}$ value matches the one in the trace-route packet

$$M_{n_i} = 2$$

END

Step 6: Repeat steps 4 and 5 until the packet reaches original source node

ROUND 2 (part B):

Step 1: Each node$n_i$searches for medial nodes

Step 2: n_i sends medial search packets with its ID to n_j that satisfy following condition:

$\{ (H_{Bn_j} < H_{Bn_i})$ and $(M_{n_j} = 0) \}$ OR $\{ (H_{Bn_j} = H_{Bn_i})$ and $(M_{n_j} = 1) \}$; $n_j \in K(i)$

Step 3: Step 2 is repeated until the medial search packet reaches a medial node

IF medial search packet is received at n_i

$Path_{n_i}$=1

$Path_{source}$stores the original source node ID

Step 2

END

Step 4: IFn_i with $M_{n_i} > 0$ receives a medial search packet

generate a trace-route message

END

Step 5: Trace-route message is transmitted to $n_j$with lowest H_{Bn_j}; where $n_j \in K(i)$and

$Path_{n_j}$=1

Step 6: IFn_i with $Path_{n_i}$=1receives a trace-route message

$Path_{n_i} = 0$ if $Path_{source}$ value matches the one in the trace-route packet

$$M_{n_i} = 2$$

END

Step 7: Repeat steps 5 and 6 until the packet reaches original source node

END of algorithm

Figure 4.3: Algorithm for Medial axis detection

4.3 Performance evaluation

We measured the system performance with the help of simulations, implemented in MATLAB 8.1. As we are looking for the algorithm to extract the medial axis of the network, we simulate the system for a range of network size and shapes. We selected networks that have no inner boundaries, networks with no sharp corners are also tested and networks that consist of both smooth edges and corners are also tested. We test the algorithm for networks with a node degree range 3 ~ 8. Table (4.2) lists network sizes and average node degrees for the networks shown in Figure (4.4).

Table 4.2: Average node degrees for networks tested

Network / Connectivity	**Face**	**Odd**	**U shaped**	**Window**
4	3.50	3.70	3.84	3.87
8	6.84	7.15	7.53	7.66
No. of nodes	496	550	1000	1728

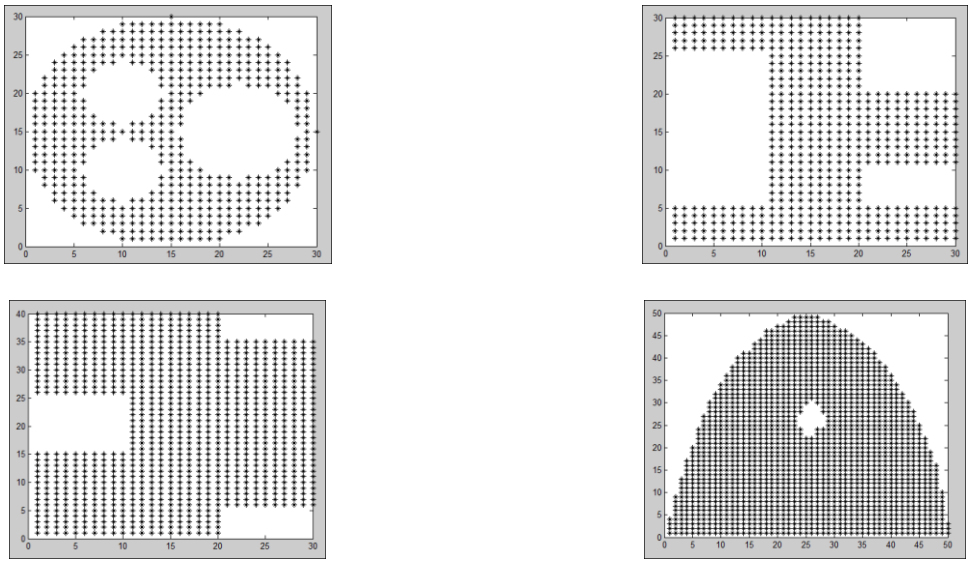


Figure 4.4: Various networks tested under the medial detection algorithm; (a) FACE network; (b) (left) ODD shaped network; (c) (bottom-right) U-shaped network; (d) WINDOW network

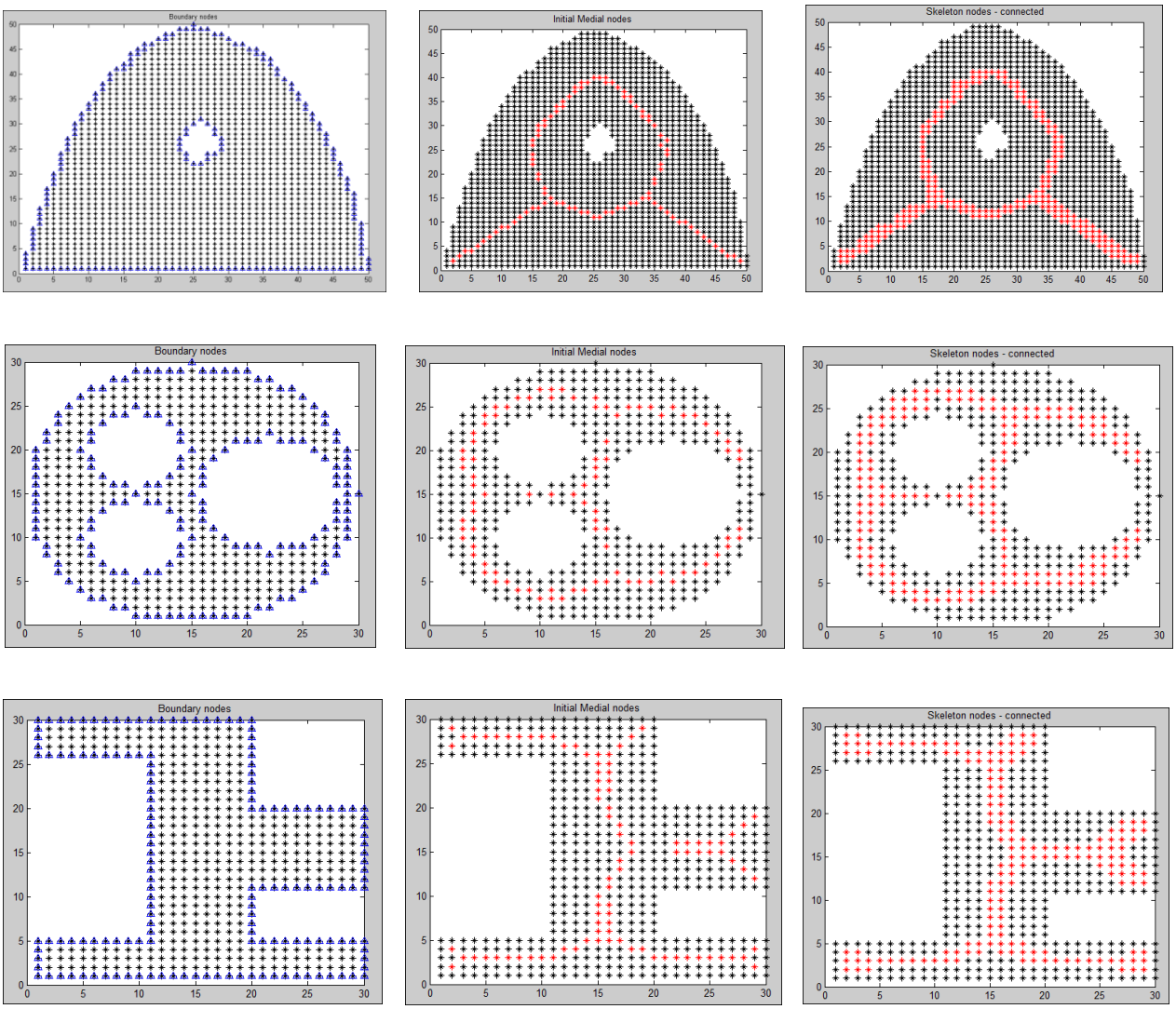


Figure 4.5: Medial Axis of the networks with 4 node connectivity. The left most column shows detected boundary nodes. Middle column shows Round 1 results and rightmost column shows Round 2 results for the algorithm. WINDOW, FACE and ODD shaped networks

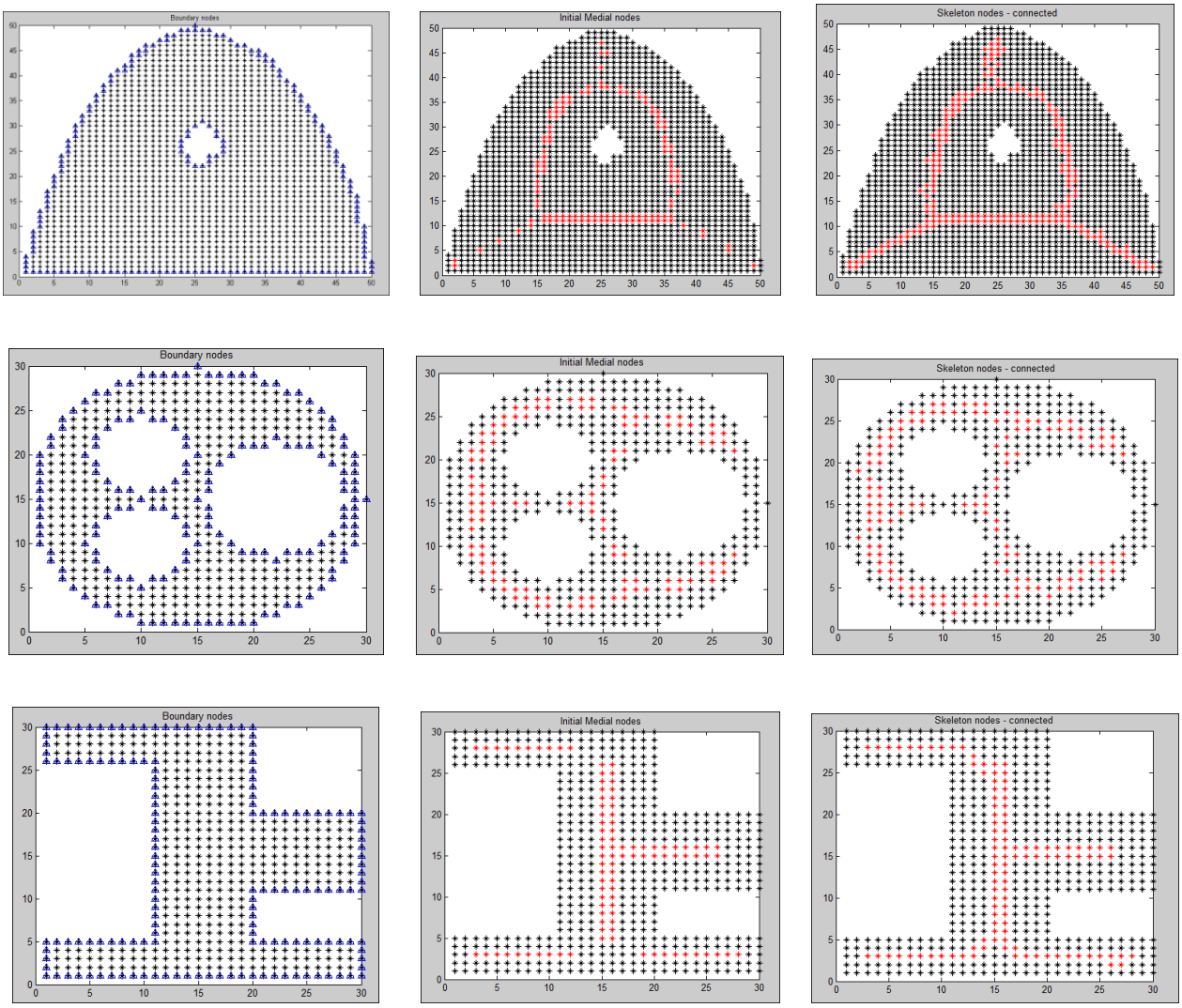


Figure 4.6: Medial Axis of the networks with 8 node connectivity. The left most column shows detected boundary nodes. Middle column shows Round 1 results and rightmost column shows Round 2 results for the algorithm. WINDOW, FACE and ODD shaped networks

The results obtained for medial node detection through the proposed algorithm are shown in Figure (4.5) and (4.6). Till date, medial axis or straight skeleton detection for wireless sensor networks has worked only for higher node degrees. However, the proposed algorithm works for average node degrees starting from as low as 3.5. The algorithm is distributed and works locally.

We can see from Figure (4.5.FACE) and Figure (4.6.FACE) that it is robust to boundary noise. The chart in Figure (4.7) shows that medial axis computation cost ratio for the algorithm i.e.

$$Computation\ Cost\ Ratio = \frac{Round\ 1\ cost + Round\ 2\ cost}{N} \quad (4.1)$$

$$Computation\ Cost\ Ratio = 3 * (N) \quad (4.2)$$

Where; $Round\ 1\ Cost$: Number of messages required to conduct Round 1 of the algorithm

$Round\ 2\ Cost$:Number of messages required to conduct Round 2 of the algorithm

N: Total size of the network

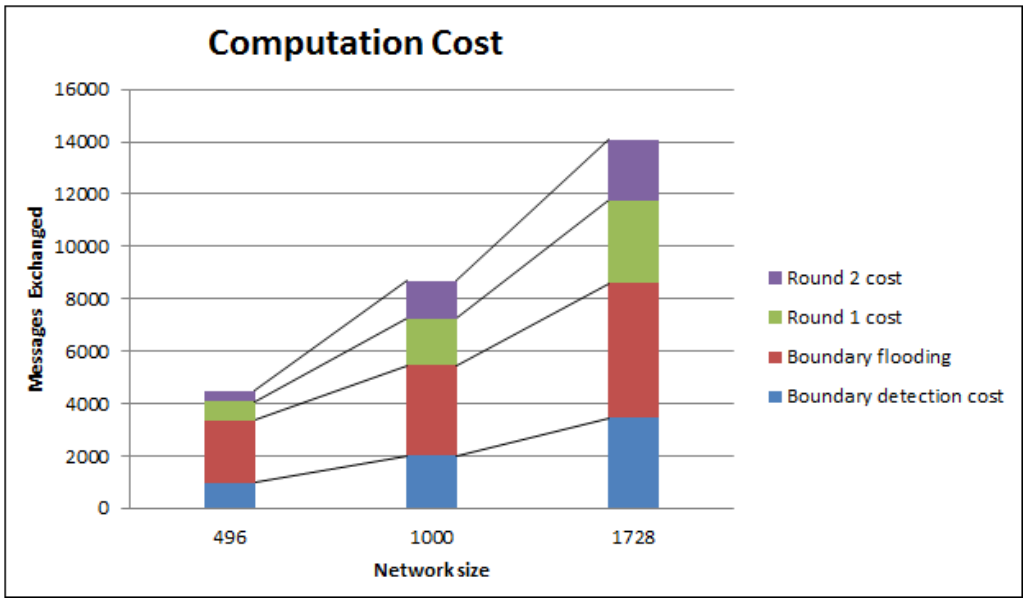


Figure: 4.7: Stepwise Medial Axis Computation Cost of the Algorithm for various network sizes

Figure (4.7) illustrates the step wise split up of computation cost for the medial axis detection algorithm. We can see that computation cost is proportional to the network size. However, the computation cost ratio as defined above in Eq. (4.1) remains constant for all the network shapes and sizes, at 3 on average.

MAP uses Definition 1 for medial axis detection. It further uses boundary noise cancellation technique as mentioned earlier in section 1 of this chapter. But selecting medial nodes that are equidistant to 2 or more boundary nodes introduces a lot of boundary noise even after noise elimination techniques. We studied that to efficiently eliminate the noisy nodes, the width of the network should be known and it should be constant throughout the network. Whereas, Figure (4.5.FACE) and Figure (4.6.FACE) show that proposed Medial axis detection scheme works fine even in the presence of curve boundaries in the network.

Distance Transformed-Based Skeleton Extraction (DIST) works towards extracting the network skeleton with the availability of less than 100% of boundary nodes. It also follows the conventional definition of medial nodes i.e. Definition 1.It claims that the medial axis for a network can be obtained with incomplete boundary nodes. Medial nodes conduct flooding more than three times. DIST also claims to attain better results than MAP. It shows better results than CASE for smooth curves. However, the communication cost is very high. The method is very lengthy and expensive. In fact, the communication cost is highest among MAP, CASE and DIST for given network size. The Computation Cost Ratio is greater than 3. DIST works with very high node degrees. The results framed are with average node degrees 13.59 and 6.1.

CASE presents a modification in the conventional definition of medial nodes. CASE gives better results than MAP by reducing undesirable nodes. Where the algorithm gives good results, the performance depends on parameters H and threshold. Besides, the maximal hop count between two nodes in the network - a parameter on which further decisions are based - is computed intuitively. The message complexity for computing the skeleton graph increases significantly by boundary segmentation process. The boundary segmentation and coarse skeleton arc refinement process needs an end skeleton node. This does not work for networks with smooth

outer boundaries. The algorithm works effectively towards skeleton extraction of networks with average node degree of 20.9. Moreover, CASE cannot work on networks such as the Face network shown in Figure (4.5.a) because it does not have any sharp corners.

The presented algorithm performs better than CASE. MALD gives efficient results with or without sharp edges, with or without inner boundaries and lastly with or without high node degree.

4.4 Applications of Medial axis in a WSN

There is a lot of work done that emphasizes the applications of medial axis of a sensor network. Medial nodes of a network are centrally located at any cross-section of the network. It implies that having detected medial nodes of a network can help us derive information about the network. Being centrally located to the surrounding boundary nodes, medial nodes are aware of their distance from the boundary and the width of the local cross-section of the network. This information can be used for several other applications such as network segmentation; where a network is partitioned at the relatively thin regions which act as bottlenecks while routing. Information extracted from medial axis can be used to segment the network efficiently to partition at the critical areas that are relatively thinner than the neighborhood regions. Work presented in [37] segments the network into pieces by clustering network with medial nodes with similar hop counts from the boundary. The segmentation helps to balance the routing load on the nodes in the network. This reduces the involvement of boundary nodes and includes inner nodes in the network. Medial axis is proposed as a basis for network segmentation in [33] also. Medial axis can also be used to establish an efficient VC scheme in the network [34]. This can further be extended to route packets within the network. Lost packets can be routed along the medial axis to

improve the routability of the network. Efficient data storage and retrieval is another widely used application of medial nodes where medial nodes are treated as data centers and can be used as backups in case of node failure events [37].

Knowing the medial nodes of a network can help in a lot of applications. With the medial axis extraction scheme proposed in this chapter, we can extend this knowledge to several other WSNs and utilize it in such application.

4.5 Summary and Conclusions

We have proposed a distributed medial axis detection algorithm for networks with lower node degrees, using only network connectivity information. The algorithm works for basic boundary detection protocol. It functions in a distributed and localized fashion. Comparison with CASE and MAP techniques shows that MALD gives better results with respect to the scope and performance of the system. We have solved few very crucial problems such as boundary noise in medial axis, disconnected components of medial graph and inability to work at low node degrees. We emphasize that our algorithm is robust, flexible towards the average node degree and the shape or size of the network.

Chapter 5

SUMMARY, CONCLUSIONS AND FUTURE WORK

5.1 Summary and Conclusions

Virtual Coordinates (VCs) are a very attractive and efficient way of dealing with wireless sensor nodes because of its simplicity, low computation cost and no additional hardware requirement as GPS. However, VCs are vulnerable to network events of node failure and faulty links. In Chapter 3, we studied the effects of such node failure events on VCs and thus the network as a whole. We also present a simple, distributed and locally operated network algorithm that works towards repairing the affected VCs because of node failure in the network. Chapter 3 presents the performance of the various systems under different network conditions as a response to the algorithm after node failure. The system response to node failure and VC repair algorithm has been discussed. We also have a look at the algorithm from its strengths and weaknesses point of view. The algorithm works for networks of all sizes and node degrees.

This thesis also presents a novel definition and thus a novel way to detect medial axis nodes in a WSN. Using only the connectivity information of the network, we are able to extract a connected medial axis for the network in not more than two rounds of algorithm. We propose a distributed algorithm that works locally with very low memory requirements and low computation cost. Medial Axis Detection algorithm for Low Degree networks (MALD) works for low degrees as well as higher degrees. The computation cost lowers as the node degree increases; because shorter paths are available. MALD works for average node degrees as low as 3 ~ 8. It performs better than other medial axis detection algorithms proposed till now such as MAP, CASE and DIST in aspects such as communication cost, scope of the network shape and

algorithm complexity. Being able to detect medial nodes of a low degree network makes it possible to utilize these nodes in several applications such as data storage, routing, topology generation and network segmentation; which have further applications of them.

The focus of this thesis is on node failure recovery of the VCs for better system performance. The algorithm provides significant improvement over the routability of the network after node failure. It also helps to maintain graceful degradation of the network post node failure. The algorithm has also been proved to be computationally efficient as compared to the conventional network amendment method. Properties of VC repair algorithm such as low communication cost, energy efficiency and low memory requirements contribute to increased lifetime of the WSN whilst providing a foundation for better system performance.

5.2 Future work

The VC repair algorithm works efficiently as demonstrated by the simulation results for single or multiple node failure. It also helps to recover the lost connectivity for internal network isolation where none of the anchor nodes are in the isolated network region.

We studied the algorithm performance mainly on 2D grid networks. All the simulations were subjected to network with rectangular connectivity and 1-hop coverage. We are interested in testing the algorithm performance in case of increased hop coverage while maintaining optimum energy consumption. With the enhanced reach of each sensor node, we predict this will lead to efficient VC repair, increased network lifetime and lower probability of network partition due to node failure.

We are also interested in observing the effects of changed network connectivity over the system performance. Although rectangular connectivity provides fairly good energy

consumption and reliability; hexagonal, triangular and strip based connectivity are also deployed in practical WSNs. In general, when sensor nodes are randomly deployed, they are n-connected. We can extend the work of this algorithm to a generalized version in case of n-node connectivity.

We intend to take this work further in order to be able to recover the WSN when the network is partitioned due to node failure. Few meaningful amendments in the way the algorithm detects reliable nodes will help to determine the repairable VCs and unrepairable VCs. Knowing the network better will help in concluding whether the network is partitioned; with lesser number of communications overhead.

The greater picture will be to modify the VC repair algorithm in order to make the system self-learning and autonomous in nature so as to be able to adjust with dynamic network topology.

We would like to experiment with different boundary detection techniques along with the proposed Medial Axis detection algorithm for low degree networks. Medial axis has several applications. We would like to work towards extending the algorithm for different applications and evaluate its efficiency towards applications such as data storage, routing and topology extraction. We also look forward to extend the algorithm to 3D networks, where the complexity of boundary detection increases further.

REFERENCES

1. C. Long, C. Wu, Y. Zhang, H. Wu, M. Li and C. Maple, "A survey of localization in wireless sensor network," International Journal of Distributed Sensor Networks 2012.
2. Q. Han, L. Paradis, "A survey of fault management in wireless sensor networks," Journal of network and systems management 15.2: 171-190, 2007.
3. S. Funke, "Topological hole detection in wireless sensor networks and its applications," Proc. of Joint Workshop on Foundations of Mobile Computing, ACM, 2005.
4. D. C. Dhanapala and A. P. Jayasumana, "Topology preserving maps from virtual coordinates for wireless sensor networks, "IEEE 35th Conf. on Local Computer Networks, 2010.
5. D. C. Dhanapala and A. P. Jayasumana, "Topology preserving maps - extracting layout maps of wireless sensor networks from virtual coordinates," IEEE/ACM Transactions on Networking, Vol. 22, No. 3, 2014.
6. D. Estrin, D. Braginsky,"Rumor routing algorthim for sensor networks, " Proc. of the 1st ACM International Workshop on Wireless Sensor Networks and Applications, 2002.
7. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci, "Wireless sensor networks: a survey," Computer networks 38.4: 393-422, 2002.
8. H. Liu, A. Nayak, and I. Stojmenović, "Fault-tolerant algorithms/protocols in wireless sensor networks." Guide to Wireless Sensor Networks. Springer London, 261-291, 2009.
9. D. S. Park, "Fault tolerance and energy consumption scheme of a wireless sensor network," International Journal of Distributed Sensor Networks, 2013.
10. P. Maestrini and S. Chessa, "Fault recovery mechanism in single-hop sensor networks," Computer communications 28.17: 1877-1886, 2005.

11. I. Koren and M. Gregoire, "An adaptive algorithm for fault tolerant re-routing in wireless sensor networks,"5th Annual IEEE International Conference onPervasive Computing and Communications Workshops, 2007.
12. M. Younis and S. Lee, "Recovery from multiple simultaneous failures in wireless sensor networks using minimum steiner tree," Journal of Parallel and Distributed Computing 70.5: 525-536, 2010.
13. Y. K. Joshi and M. Younis, "Autonomous recovery from multi-node failure in Wireless Sensor Network," Global Communications Conference (GLOBECOM), IEEE 2012.
14. W. L. Lee, A. Datta, and R. Cardell-Oliver, "Winms: Wireless sensor network-management system, an adaptive policy-based management for wireless sensor networks," 2006.
15. L. B. Ruiz, I. G. Siqueira, L. B. eOliveira, H. C. Wong, A. A.Loureiro, et al., "Fault management in event-driven wireless sensor networks," Proc. 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems, 2004.
16. J. Du, L. Xie, X. Sun, and R. Zheng, "Application-oriented fault detection and recovery algorithm for wireless sensor and actor networks,"International Journal of Distributed Sensor Networks, 2012.
17. G. Venkataraman, S. Emmanuel and S. Thambipillai, "Energy-efficient cluster-based scheme for failure management in sensor networks," IET communications 2.4: 528-537, 2008.
18. A. Akbari, A. Dana, A. Khademzadeh and N. Beikmahdavi, "Fault detection and recovery in wireless sensor network using clustering," International Journal of Wireless & Mobile Networks (IJWMN)3.1: 130-138, 2011.
19. P. Santi and S. Chessa, "Crash faults identification in wireless sensor networks," Computer Communications 25.14: 1273-1282, 2002.
20. Y.-H. Choi and M.-H Lee, "Fault detection of wireless sensor networks," Computer Communications 31.14: 3469-3475, 2008.
21. S. Kher, A. Somani and J. Chen, "Distributed fault detection of wireless sensor networks," Proc. Workshop on Dependability Issues in Wireless Ad Hoc Networks and Sensor Networks, ACM, 2006.

22. S. Iyengar and K. Bhaskar, "Distributed bayesian algorithms for fault-tolerant event region detection in wireless sensor networks," IEEE Transactions on Computers 53.3: 241-250, 2004.

23. A. Caruso, S. Chessa and S. De, "GPS free coordinate assignment and routing in wireless sensor networks," INFOCOM, 24th Annual Joint Conf. of the IEEE Computer and Communications Societies, 2005.

24. A. Gautam and A.. Gautam, "Accurate localization technique using virtual coordinate system in wireless sensor networks," International Journal of Recent Trends in Engineering, Vol. 2, No. 5, Nov. 2009.

25. T. Grubman, Y. A. Sekercioglu, and N. Moore, "Virtual localization for robust geographic routing in wireless sensor networks."

26. L. Ma, J. He, D. Ma, "A localization algorithm based on virtual beacon nodes for wireless sensor network," Proc. of International Conference on Information Acquisition, 2007.

27. A. Karima, B. Mohammed and B. Azeddine, "New virtual coordinate system for improved routing efficiency in sensor network," International Journal of Computer Science Issues (IJCSI); Vol. 9 Issue 3, May 2012.

28. D. C. Dhanapala and A. P. Jayasumana, "Directional virtual coordinate systems for wireless sensor networks," Proc. of IEEE International Conference on Communications (ICC), June 2011.

29. D. C. Dhanapala and A. P. Jayasumana, "Anchor selection and topology preserving maps in wsns – a directional virtual coordinate based approach," Proc.36th IEEE Conference on Local Computer Networks, Oct. 2011.

30. Q. Cao and T. Abdelzaher, "Scalable logical coordinates framework for routing in wireless sensor networks," ACM Transactions on Sensor Networks,Vol. 2,pp. 557-593, Nov 2006.

31. A. Kroller, D. Pfisterer, S. Fischer and S. P. Fekete, "Neighborhood-based topology recognition in sensor networks," Algorithmic Aspects of Wireless Sensor Networks. Springer Berlin Heidelberg, 123-136, 2004.

32. D. C. Dhanapala, A. P. Jayasumana, and S. Mehta. "On boundary detection of 2-d and 3-d wireless sensor networks," Global Telecommunications Conference (GLOBECOM), IEEE, 2011.
33. Y. Wang, J. Gao and J. S. B. Mitchell, "Boundary recognition in sensor networks by topological methods," Proc. 12th Annual International Conference on Mobile Computing and Networking, ACM, 2006.
34. J. Bruck, J. Gao and A. Jiang, "MAP: Medial axis based geometric routing in sensor networks," Wireless Networks 13.6: 835-853, 2007.
35. H. Jiang, W. Liu, D. Wang, C. Tian and X. Bai, "CASE: Connectivity-based skeleton extraction in wireless sensor networks," INFOCOM, IEEE, 2009.
36. L. Lin and H. Lee, "A dynamic medial axis model for sensor networks," Proc. 13th IEEE International Conference onEmbedded and Real-Time Computing Systems and Applications, 2007.
37. X. Zhu, R. Sarkar and J. Gao, "Shape segmentation and applications in sensor networks," INFOCOM, 26th IEEE International Conference on Computer Communications, 2007.
38. S. Xia, N. Ding, M. Jin and H. Wu, "Medial axis construction and applications in 3D wireless sensor networks," INFOCOM, 2013.
39. W. M. Pang and K. S. Choi, "Distance transform-based skeleton extraction and its applications in sensor networks," 2013.
40. W. Liu, H. Jiang, X. Bai, G. Tan and C. Wang , "Skeleton extraction from incomplete boundaries in sensor networks based on distance transform," Proc. 32ndIEEE International Conf. on Distributed Computing Systems (ICDCS), 2012.

APPENDIX A

SIMULATION OF VIRTUAL COORDINATE REPAIR ALGORITHM

MATLAB code for simulating Virtual Coordinate Repair Algorithm (VCRA) is given below. The code has been presented in parts where Section A.1 gives the code for network establishment. Section A.2 gives code for inserting node failures as per desired patterns. Section A.3 gives code for reliability check, VC update and performance evaluation. Section A.4 gives code for routing function used to perform routing in the network at different intervals.

A.1 Network establishment

```
%INPUT: XY coordinates of sensor nodes in the network
%OUTPUT: Adjacency matrix for the network, Virtual Coordinates for the network for ENS
anchor nodes

%Initialization of parameters being monitored/ measured----------
MSG = 0;                %total msgs exchanged
VCaffected = 0;         %total VCs affected due to accidental death of nodes
VCcorrected = 0;        %total VC that this algo was able to correct from affected VCs
NODEcorrected = 0;   %total nodes that this algo was able to correct from affected nodes
Affected = [];
%deadNodes - total nodes died suddenly
%no_nodes - size of the network
%no_anchors - determined by ENS algo.
%GnodeXY - X and Y coordinates of the sensor nodes
%GnodePos - network node indices

% Prepare network----------------------------------------------
global GnodeXY;
GnodeXY = [];
global GnodePos;
GnodePos = [];
A = zeros;
GnodePos = zeros;
GnodeXY = input('enter coordinates :');
[r,c]=size(GnodeXY);
for i=1:r
     A(GnodeXY(i,1),GnodeXY(i,2))=1;
```

```
end
[r,c] = size(A);
n=1;
i=1;
j=1;
for i=1:r
  for j=1:c
     if A(i,j)==1
        GnodePos(i,j)=n;
        n=n+1;
     end
  end
end
no_nodes = n-1;


% Connectivity information is stored in adjacency matrix
%adjacency matrix for 4 connected neighbors:---------------------------------
global adj;
row=r;
column=c;
[r,c] = size(GnodePos);                    %# Get the matrix size
diagVec1 = repmat([ones(c-1,1); 0],r,1);  %# Make the first diagonal vector
                                           %#   (for horizontal connections)
diagVec1 = diagVec1(1:end-1);              %# Remove the last value
diagVec2 = ones(c*(r-1),1);                %# Make the second diagonal vector
                                           %#   (for vertical connections)
adj = diag(diagVec1,1)+...                 %# Add the diagonals to a zero matrix
     diag(diagVec2,c);
adj = adj+adj.';
%this is basic adj..now substitute 0 for removed nodes.check by [n,m]
i=1;
for n = 1:row
   for m = 1:column
         if GnodePos(n,m) == 0;
            adj(i,:)=[];
            adj(:,i)=[];
            i = i-1;
         end
            i = i+1;
   end
end
Oadj = adj;
```

```
% Getting VCs:-----------------------------------------
% Dijkstra's algorithm is used to calculate the VCs by computing the shortest hop distances
between an anchor and a node
% algorithm works on adjacency matrix for connectivity
Anchors = input('Source node array:');
S = sparse(adj);
for i = 1:numel(Anchors)
   VCmatrix(i,:)=dijkstra_sp(S,Anchors(i));
end
```

A.2 Inserting desired node failure pattern

```
%INPUT: Network information, node failure pattern, node failure percentage
%OUTPUT: Network information for network with node failure

%Insert node failures:-----------------------------------------
OGnodePos = GnodePos;
deadNodes=0;
mynumber = input('Enter type of failure\n 1: Randomly distributed\n 2: Mixed\n 3: Clustered:');
adj_temp = adj;
[r,c]=size(adj_temp);
add = (1:r);
matrix = cat(1,add,adj_temp);
add = cat(2,0,add);
matrix = cat(2,add',matrix);
adj_temp1 = matrix;
adj_temp2 = matrix;

switch mynumber

   case 1  %1:Randomly distributed
       percentage = input('Enter percentage to be deleted: ');
      zz=fix(no_nodes*percentage/100);
      flaw=[];
      while numel(flaw)<=zz
        s = randsample(no_nodes,1);
        % updating adj matrix for deleted node
        sss = find(adj_temp1(1,:)==s);
        adj_temp1(sss,:)=[];
        adj_temp1(:,sss)=[];

        if (~any(Anchors==s)) && (checkconnected(adj_temp1)) %check the connectivity here.
          disp('connected');
          flaw = cat(2,flaw,s);
```

```
        flaw=unique(flaw);
        adj_temp2 = adj_temp1;
      else
        disp('Disconnected');
        adj_temp1 = adj_temp2;
      end
    end

case 2  %2:half distributed..half clustered
    percentage = input('Enter percentage to be deleted: ');
    %sparse failure
    zz=fix(no_nodes*percentage/200);
    flaw=[];
    while numel(flaw)<=zz
      s = randsample(no_nodes,1);
      % updating adj matrix for deleted node
      sss = find(adj_temp1(1,:)==s);
      adj_temp1(sss,:)=[];
      adj_temp1(:,sss)=[];
      if (~any(Anchors==s)) && (checkconnected(adj_temp1)) %check the connectivity here.
        %Connected
        flaw = cat(2,flaw,s);
        flaw=unique(flaw);
        adj_temp2 = adj_temp1;
      else
        %Disconnected
        adj_temp1 = adj_temp2;
      end
    end
    %cluster failure
    zz=fix(no_nodes*percentage/100);
    pool=[];
    s = randsample(no_nodes,1);
    if any(Anchors==s)
    else
       flaw=cat(2,flaw,s);
    end
    nebrs=[];
    nebrs=find(adj(s,:)); %storing neighbors of nodes being deleted
    pool=cat(2,pool,nebrs);
    while numel(flaw)<zz
      s = randsample(pool,1);
      % updating adj matrix for deleted node
      sss = find(adj_temp1(1,:)==s);
      adj_temp1(sss,:)=[];
      adj_temp1(:,sss)=[];
```

```
      % conditions to actually delete the selected node from the n/w
      %delete if = it is NOT an anchor + the n/w stays connected after
      %its failure
      if (~any(Anchors==s)) && (checkconnected(adj_temp1))%check the connectivity here.
        %Connected
        flaw = cat(2,flaw,s);
        flaw=unique(flaw);
        nebrs=[];
        Rnebrs = [];
        sss = find(adj_temp2(1,:)==s);
        nebrs=find(adj_temp2(sss,:)==1);
        [r,c]=size(nebrs);
        if numel(nebrs)>=1
          for i=1:c
             pin = nebrs(i);
             Rnebrs=cat(2,Rnebrs,adj_temp2(1,pin));
          end
        end
        pool=cat(2,pool,Rnebrs);
        pool=unique(pool);
        adj_temp2 = adj_temp1;
      else
        %Disconnected
        adj_temp1 = adj_temp2;
      end %iF conditional statement ends here
    end   %while loop ends here

case 3  %3:Clustered
    percentage = input('Enter percentage to be deleted: ');
    zz=fix(no_nodes*percentage/100);
    flaw=[];
    pool=[];
    s = randsample(no_nodes,1);
    if any(Anchors==s)
    else
       flaw=cat(2,flaw,s);
    end
    nebrs=[];
    nebrs=find(adj(s,:)); %storing nebrs of nodes being deleted
    pool=cat(2,pool,nebrs);
    [~,ID]=find(pool==0);
    pool(ID)=[];
  if size(adj_temp1)>0
    while numel(flaw)<=zz
      s = randsample(pool,1);
      % updating adj matrix for deleted node
```

```
        sss = find(adj_temp1(1,:)==s);
        adj_temp1(sss,:)=[];
        adj_temp1(:,sss)=[];
        % conditions to actually delete the selected node from the n/w
        %delete if = it is NOT an anchor + the n/w stays connected after
        %its failure
        adio=adj_temp1;
         adio(:,1)=[];
         adio(1,:)=[];
        if (~any(Anchors==s)) && (checkconnected(adj_temp1))%check the connectivity here.
          %Connected
          flaw = cat(2,flaw,s);
          flaw=unique(flaw);
          nebrs=[];
          Rnebrs = [];
          sss = find(adj_temp2(1,:)==s);
          nebrs=find(adj_temp2(sss,:)==1);
          [r,c]=size(nebrs);
          if numel(nebrs)>=1
            for i=1:c
               pin = nebrs(i);
               Rnebrs=cat(2,Rnebrs,adj_temp2(1,pin));
            end
          end
          pool=cat(2,pool,Rnebrs);
          pool=unique(pool);
          adj_temp2 = adj_temp1;
        else
          %Disconnected
          adj_temp1 = adj_temp2;
        end %iF conditional statement ends here
      end %while loop ends here
    end
  otherwise
     disp('Wrong choice! Please make a valid selection');
end

flaw=sort(flaw,'descend');
[~,ID]=find(flaw==0);
flaw(ID)=[];
for pointer=1:numel(flaw)
  i = flaw(pointer);
  [n,m] = find(GnodePos==i,1);
  GnodePos(n,m) = -1;
  A(n,m) = 0;
  VCmatrix(:,i)=[];
```

```
    adj(i,:)=[];
    adj(:,i)=[];
    deadNodes=deadNodes+1;
end
[r,c]=size(GnodePos);
 n=1;
 for i=1:r
   for j=1:c
      if A(i,j)==1
         GnodePos(i,j)=n;
         n=n+1;
      end
   end
 end
 no_nodes=n-1;
```

A.3 VCRA – Reliability check, VC update and performance evaluation

```
%INPUT: Network information
%OUTPUT: Routability, average hop count, total messages transmitted

%Compute data:-----------------------------------
%  The nodes calculate neighbor change if any after node failure event
% Nodes with neighbor change initiate reliability check
global Right_wrong;
Right_wrong = [];
nebrChange=[];
for n=1:no_nodes
    node_neighbors = [];
    node_neighbors = find(Oadj(n,:));
    nebrNumBefore(n)= numel(node_neighbors(1,:));
    node_neighbors = [];
    node_neighbors = find(adj(n,:));
    nebrNumAfter(n)= numel(node_neighbors(1,:));
    nebrChange(n) = nebrNumBefore(n) - nebrNumAfter(n);
end
for p = 1:numel(flaw(1,:))
    val = flaw(p);
    [i,j] = find(OGnodePos==val,1);
    if GnodePos(i,j)==(-1)
       nebrChange(val) = -1;   %if the node is dead..nebrChange=(-1)
    end
end
 no_anchors = numel(Anchors);
```

```
%reliability check:-------------------------------------------------------------------------------
for j=1:no_anchors
  for i=1:no_nodes
    Right_wrong(i,j)=1; %setting default value as 1
  end
end
%repeated  until we can no more find any update in Rel[]
unrelNodes=[];
checkDone = [];
nebrI = [];
%check the reliability for nodes with neighbor change  [ROUND #1]
  for i = 1:no_nodes
        if (nebrChange(i) > 0)
           [rows columns] = size(VCmatrix);
          for a = 1: rows    %check rel for coordinates corresponding to all the anchors
            %checks VCs for all the neighbors
            node_neighbors = [];
            node_neighbors = find(adj(i,:));  %getting its nebrs from new adjmatrix
            if ((numel(node_neighbors(1,:)))~=0)
             for n = 1:(numel(node_neighbors(1,:)))
                  nebr = node_neighbors(1,n);
                   if VCmatrix(a,nebr) == (VCmatrix(a,i)-1)
                     R = 1;
                     break;
                   else
                     R = 0;
                   end
             end
             nebrI = cat(2,nebrI,node_neighbors(1,:));
            %if for any neighbor, we dont find a VC == VC-1 then VC for current node is affected
             if ( R ==0 & (VCmatrix(a,i)~=0) )
                Right_wrong(i,a) = 0;
                unrelNodes = cat(2,unrelNodes,i);
             end
            end
          end
          checkDone = cat(2,checkDone,i);
        end
  end
  nebrI = unique(nebrI);    %taking out repetative elements

%now conduct the check for neighbor of nodes with neighbor-change-----go through nebrI
 nebrII=[];
       for n = 1:(numel(nebrI(:,:)))
       i = nebrI(n);
         if (~(any(i==checkDone)))
```

```
          node_neighbors = find(adj(i,:));
          [rows columns] = size(VCmatrix);
         for a = 1: rows
          %checks VCs for all the neighbors
          node_neighbors = find(adj(i,:));
          if ((numel(node_neighbors(1,:)))~=0)
           for n = 1:(numel(node_neighbors(1,:)))
                nebr = node_neighbors(1,n);
                 if VCmatrix(a,nebr) == (VCmatrix(a,i)-1)
                   R = 1;
                   break;
                 else
                   R = 0;
                 end
           end
           nebrII = cat(2,nebrII,node_neighbors(1,:));
          %if for any neighbor, we dont find a VC == VC-1 then VC for current node is affected
          if ( R ==0 & (VCmatrix(a,i)~=0) )
             Right_wrong(i,a) = 0;
             unrelNodes = cat(2,unrelNodes,i);
          end
          end
         end
         checkDone = cat(2,checkDone,i);
       end
       end
      %copy data from nebrII to nebrI
      nebrI = nebrII;
      nebrI = unique(nebrI);
      %notify nebr to check it's reliability
      zz = sum(nebrChange(:,:)>0);
      MSG = MSG + zz;

%Reliability check for remaining nodes..if any!
 for i = 1:no_nodes
    if (~(any(i==checkDone)))  % it checks if the node is already done with the reliability check
    node_neighbors = [];
    node_neighbors = find(adj(i,:));
    % loop to check VCs corresponding to all the anchors
    [rows columns] = size(VCmatrix);
         for a = 1: rows
          %checks VCs for all the neighbors
          node_neighbors = find(adj(i,:));
          if ((numel(node_neighbors(1,:)))~=0)
           for n = 1:(numel(node_neighbors(1,:)))
                nebr = node_neighbors(1,n);
```

```
                if VCmatrix(a,nebr) == (VCmatrix(a,i)-1)
                  R = 1;
                  break;
                else
                  R = 0;
                end
             end
             nebrII = cat(2,nebrII,node_neighbors(1,:));
            %if for any neighbor, we dont find a VC == VC-1 then VC for current node is affected
            if ( R ==0 & (VCmatrix(a,i)~=0) )
               Right_wrong(i,a) = 0;
               unrelNodes = cat(2,unrelNodes,i);
            end
            end
          end
          checkDone = cat(2,checkDone,i);
     end
 end
 nebrII = unique(nebrII);    %taking out repetative elements
unrelNodes = unique(unrelNodes);
relCount=0;
discovered=1;
unrelNebrs = [];
NewunrelNodes = [];
seen = [];
L1=0;
L2=1;

while L1<L2  %scanning nebrs of unRel nodes until we stop finding new unrel nodes
%list nebrs of the nodes that have rel[]=0 in previous round
%store nebrs of unrelNodes
   discovered = 0;
   for i = 1:numel(unrelNodes)
      pointer=unrelNodes(i);
      node_neighbors = [];
      node_neighbors = find(adj(pointer,:));
      unrelNebrs = cat(2,unrelNebrs ,node_neighbors(1,:));
   end
   unrelNebrs = unique(unrelNebrs);
   %check for reliability of these unreliable neighbors
   for i = 1:numel(unrelNebrs)
      pointer=unrelNebrs(i);
      node_neighbors = [];
      node_neighbors = find(adj(pointer,:));
     [rows columns] = size(VCmatrix);
        for a = 1: rows
```

```
          %checks VCs for all the neighbors
          node_neighbors = find(adj(pointer,:));
          if ((numel(node_neighbors(1,:)))~=0)
           for n = 1:(numel(node_neighbors(1,:)))
                nebr = node_neighbors(1,n);
                if Right_wrong(nebr,a) ==1;
                 if VCmatrix(a,nebr) == (VCmatrix(a,pointer)-1)
                   R = 1;
                   break;
                 else
                   R = 0;
                 end
                end
           end
          %if for any neighbor, we dont find a VC == VC-1 then VC for current node is affected
          if ( R ==0 && (VCmatrix(a,pointer)~=0) )
             Right_wrong(pointer,a) = 0;
             discovered = 1;
             NewunrelNodes = cat(2,NewunrelNodes,pointer);
          end
          end
        end
    end
    unrelNodes  = NewunrelNodes;
    L1=numel(seen);
    seen = cat(2,seen,unrelNodes);
    seen = unique(seen);
    L2=numel(seen);
    NewunrelNodes = [];
    unrelNodes = unique(unrelNodes);
 relCount=relCount+1;
 end
```

```
%Number of affected VCs (VCaffecetd) and number of affected nodes (Affected)are measured
here
z = sum(Right_wrong(:,:)==0);
VCaffected = z(1)+z(2);
Affected = [];
for p=1:no_nodes
  zz = sum(Right_wrong(p,:)==0);
    if zz>0
       Affected = cat(2,Affected,p);
    end
```

```
end
AffectedPool = Affected;


% Routing check ----------------------------BEFORE NODE FAILURE--*******
%destinationPool = all affected nodes
destinationPool = oldAffected;
%sourcePool = all non-affected nodes = (no_nodes)-(affected)
sourcePool = (1:Bno_nodes);
for i=1:numel(oldAffected)
   sample = oldAffected(i);
   if any(sourcePool==sample)
      [~,II]=find(sourcePool==sample);
      sourcePool(II)=[];
   end
end
%calculate the avg Routability percentage NON affetced to Affeted nodes prior
%to update
DDD = [3 6];
[N_A_avgRoutability_Original,N_A_avgHopCount_Original] =
route(sourcePool,destinationPool,DDD,BVCmatrix,Bno_nodes,Anchors,Badj);
[A_N_avgRoutability_Original,A_N_avgHopCount_Original] =
route(destinationPool,sourcePool,DDD,BVCmatrix,Bno_nodes,Anchors,Badj);


% Routing check ----------------------------------BEFORE VC update--*******
%destinationPool = all affected nodes
destinationPool = Affected;
%sourcePool = all non-affected nodes = (no_nodes)-(affected)
sourcePool = (1:no_nodes);
for i=1:numel(Affected)
   sample = Affected(i);
   if any(sourcePool==sample)
      [~,II]=find(sourcePool==sample);
      sourcePool(II)=[];
   end
end
%calculate the avg Routability percentage NON affetced to Affeted nodes prior
%to update
[N_A_avgRoutability_Pre,N_A_avgHopCount_Pre] =
route(sourcePool,destinationPool,DDD,VCmatrix,no_nodes,transAnchors,adj);
[A_N_avgRoutability_Pre,A_N_avgHopCount_Pre] =
route(destinationPool,sourcePool,DDD,VCmatrix,no_nodes,transAnchors,adj);


%--------------VC updating----------------------------------------
```

```
node_neighbors = [];
for i=1:numel(Affected)
   pointer = Affected(i);
   nebrs = [];
   nebrs = find(adj(pointer,:)); %stores all the nebrs of Affected nodes
   node_neighbors = cat(2,node_neighbors,nebrs);
end
boundary = [];
boundary = node_neighbors;
boundary = unique(boundary);
for i=1:numel(Affected)
   pointer = Affected(i);
   if any(boundary==pointer)
      zz = find(boundary==pointer);
      boundary(zz) = [];
   end
end
B=boundary; %these are the reliable nodes surrounding affected nodes

% number of nodes which undergo VC =0 are AFFECTED nodes.
%every node braodcasts its Rel[] to neighbors if  ==0
MSG = MSG + numel(Affected);
%when the unreliable node tells its neighbors that it is unreliable.. its neighbors check for
reliability again

%now we have Affected nodes and their boundary Reliability nodes
%Reliability nodes will try to update their unreliable neighbors

while (numel(boundary)>0)

   needed = [];
   for i=1:numel(boundary)
     pointer = boundary(i);
    nebrs = [];
    nebrs = find(adj(pointer,:));  %collect neighbors of Reliable nodes
      needed = cat(2,needed,nebrs);
   end
   needed = unique(needed);
   %keep only unrel nodes
   A=needed;
   for i=1:numel(A)
      pointer = A(i);
      if any(Right_wrong(pointer,:)==0)
      else
        zz=find(needed==pointer);
        needed(zz)=[];
```

```
      end
   end
MSG = MSG+numel(needed); %these unreliable nodes ask their neighbors for help-1 broadcast
%nodes in need get VC update from their Reliable neighbors
   for i=1:numel(needed)
      %process each unrel node
       pointer = needed(i);
       helpers = [];
       helpers = find(adj(pointer,:));
       %remove unrel nodes from nebr list
       A=helpers;
       for j=1:numel(A)
           if any(Right_wrong(A(j),:)==0) %remove the node if unrel
             zz=find(helpers==A(j));
             helpers(zz)=[];
           end
       end
       ME=needed(i);
       %See what VC the unrel node needs
       [rows columns] = size(VCmatrix);
       CHECK = [];
       for a = 1: rows
          CHECK(1,a) = 0; %stores zeros in CHECK as default value
       end
       STORE = [];
       for a = 1: no_anchors
          STORE(1,a) = -1; %default value in STORE is -1
       end
       for a = 1: rows
          if (Right_wrong(ME,a) == 0)
            CHECK(1,a) = -1; %this tells which VCs are faulty and needs to be looked for
            STORE(1,a) = -1;
          end
       end
       %look for the VCs needed
       for n = 1:(numel(helpers))
          nebr = helpers(n);
          for a = 1: rows %we check all the VCs at one nebr 1st and then move to other nebr's data
             if CHECK(1,a) == -1
                if STORE(1,a) == -1
                  STORE(1,a) = VCmatrix(a,nebr)+1; %Storing the VCs that we need
            else if (VCmatrix(a,nebr)+1) < STORE(1,a) %if we already got a Rel VC for what we
were searching then we store the one which is smallest
                  STORE(1,a) = VCmatrix(a,nebr)+1; %overwrite the prev value by the new smaller
one
               end
```

```
            end
          end
        end
      end
      %at the end of this loop, the unreliable node has smallest VC value derived
      %from its Reliable neighbor
      %update the unreliable node - pointer
      for a = 1: rows
        if (STORE(1,a) > -1)
          VCmatrix(a,ME) = STORE(1,a); %updating
        end
      end
      %also check if we have corrected R_w[] value for ME
      R=0;
      node_neighbors = [];
      node_neighbors = find(adj(ME,:));  %so neighbors are stored in this 1D matrix
      % loop to check VCs corresponding to all the anchors
      [rows columns] = size(VCmatrix);
      for a = 1: rows
        %checks VCs for all the neighbors
        node_neighbors = find(adj(ME,:));
        if ((numel(node_neighbors(1,:)))~=0)
          for n = 1:(numel(node_neighbors(1,:)))
            nebr = node_neighbors(1,n);
            if Right_wrong(nebr,a)==1
              if VCmatrix(a,nebr) == (VCmatrix(a,ME)-1)
                R = 1;
                break;
              end
            end
          end
          %if for any neighbor, we don't find a VC == VC-1 then VC for current node is
affecetd
          if R==1
            Right_wrong(ME,a) = 1;
          end
          if (sum(Right_wrong(ME,:)==0)==0) %if ME is completely updated
            zz=find(AffectedPool==ME);
            AffectedPool(zz)=[];       %remove it from the AffectedPool
          end
        end
      end
    end
    %at the end of this loop, all the 1st level unrel nodes are updated.
    %This is repeated and the wave is spread inwards
    %save the nebrs of unrel nodes - keep only unrel nodes and
```

```
   %repeat the process for them.
   boundary = needed;
   MSG = MSG + numel(boundary); %when a node changes into Rel, broadcast its nebrs
end %end the while loop here

%calculate the Routability percentage Affeted to NON affetced nodes post
%the update
[N_A_avgRoutability_Post,N_A_avgHopCount_Post] =
route(sourcePool,destinationPool,DDD,VCmatrix,no_nodes,transAnchors,adj);
[A_N_avgRoutability_Post,A_N_avgHopCount_Post] =
route(destinationPool,sourcePool,DDD,VCmatrix,no_nodes,transAnchors,adj);
```

A.4 Routing function

```
%INPUT: Set of source nodes, set of destination nodes, network information
%OUTPUT: Routability, average hop count

%% Routing function----------------------------------------------**********
% Tries to route messages from specified set of source and destination. Returns
% DDD is the pair of Directional Virtual Coordinates (DVCs) that gives best result for routing in
the given Wireless Sensor Netork


function[avgRoutability,avgHopCount] =
route(sourcePool,destinationPool,DDD,VCmatrix,no_nodes,transAnchors,adj)
avgHopCount = 0;
avgRoutability = 0;
RO = 0;

%% Calculating Anchor to anchor hop count:-----------------------------------------------
C = factorial(numel(transAnchors))/(factorial(2)*factorial((numel(transAnchors))-2));
n=1;
H=[];
for i=1:numel(transAnchors)
   VCa = VCmatrix(:,transAnchors(i));
   for j=i+1:numel(transAnchors)
      H(1,n) = i;
      H(2,n) = j;
      H(3,n) = VCmatrix(j,transAnchors(i));
      n=numel(H(3,:))+1;
      if numel(H(3,:))==C
        break;
      end
   end
```

```
end

%% Calculate DVCs for the nodes:----------------------------------------
DVC = [];
X = [];
Y = [];
for i=1:no_nodes
   for point=1:numel(H(3,:))
      a=H(1,point);
      b=H(2,point);
      h1 = VCmatrix(a,i);
      h2 = VCmatrix(b,i);
      DVC(i,point)= (h2-h1)*(h2+h1) /(2*H(3,point));
   end
end


%% get shortest hop distance matrix:-------------------------------------
%Dijkstra's algorithm is used to compute the VCs
%algorithm works on adjacency matrix for connectivity
arr = (1:no_nodes);
 S = sparse(adj);
for i = 1:numel(arr)
   Hs_matrix(i,:)=dijkstra_sp(S,arr(i));
end


%% routing process :-----------------------------------------------------------
 RO = 0; % gives number of messages routed successfully
 disp(H);
 Routed=0;
 TotalHopCount = zeros(1,numel(destinationPool));
 percent = zeros(1,numel(destinationPool));
 for d=1:numel(destinationPool)
    dest = destinationPool(d);
    Routed=0;
    HopCount = 0;
    RO = 0; %count routed msgs separately for each destination
    for s=1:numel(sourcePool)
       source = sourcePool(s);
       Routed=0;
       distance = [];
       jump = source;
       travelled=0;
    while (jump~=dest)
       distnebr = [];
```

```
        %compute odiff1[] and odiff2[]
        node_neighbors = [];
        node_neighbors = find(adj(jump,:));
        for ii=1:numel(node_neighbors)
           nebr = node_neighbors(ii);
           odiff1 = (DVC(nebr,DDD(1))-DVC(dest,DDD(1)))^2;
           odiff2 = (DVC(nebr,DDD(2))-DVC(dest,DDD(2)))^2;
           distance(ii) = sqrt(odiff1+odiff2);
        end
          node_neighbors = [];
          node_neighbors = find(adj(jump,:));
          %before we check for distance, check if the node index is in
          %nebrs
          if any(node_neighbors==dest)
             Routed = 1;
             RO=RO+Routed;
             %Routed!
             break;
          end
          [~,ID] = min(distance);
          node = node_neighbors(ID);
          jump = node;
          HopCount = HopCount+1;
          distance = [];
          travelled = travelled+1;
          if travelled>100
              break;
          end
          if jump==dest
             Routed = 1;
             RO=RO+Routed;
             %Routed!
             break;
          end

     end %while loop ends here
    end %for loop for each source ends here

    percent(d)=(RO*100/(numel(sourcePool)));
    [~,ID]=find(Hs_matrix(d,:)==Inf);
    Hs_matrix(d,ID)=0;
    Hs_avg = sum(Hs_matrix(d,:))/ numel(Hs_matrix(d,:)); %avg of shortest hop count for one
node
    TotalHopCount(d)=HopCount/Hs_avg;
 end % destination loop ends here
 %We take average of prercentage routability:
```

```
 avgRoutability = sum(percent)/numel(percent);
 avgHopCount = sum(TotalHopCount)/numel(TotalHopCount);

end %function ends here
```

APPENDIX B

SIMULATION OF MEDIAL AXIS DETECTION ALGORITHM

MATLAB code for simulating Medial Axis Detection Algorithm is given below. The code has been presented in parts where Section B.1 gives the code for obtaining the network connectivity information through adjacency matrix with 8 node neighbor connectivity. Section B.2 gives code for Round 1 of the algorithm. Section B.3 gives code for Round 2 of the algorithm.

B.1 Network connectivity information

```
 %The network is established using code in Section A.1. It also provides code for obtaining
adjacency matrix in case of 4 node connected network.
%INPUT: Network information
%OUTPUT: Adjacency matrix


% adjacency matrix stores the connectivity information of the network
% Section A.1 gives the adjacency matrix for a network with 4 neighbor connectivity
% 8- neighbor connected - adjacency:-----------------------------------
global adj;
adj=[];
row=r;
column=c;
 diagVec1 = repmat([ones(c-1,1); 0],r,1);  %# Make the first diagonal vector
                            %#   (for horizontal connections)
diagVec1 = diagVec1(1:end-1);            %# Remove the last value
diagVec2 = [0; diagVec1(1:(c*(r-1)))];    %# Make the second diagonal vector
                            %#   (for anti-diagonal connections)
diagVec3 = ones(c*(r-1),1);              %# Make the third diagonal vector
                            %#   (for vertical connections)
diagVec4 = diagVec2(2:end-1);            %# Make the fourth diagonal vector
                            %#   (for diagonal connections)
adj = diag(diagVec1,1)+...               %# Add the diagonals to a zero matrix
     diag(diagVec2,c-1)+...
     diag(diagVec3,c)+...
     diag(diagVec4,c+1);
```

```
adj = adj+adj.';                    %'# Add the matrix to a transposed
                                    %#   copy of itself to make it
                                    %#   symmetric
%substitute 0 for removed nodes
i=1;
for n = 1:row
   for m = 1:column
         if GnodePos(n,m) == 0;
            adj(i,:)=[];
            adj(:,i)=[];
            i = i-1;
         end
            i = i+1;
   end
end
```

B.2 Medial Axis Detection Algorithm – Round 1

```
%INPUT: Adjacency matrix, Boundary nodes
%OUTPUT: Medial nodes, total message transmissions during Round 1 of the algorithm

% Round 1: Limited flooding from boundary nodes to normal nodes
% Nodes store the shortest hop count received from any boundary node flooded packet
global R1cost;
global Bflood;
Bflood = 0;
for i=1:no_nodes
   Bdistance(i).ID=-1;
   Bdistance(i).hops=-1;
end
for B=1:numel(Boundary_nodes)
   BN = Boundary_nodes(B);
   Need=1;
   nebrs=BN;
   own_hop_count=1;
   while(Need==1)
        Need = 0;
        nebrsI=[];
        for zz=1:numel(nebrs)
           nn=nebrs(zz);
           addnn=find(adj(nn,:));
           nebrsI=cat(2,nebrsI,addnn);
        end
        nebrsI=unique(nebrsI);
```

```
    %take out bounadry nodes from the neighbors
        commons = intersect(nebrsI,Boundary_nodes);
        nebrsI = setxor(nebrsI,commons);
        %forwarding MSG to current nebrs
        if numel(nebrsI)>0 % then only proceed else move on to next boundary node
           Bflood = Bflood +1;
        for n=1:numel(nebrsI)
           node=nebrsI(n);
           if (Bdistance(node).hops>0)
              if (Bdistance(node).hops>own_hop_count)
                 Bdistance(node).hops=own_hop_count;
                 Bdistance(node).ID=BN;
                 Need=1;
              else if (Bdistance(node).hops==own_hop_count)
                    %else if 2 nodes are equidistant..Concatenate (cat) both IDs
                    Bdistance(node).ID=cat(2,Bdistance(node).ID,BN);
                    Need=1;
                 end
              end
           else
             %else overwrite the hop count value
             Bdistance(node).hops=own_hop_count;
             Bdistance(node).ID=BN;
             Need=1;
           end
        end
        own_hop_count = own_hop_count+1;
        nebrs = [];
        nebrs = nebrsI;
end %if condition includes the entire for loop

   end
end

%-------------------------------------------------------------------------

%% Self detect nodes whose nebrs have hops(>=self)....skeleton nodes
R1cost = 0; %number of messages required to execute Round 1

skeleton = zeros(1,no_nodes);
for i=1:no_nodes
   nebrs = [];
   nebrs = find(adj(i,:));
   %take out boundary nodes
   [~,ID]=find(Bflag(nebrs)==1);
   nebrs(ID)=[];
```

```
    count = 0;
    for n=1:numel(nebrs)
        if (Bdistance(nebrs(n)).hops<=Bdistance(i).hops)
            count = count+1;
        end
    end
    if count>0 && count==numel(nebrs)
        skeleton(i)=1;
    end
end
R1cost = R1cost + no_nodes - numel(Boundary_nodes);
MedialFlag = skeleton;
Medial_nodes = find(MedialFlag>0);
R1cost = R1cost + no_nodes - numel(Medial_nodes);
```

B.3 Medial Axis Detection Algorithm – Round 2

```
%INPUT: Adjacency matrix, Boundary nodes, Medial nodes from Round 1
%OUTPUT: Medial nodes, total message transmissions during Round 2 of the algorithm

% Round 2: Medial node inter-connection.....part A
global R2cost; %number of messages required to execute Round 1
R2cost = 0;
for NEED=1:3 % do until all medial nodes have at least 2 medial neighbors
  for i=1:numel(Medial_nodes)
    node=Medial_nodes(i);
    %check if the node needs to search for friends:
    nebrs=[];
     nebrs = find(adj(node,:));
     count=0;
     for mm=1:numel(nebrs)
         m=nebrs(mm);
         if MedialFlag(m)>0
            count=count+1;
         end
     end
     if count<2   %it means the search is needed...
    found=0;
    disp(node);
    TTL=1;
    path=[];
    nebrs=[];
    nebrs = find(adj(node,:));
     %eliminate old Medial neighbors to avoid backtracking
```

```
    flag=0;
    pointer=1;
    while(flag==0)
       flag=0;
       if pointer<=numel(nebrs)
         pin=nebrs(pointer);
       else
          break;
       end
       if MedialFlag(pin)>0
         ID = find(nebrs==pin);
         nebrs(ID)=[];
         flag=1;
       end
       pointer=pointer+1;
    end
past=node;
   while (TTL<11)
       if TTL>1
        %get nebrs
        nebrs=[];
        nebrs = find(adj(node,:));
        %eliminate PAST nebr to avoid backtracking - node
         flag=0;
         pointer=1;
         while(flag==0)
             flag=0;
             if pointer<=numel(nebrs)
               pin=nebrs(pointer);
             else
                break;
             end
             if pin==past
               ID = find(nebrs==pin);
               nebrs(ID)=[];
               flag=1;
             end
             pointer=pointer+1;
          end
        end
       %take out boundary nodes from nebrs
       commons = intersect(nebrs,Boundary_nodes);
       nebrs = setxor(nebrs,commons);
       %keep only the ONES -with higher Bdistance
       store=[];
       for zz=1:numel(nebrs)
```

```
            nn=nebrs(zz);
            store = cat(2,store,Bdistance(nn).hops);
          end
          [~,ID]=find(store==max(store));
          %if >1 nodes have max Bdist
          %store all
          senses=[];
          sensible_nebr = [];
          for ii=1:numel(ID)
             senses=cat(2,senses,nebrs(ID(ii)));
          end
          %store medial node if any
          if numel(senses)>0
          if any(MedialFlag(senses)>0)
             for ss=1:numel(senses)
                if MedialFlag(senses(ss))>0
                  sensible_nebr=senses(ss);
                  break;
                end
             end
          else
             sensible_nebr=senses(1);
          end
          end
          %check if this is a Medial node
          if (MedialFlag(sensible_nebr)>0)
            found=1; %Medial node found! route back and leave a trail
            %store the path of sensible_nebrss and mark all those if found
             if numel(path)>0
               MedialFlag =
traceroute(Medial_nodes(i),sensible_nebr,no_nodes,adj,MedialFlag,path,1);
                R2cost = R2cost + numel(path);
             end
             break;
          else
            past=node;
            node=sensible_nebr;
            path = cat(2,path,sensible_nebr);
          end
          TTL = TTL+1;
      end %while loop ends here
    end %for neccessity check
  end
  Medial_nodes = find(MedialFlag>0);
end% the main for loop of NEED ends here-do 3 times
```

```
%% ROUND 2 : Medial node inter-connection.....part B
%controlled search for other medial nodes in sensible directions
  for i=1:numel(Medial_nodes)
      node = Medial_nodes(i);
      destination = node;
      nebrsII = node;
      count = 1;
      found = 0;
      allowedNebrs = node;
      while (numel(allowedNebrs)>0)
           nebrsI = [];
           for nnn=1:numel(nebrsII)
             addI = find(adj(nebrsII(nnn),:));
             nebrsI = cat(2,nebrsI,addI);
           end
           nebrsI = unique(nebrsI);
            %take out past nodes from nebrs
            commons = intersect(nebrsI,nebrsII);
            nebrsI = setxor(nebrsI,commons);
            if count==1
               %take out present Medial nodes
               [~,ID] = find(MedialFlag(nebrsI)==1);
               nebrsI(ID)=[];
            end
            count = count+1;
            %take out boundary nodes from nebrs
            commons = intersect(nebrsI,Boundary_nodes);
            nebrsI = setxor(nebrsI,commons);
            allowedNebrs=[];
            for n=1:numel(nebrsI)
               found = 0;
               %allow nebrs only if (=current-1;NONmedial) or (=current;medial)
               if (Bdistance(nebrsI(n)).hops==(Bdistance(nebrsII(1)).hops-1)) &&
(MedialFlag(nebrsI(n))==0)
                  %save that nebr
                  allowedNebrs = cat(2,allowedNebrs,nebrsI(n));
               else if  (Bdistance(nebrsI(n)).hops==Bdistance(nebrsII(1)).hops) &&
(MedialFlag(nebrsI(n))==1)
                     %save that nebr
                     allowedNebrs = cat(2,allowedNebrs,nebrsI(n));
                  end
               end
            end
            R2cost = R2cost + numel(allowedNebrs);
           if numel(allowedNebrs)>0
             %stop the wave only if (=current;medial) or (=current;NONmedial)
```

```
            comparewith = nebrsII;
            for n=1:numel(allowedNebrs)
               if (Bdistance(allowedNebrs(n)).hops==Bdistance(comparewith(1)).hops) &&
(MedialFlag(allowedNebrs(n))==1)
                  %destination found
                  found = 1;
                  mark = allowedNebrs(n);
                  break; %breaks FOR loop
               else
                  nebrsII = allowedNebrs;
               end
            end
         end
         if found==1
            break; % breaks WHILE loop
         end
      end%end of while loop
      if (found==1)
         %back track from mark to destination
         nebrs = [];
         nebrs = find(adj(mark,:));
         past = mark;
         %eliminate old nebrs to avoid backtracking
         [~,ID] = find(nebrs==past);
         nebrs(ID)=[];
         %eliminate old Medial nodes
         [~,ID] = find(MedialFlag(nebrs)==1);
         nebrs(ID)=[];
         %take out boundary nodes from nebrs
         commons = intersect(nebrs,Boundary_nodes);
         nebrs = setxor(nebrs,commons);
         scan = nebrs;
         nebrs = [];
         %keep only sensible nebrs...hop=self
         for m=1:numel(scan)
            if (Bdistance(scan(m)).hops==Bdistance(mark).hops)
               nebrs = cat(2,nebrs,scan(m));
            end
         end
         nebrs = unique(nebrs);
         %if we find medial node in the 1st hop itself.. that means this
         %connection is not required
          if any(MedialFlag(nebrs)==2)
             %do nothing and we will go to next medial node
          else
             %jump to any 1
```

```
            jump = nebrs(1);
            R2cost = R2cost + 1;
             while(jump~=destination)
                MedialFlag(jump)=2; %implies that these are newly formed Medial nodes
                nebrs = [];
                nebrs = find(adj(jump,:));
                %take out boundary nodes from nebrs
                commons = intersect(nebrs,Boundary_nodes);
                nebrs = setxor(nebrs,commons);
                %remove past nodes to avoid backtracking
                [~,ID] = find(nebrs==past);
                nebrs(ID)=[];
                scan = nebrs;
                nebrs = [];
                %keep only sensible nebrs...hop=self
                for m=1:numel(scan)
                  if (Bdistance(scan(m)).hops==(Bdistance(jump).hops)+1)
                    nebrs = cat(2,nebrs,scan(m));
                  end
                end
                %check if any of it is Medial
                if any(MedialFlag(nebrs)==2)
                   [~,ID] = find(MedialFlag(nebrs)==2);
                   jump = (nebrs(ID(1)));
                   destination=jump;
                   R2cost = R2cost + 1;
                else if any(MedialFlag(nebrs)==1)
                   [~,ID] = find(MedialFlag(nebrs)==1);
                   jump = (nebrs(ID(1)));
                   R2cost = R2cost + 1;
                   else
                   %else ... jump to any 1
                   past = jump;
                     if numel(nebrs)>0
                        jump = nebrs(1);
                        R2cost = R2cost + 1;
                     else
                         break;
                     end
                   end
                end
             end %while loop ends here
          end %need check if condition ends here
       end
end% the main for loop of all the Medial nodes
```

```
% Traceroute function:
% the function routes from given source to given destination and marks the path nodes as medial
nodes

function[MedialFlag] =
traceroute(sourcePool,destinationPool,no_nodes,adj,MedialFlag,path,val)
%% routing-----------------------------------------------------------

 RO = 0; % gives number of messages routed successfully

 Routed=0;

 for d=1:numel(destinationPool)
    dest = destinationPool(d);
    Routed=0;
    HopCount = 0;
    RO = 0; %count routed messages separately for each destination
    for s=1:numel(sourcePool)
       source = sourcePool(s);
       Routed=0;
       distance = [];
       jump = source;
       travelled=0;
       TTL=0;
       past=jump;
    while (jump~=dest)
       distnebr = [];

       node_neighbors = [];
       node_neighbors = find(adj(jump,:));

         node_neighbors = [];
         node_neighbors = find(adj(jump,:));

         %check if the destination node is in neighborhood
         if any(node_neighbors==dest)
            Routed = 1;
            RO=RO+Routed;
            break;
         end

         %look for sensible_nebrs androute
         commons = intersect(node_neighbors,path);
           if numel(commons)>0
               %remove past nodes from commons
               if any(commons==past)
```

```
            xx=find(commons==past);
            commons(xx)=[];
          end
        end
          past=jump;
          jump = commons;
          TTL=TTL+1;
          if TTL>20
             break;
          end
          MedialFlag(jump)=val;  %marks the nodes in path as Medial nodes

      distance = [];
      travelled = travelled+1;
      if jump==dest %routing successful!!
        Routed = 1;
        RO=RO+Routed;
        break;
      end
   end %while loop ends here
  end %for loop for each source ends here

 end % destination loop ends here


end %function ends here
```

LIST OF ABBREVIATIONS

CASE	Connectivity Based Skeleton Extraction
DIST	Distance Transformed-Based Skeleton Extraction
GC	Geographical Coordinate
GPS	Global Positioning System
ID	Identification Detail
MALD	Medial Axis detection scheme for Low Degree networks
MAP	Medial Axis based naming and routing Protocol
TTL	Time To Live
VC	Virtual Coordinate
VCap	Virtual Coordinate assignment protocol
VCS	Virtual Coordinate System
WSN	Wireless Sensor Network
VCRA	Virtual Coordinate Repair Algorithm