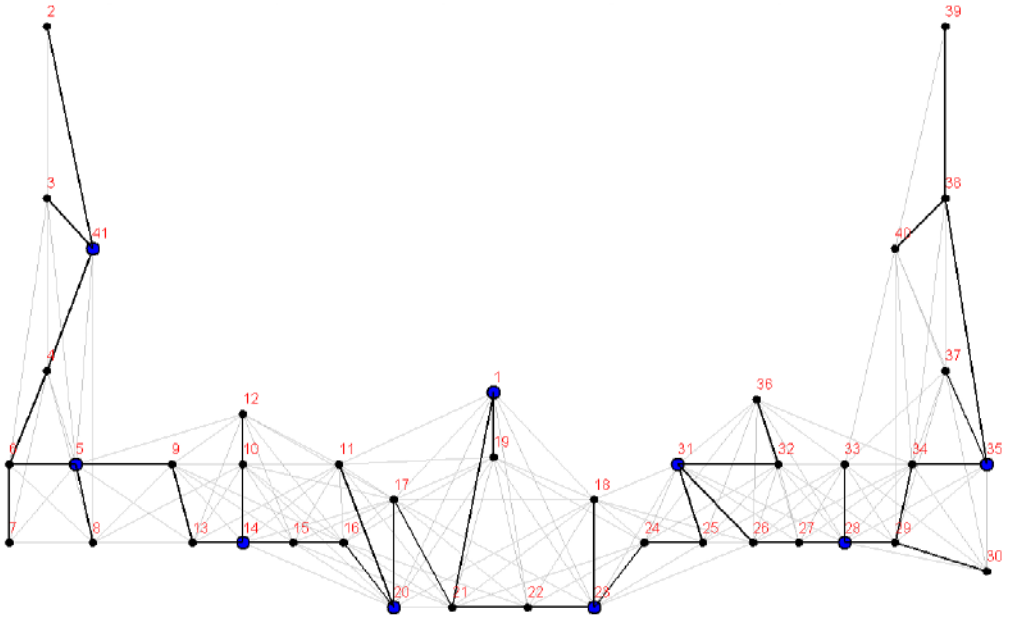
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Routing Algorithms for Safety Critical Wireless Sensor Networks

Tunc Ikikardes

Master Thesis, SS-2006



Advisors:
Dr. Markus Hofbauer, Siemens
Dr. August Kälin, Siemens
Dr. Martin May

Professor:
Prof. Bernhard Plattner

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

For Elena and
My family

Acknowledgement

This thesis gave me the opportunity to see the practical importance of the theoretical findings. Our research had its motivation from the real-world fire detection problem but it was the *graph theory* that helped us to solve the basic problems and to proceed further. There I was convinced that the cooperation of the academy and industry plays an essential role for the technological development. During this period, I also gained insight of the industry, where one has to face the practical and real problems which can be hardly found in the textbooks. After all, I have to thank to Siemens and ETH for providing me this precious experience.

I would like to thank deeply Dr. A. Kaelin not only for making this thesis possible for me but also for his stimulating suggestions, encouragements and providing me the necessary freedom to organize my work. I am very grateful to Dr. M. Hofbauer for his continuous support, his concise comments and for all the fruitful discussions. I also give my sincere thanks to Professor B. Plattner and Dr. M. May for their valuable comments and suggestions in our meetings.

I would like to thank my friend Musa for sharing all those experiences in the master study at D-ITET. I am deeply indebted to my family for their endless support and making the education in ETH possible for me. At last, I wholeheartedly thank to Elena for all the compromises and her understanding.

Table of Contents

Abstract

In this thesis, routing algorithms and optimal topologies for the safety critical wireless sensor networks and in particular for the fire detection application called CWSN are considered. As the wireless sensors are battery driven, these networks are very power-sensitive. On the other hand, because of the safety critical aspect, they require low latencies for communication. To match these two challenging and also conflicting requirements, we focused on tree-based structures in particular spanning tree algorithms for routing.

Most of the standard tree-based algorithms require global knowledge. Hence, a generic routing algorithm called *Pairing Algorithm* was developed to localize standard global algorithms such as *Minimum Spanning Tree* (MST) and *Shortest Path Tree* (SPT) algorithms. To be able to keep the power within the required limits of the CWSN application ($P_{max} \approx 100\ \mu W$) we chose a static routing algorithm that can handle link breaks without re-routing the network. A generic algorithm termed *N-SafeLinks Algorithm* was devised that can construct routing trees with different cost functions and generate lookup tables for the routes. The algorithm guarantees connectivity and loop-freeness (a tree-structure) even in case of multiple link breaks (up to N-1 links per node). In order to evaluate the existing algorithms and to develop new algorithms, we utilized a graph analyzer tool (GUI) that we have implemented in MATLAB.

As an optimal power-efficient topology for the CWSN application, the *Double-Tree Concept* was devised, where a fast backbone tree is used for the alarms and a power- efficient monitoring tree is used for normal operation. With this concept, the power efficiency is increased while keeping alarm latencies under the required limits. With the introduced *Local Sinks Concept*, subtrees (clusters) are generated in the monitoring tree. These subtrees are monitored with special nodes called *Local Sinks*. This concept further increases the power efficiency and was implemented in the *Local Sinks Algorithm*.

Extensive simulations and comprehensive analyses of the developed algorithms were carried out. Compared to the standard algorithms, the loss in terms of the metric and the required node connectivity are found to be limited and in practice irrelevant.

The network and application layers of the CWSN were implemented in GloMoSim. The simulation results indicate that the required timeliness is met while achieving a power consumption of about $100\mu W$. At the same time the robustness against link breaks is provided.

1. Introduction

1.1 Wireless Sensor Networks (WSNs)

Wireless Sensor Networks (WSNs) are computer networks that are comprised of distributed nodes using a wireless communication module. The nodes of a WSN have in particular a sensor module to monitor their environment, e.g., temperature, sound, pressure. The wireless nodes usually have a microcontroller to process the environmental conditions that they sampled with their sensors and these signals are processed to detect certain events, e.g., fire, explosion, glass crash. The nodes of the WSNs exchange information with other nodes in the network through wireless links. Most of the WSNs have several wireless sensor nodes and all these nodes cannot have feasible direct links to each other and hence they are mostly organized in a *multi-hop* fashion where the information transfer between nodes is realized by hopping through other relay nodes. As the nodes do not have any wired connection they are usually battery driven. Hence the power is an essential limiting factor for the wireless sensor networks and it must be considered for the organization of the WSN.
Wireless Sensor Networks have gained importance in recent years because of their simplicity and growing functionality. The commercial WSN products are spread out and many researchers are working on different levels of WSNs to add them more functionality and to make them more efficient. The application areas of the WSNs varies from biomedical applications and seismic detection systems to military applications and area monitoring for buildings. Today, WSNs are replacing the old platforms in many different application areas and they are also introducing new application possibilities which were not realizable with the old wired systems.

1.1.1 Routing in WSNs

The focus of this thesis is on the area safety applications of WSNs, e.g., fire detection systems. These applications use several wireless sensor nodes that are spatially distributed in a building and a sink node that gathers data from the sensors and process it. In order to provide the data collection every sensor has to find the path to the sink in order to transmit its observations. Finding the optimum paths to reach the sink introduces the *routing problem* for the WSNs. The paths must minimize communication costs (power) but they must be robust and provide low latencies. Moreover, a routing algorithm that determines the optimum routing paths cannot be to complex as it must be implemented in the simple nodes on a microcontroller. To decrease the installation efforts, it is demanded that the routing algorithm can perform in a distributed fashion without needing global knowledge such as exact node coordinates, link qualities etc. At last the routing algorithm must be power efficient and it should produce low communication traffic.

1.2 Problem Statement

1.2.1 Critical Wireless Sensor Network

A Critical Wireless Sensor Network (CWSN) is wireless multi-hop sensor network with several sensor nodes and one sink node. The sensors sample the environment and inform the sink if there is a specific event or not, e.g., fire, glass crash, etc. In case of such an event, the sink receives an alarm and it informs the necessary units such as police and fire department.

1.2.2 General Requirements

For CWSN applications there are two main issues:

- The first issue is *power efficiency*. It is very critical because the nodes are battery driven and power efficiency determines directly the network's life. For today's networks, 3 to 5 years of network life can be demanded. It is important to note that a WSNs life is limited by the worst-case node. Hence the power consumption should be very low and it must be distributed uniformly over the nodes in the network.

- The second limiting factor is the *timeliness*. As these applications are mostly safety critical, the alarms must arrive within a specified interval to the sink. Moreover, the sink must be informed within a specified time, if some of the nodes crash or there are permanent communication failures.

1.2.3 A Specific CWSN Application for Fire Detection and its Requirements

In this specific fire detection WSN application [1], the multi-hop wireless network is comprised of fire detection nodes and a sink node. Every fire detection node notifies the sink if there is a fire event. During normal operation, where there is no fire event, the nodes send periodic messages over hops to the sink, indicating that they are present. This CWSN application is comprised of two phases:

- The first phase is the *initialization phase*, which starts when the fire detection nodes are turned on. In the initialization phase, the nodes have to organize themselves, i.e., they have to decide on their next hop on the path to sink and synchronize themselves according to some pattern.

- The second phase is called *operation phase* and it starts when the network is organized and when the sink decides that the operation can start. In this phase, the normal operation takes place.

The wireless fire network requirements are as follows

- technical node failures have to be detected within τ_{Fail} = 300s at the sink node. This requires periodic monitoring messages.
- a fire alarm has to be reported to the sink node within τ_{Fire} = 10s.
- the fire has to be indicated by certain nodes, controlled by the sink within $\tau_{Indicator}$ = 20s.
- the total power consumed in the normal operation phase should be in the order of P_{target} = 100 μW for the worst case node.
- the protocol should use *C* different frequencies and handle up to C_{Fail} channel failures.
- the routing algorithm should handle several sudden link breaks, at most 1 link per node.

Symbol	Description	Value
τ_{Fail}	Timing constraint for the notification of a node crash	300s
τ_{Detect}	Timing constraint for the interval of sensing the environment	2s
τ_{Fire}	Timing constraint for the notification of a fire condition	10s
$\tau_{Indicator}$	Timing constraint for the indication of a fire condition	20s

P_{target}	Target energy consumption	100 µW
C	Number of channels	16
C_{Fail}	Number of maximum channel failures	4
B	Bitrate	10kbps

Table 1: Requirements for the fire detection network protocol[1] [1].

1.2.4 Requirements for the Routing Algorithm

The requirements of the FD-WSN application bring strict limitations on timing and power consumption in the network. Therefore, there is a need for a very power efficient routing protocol that must be reactive enough to provide timeliness even in case of communication failures. The protocol should determine the routing graph, which sets the next hop, i.e., parent. The protocol must be able to react and find an alternative parent when link breaks occur. The routing should organize the nodes in an optimal way for low power consumption and it must not need many message exchanges and frequent re-routing in order to keep power consumption low. Also the network should be organized in a proper way that it fulfills the low latency requirements given in Table 1.

1.3 Network Model and Notations

The wireless sensor network is modeled as a graph $G(V,E)$ with the nodes or vertices V, among which there is one sink and the edges E. There is a link $\{i, j\} \in E$ when node i and node j can communicate with each other directly. Every edge $\{i, j\} \in E$ has a weight w_{ij}, which is the communication cost for that link. The communication is assumed to be symmetric[2], namely $w_{ij} = w_{ji}$. The routing algorithms described in this thesis find a subgraph $T(V,E_T)$ of G, with nodes V and edges E_T (much more sparse than E). T is a tree with a sink as the root node. The path from the node j to the sink is denoted as P_j and is defined as the concatenation of all the edges on the path from the node j to the sink, i.e.,
$P_j = \left(\{j, node_1\}, \{node_1, node_2\}, ..., \{node_n, sink\}\right)$.

1.4 Contributions

The contributions of this thesis can be listed as follows:

- A new routing algorithm called *Pairing Algorithm* is introduced and verified that localizes the standard optimum global routing algorithms. The developed algorithm is generic, i.e., it can be utilized with different cost functions. The performance of the algorithm is found to be very close to the standard optimal algorithms with the respective cost function.

[1] For the complete list of the protocol parameters refer to **Fehler! Verweisquelle konnte nicht gefunden werden.**

[2] In reality the communication cost of links are not strictly symmetric, especially in the closed environments, e.g., buildings, factories. However for the given problem description [1] and from the scenarios provided from SBT we limit ourselves with symmetrically weighted graphs.

- A generic routing algorithm called *N-SafeLinks Algorithm* is developed. The algorithm provides robustness against link breaks and it guarantees the connectivity of the graph in case of link breaks (up to N-1 links per node). A mathematical proof is given for the loop-freeness of the algorithm.

- For the CWSN application, a topology concept called the *Double-Tree Concept* is developed that introduces one fast alarm tree to minimize the latency of transmissions and one monitoring tree to minimize the power consumption during the normal operation.

- For the CWSN application, a supervising concept called the *Local Sinks Concept* is devised. The concept distributes the tree into clusters and realize the monitoring of clusters separately to maximize power efficiency. An algorithm called *Local Sinks Algorithm* is also introduced that utilizes this concept.

- A GUI termed *Graph Analyzer Tool* is developed in MATLAB. The tool is used to develop, display, test and simulate the introduced routing algorithms and it is also used to compare these algorithm with the standard routing algorithms.

- Extensive simulations were carried out to verify the developed algorithms to compare their differences to the standard algorithms and their promised properties, e.g., low power consumption.

- The network and application layers of the CWSN were implemented in GloMoSim. The verification tests for robustness and power efficiency were carried out and the results were analyzed.

1.5 Outline

The thesis is organized as follows:

- **2. Chapter:** The existing routing algorithms are analyzed and discussed. They are compared with each other and the algorithms that are promising for further development were selected.

- **3. Chapter:** The developed *Pairing* and *2-SafeLinks* algorithms are introduced and explained. The necessary theoretical analyses are presented.

- **4. Chapter:** The CWSN specific topology concepts, *Double-Tree Concept* and *Local Sinks Concept* are presented and analyzed

- **5. Chapter:** The MATLAB Graph Analyzer Tool is introduced. The simulation results for developed algorithms and the comparisons with standard algorithms are presented and discussed.

- **6. Chapter:** The implementation of the network and application layers on the GloMoSim is given. At last the simulations results obtained from GloMoSim are summarized.

- **7. Chapter:** The yields of the thesis are discussed and the conclusions are presented.

2. Existing Routing Algorithms for WSNs

2.1 Overview

There is an excessive literature on the routing problems in WSNs. However, because of the strict requirements of the concerning CWSN application (see sections 1.2.2 and 1.2.3) only a few of the existing algorithms are suitable and will be considered. The application uses only one source and the low data rate allows accumulation of data, therefore we limit our research on the algorithms searching for a routing tree with the sink node as root. In a WSN, the nodes should organize themselves automatically, hence the algorithms should be locally implementable. The application is very power sensitive, thus the algorithms, which work statically or the algorithms which need less re-routing are favorable. Before the existing algorithms are explored in sections 2.2 - 2.4 and discussed in section 2.6 , we want to give some preliminary definitions that are frequently used in the rest of the chapter.

2.1.1 Definitions

Definition 1: Spanning Tree

For a graph $G=(V,E)$ where V is the set of the vertices on the graph and E is the set of the edges a *Spanning Tree* is a subgraph $G_{ST}=(V,E_{ST})$ of the graph G. E_{ST} is a subset of E , such that the subgraph G_{ST} is a tree that contains all the vertices. A graph G does not have to have a unique spanning tree.

Definition 2: Minimum Spanning Tree (MST)

For a graph $G=(V,E)$ where V is the set of the vertices on the graph and E is the set of the edges a *Minimum Spanning Tree* is a subgraph $G_{MST}=(V,E_{MST})$, where E_{MST} is a subset of E , such that the subgraph G_{MST} is a spanning tree that minimizes the total weight of G_{ST}, which can be formulated as

$$E_{MST} = \arg\min_{E_{ST}} \left(\sum_{\forall \{i,j\} \in E_{ST}} w_{ij} \right) , \tag{1}$$

where E_{ST} is a subset of E that constructs with all vertices a spanning tree, $\{i,j\}$ is a single edge in E_{ST} and w_{ij} is its weight. A *Minimum Spanning Tree* that fulfils equation (1), must not be unique but for a graph there exists no other spanning tree that reduces the total weight any further than its *Minimum Spanning Tree*.

Definition 3: Shortest-Path Tree (SPT)

For a graph $G=(V,E)$ where V is the set of the vertices on the graph with a special vertex called sink and E is the set of the edges a *Shortest Path Tree* is a subgraph $G_{SPT}=(V,E_{ST})$, where E_{SPT} is a subset of E, such that the subgraph G_{SPT} is a spanning tree that minimizes the weight of every path from a node to the sink in G_{SPT}, which can be formulated as

$$E_{SPT} = \bigcup_{\forall j \in V} \left(\arg\min_{P_j} \left(\sum_{\forall \{i',j'\} \in P_j} w_{i'j'} \right) \right) , \tag{2}$$

where E_{SPT} is a subset of E and E_{SPT} constructs with all vertices a spanning tree, P_j is the path from the node j to the sink and is comprised of the single edges. The weight of the edge $\{i',j'\}$ is denoted by $w_{i'j'}$. Any tree that fulfils equation (2) is a *Shortest-Path Tree* of the graph G.

2.2 Minimum Spanning Tree (MST) Algorithms

2.2.1 Prim's Algorithm

This algorithm is developed for weighted undirected graphs with a single source and is a *Minimum Spanning Tree* algorithm. It is discovered by computer scientist Robert Prim in 1957 [6]. It starts from a source node or the sink which is the initial partial spanning tree. At each step the algorithm inserts the node which has the minimum-cost link to the current partial spanning tree via this minimum-cost link. The algorithm lets the partial spanning tree grow until all the nodes are added and the complete spanning tree is constructed. It is a *greedy* algorithm, in the sense that at each iteration one new link is added to the current tree. The algorithm also inherits the tendency to build up trees with less branching but many hops. This situation is a result of the minimum spanning tree property as relaying the connections over other nodes is cheaper than direct links in terms of link weights. The algorithm is given below.

2.2.1.1 *Algorithm 1: Prim's Algorithm*

Inputs:
E : Set of edges in the graph
V : Set of vertices
w : Set of edge-weights
Sink: Source node

Outputs:
E_{MST} : Edges of minimum spanning tree

Temporary Variables:
T : Set of the vertices added to the *MST*
T' : Set of the vertices not added to the *MST* yet
BestLink : The best link found at an iteration

Initialization:
$T := \{Sink\}$
$T' := V / \{Sink\}$
$E_{MST} := \{\}$

Main Part:
While $T \sim= V$
  For $\forall i \in \boldsymbol{T'}$
    For $\forall j \in \boldsymbol{T}$
      If $W_{ij} > BestLink$
        $BestLink := \{i,j\}$
      End
    End
  End
  $T := T \cup \{j\}$
  $T' := T' / \{j\}$
  $E_{\mathrm{MST}} := E_{\mathrm{MST}} \cup \{BestLink\}$
  $BestLink := \{\ \}$
End

2.2.1.2 *Complexity*

The Prim algorithm finds the MST with the given basic implementation in $O(mn)$ time, where n is the number of nodes in the network and $O(m)$ is the time complexity of each iteration [8]. However using binary or Fibonacci heaps, the time complexity can be reduced to $O(m \log n)$ or $O(m + n \log n)$, respectively [9].

2.2.2 Borůvka's Algorithm

This is the earliest found MST algorithm, which was invented by the Czech mathematician Otokar Borůvka in 1926. The algorithm starts with all single nodes that do not have any connections to each other. At the first part of the algorithm, each node selects its best edge in its neighborhood. The nodes which are joined are termed *supernodes*. In the further iterations each *supernode* chooses their smallest weighted edge to other *supernodes* and with these new edges, the *supernodes* unite. At the last iteration only one *supernode* remains which is the MST of the initial graph [8].

2.2.2.1 *Algorithm 2: Borůvka's Algorithm*

Inputs:
E : Set of edges in the graph
V : Set of nodes in the graph
w : Weights of the edges
N : Number of nodes in the graph

Outputs:
E_{MST} : Edges of minimum spanning tree

Temporary Variables:
SNodes : List of supernodes

F : The minimum weight link found between supernodes at some iteration

Initialization:
SNodes := {$node_1$,$node_3$, …, $node_N$}
F := {}
E_{MST} := {}

Main Part:
While *number of elements in* $E_{MST} < N - 1$
F := **FindBestLinkBetweenSupernodes (***SNodes, G*(*V*,*E*)**)**
SNodes := **Union (***SNodes*,*F***)**
$E_{MST} := E_{MST} \cup F$
F := { }
End

Sub-Routines:
Union (*SNodes*,*F***) :** it is a sub-routine that merges the supernodes of the list *SNodes* linked with list of links F and return the updated supernodes list *SNodes*.

FindBestLinkBetweenSupernodes (*SNodes, G*(*V*,*E*)**) :** it searches the minimum weighted edge for each supernode given in the list *SNodes* to the other from the information of graph *G*(*V*,*E*). It returns the found edge, if there is any.

2.2.2.2 *Complexity*

The algorithm finds the MST in $O(m \log n)$ time, where n is the number of nodes in the network. The while-loop is repeated at most $O(\log n)$ times as at each iteration number of remaining edges are reduced by a factor at least 2 and $O(m)$ is the time complexity of each iteration [8].

2.2.3 Kruskal's Algorithm

This algorithm was found by the mathematician Joseph Kruskal in 1956. The algorithm is a MST algorithm that can be applied on any weighted, undirected graph *G*(*V*,*E*). The algorithm starts with a subgraph of the graph *G*. This subgraph is initially a forest which contains all single nodes and no edges. After sorting all the edges in the ascending order, the algorithm goes over all these edges and add the ones, who joins two different partial trees, until all the nodes are united and the subgraph forms a spanning tree that is a MST [8].

2.3 Shortest Path Tree (SPT) Algorithms

2.3.1 Dijkstra's Algorithm

Dijkstra's Algorithm was found by the computer scientist Edsger Dijkstra in 1959 [7]. It is a single source shortest path tree algorithm. The algorithm is developed for directed or undirected, connected and weighted graphs, where the weights are strictly non-negative. This algorithm starts with a partial spanning tree that has initially only the source node. At each iteration a node is added to the current partial spanning tree in a way that the path from every node to the sink is minimized. At each iteration the current tree grows by the addition of one node with a link. Hence this algorithm is a *greedy* algorithm. The idea of a growing partial

spanning tree is very similar to *Prim's Algorithm*, the main difference is the metric these two algorithms try to minimize.

2.3.1.1 *Algorithm 3: Dijkstra's Algorithm*

Inputs:
E : Set of edges in the graph
V : Set of vertices
w : Set of edge-weights
Sink: Source node

Outputs:
E_{SPT} : Edges of minimum spanning tree

Temporary Variables:
T : Set of the vertices added to the *SPT*
T' : Set of the vertices not added to the *SPT* yet
M_i : The distance of node *i* to *Sink*, i.e., total cost of the path to *Sink.*
BestLink : The best link found for an iteration

Initialization:
$T := \{Sink\}$
$T' := V / \{Sink\}$
$E_{SPT} := \{\ \}$
$M_i := \{\ \}$

Main Part:
While $T \sim= V$
  For $\forall i \in \boldsymbol{T'}$
    For $\forall j \in \boldsymbol{T}$
      If $W_{ij} + D_i > BestLink$
        $BestLink := \{i,j\}$
      End
    End
  End
  $T := T \cup \{j\}$
  $T' := T' / \{j\}$
  $E_{MST} := E_{MST} \cup \{BestLink\}$
  $M_j := W_{ij} + D_i$
  $BestLink := \{\ \}$
End

2.3.1.2 *Complexity*

The Dijkstra algorithm finds the SPT with the given basic implementation in $O(mn)$ time, where n is the number of nodes in the network and $O(m)$ is the time complexity of each iteration [8]. However with modifications, the complexity can be reduced to $O(m + n\log n)$ [9].

2.3.2 Bellman-Ford (BF) Algorithm

This algorithm is found by the mathematician Richard Bellman in 1958 . It is a single-source shortest path algorithm, which can be applied in more general cases than *Dijkstra's Algorithm*. The basic advantage of the *Bellman-Ford Algorithm* is that it can also work with graphs that have negative edge weights whereas *Dijkstra's Algorithm* can only find the *Shortest Path Tree* when the edge-weights are strictly non-negative. The algorithm assigns to each node a shortest-path to source value. In opposition to *Dijkstra's Algorithm*, the *Bellman-Ford Algorithm* updates the shortest path values for each node more than once[3]. The shortest paths are determine only after all updates are finished. The algorithm finds a shortest path tree for every case except if there exist cycles comprised of only negative edges in the graph [8].

2.3.2.1 *Algorithm 4: Bellman-Ford Algorithm*

Inputs:
E : Set of edges in the graph
V : Set of vertices
W : Set of edge-weights
Sink : Source node
N : Number of nodes in the graph

Outputs:
E_{SPT} : Edges of minimum spanning tree

Temporary Variables:
D_i : The distance of node i to *Sink*, i.e., total cost of the path to *Sink*.
$NHop_i$: The next hop of the node i

Initialization:
$E_{SP\,T}$:= { }
D_i := { }, $\forall i \in V$
$NHop_i$:= { }, $\forall i \in V$

Main Part:
For *step*:=1 **to** N-1
  For $\forall i \in V$
    For $\forall j \in V$
      If $D_j > D_i + w_{ij}$
        $NHop_j := i$
        $D_j := D_i + w_{ij}$
      End
    End
  End
End

For $\forall i \in V \setminus Sink$
  $E_{MST} := E_{MST} \cup \{i, NHop_i\}$
End

[3] Exactly *number of nodes* – 1 times

2.3.2.2 *Complexity*

The Bellman-Ford algorithm has $O(n)$ iterations, where n is the number of nodes in the network. Each iteration takes $O(m)$ time, hence the overall time complexity of the algorithm becomes $O(mn)$ [8].

2.4 Distributed Algorithms

2.4.1 Distributed Bellman-Ford (DBF) Algorithm

This algorithm is an asynchronous distributed version of the Bellman-Ford Algorithm and it is found by D. Bertsekas in 1987 [10]. The algorithm models the communication network as a weighted graph, where each link has a certain cost, which might fluctuate in time. In this algorithm, the nodes exchange asynchronously their *routing vectors*. The *routing vector* of a node contains several entries. Each entry is a pair: a destination node ID and the shortest path to this node. The shortest paths are calculated by Bellman-Ford iteration, see Algorithm 4. Each entry in the *routing vector* is a message and whenever a node updates its routing vector, it sends a broadcast message to its neighbors. Every node i has an additional distance matrix $D_i = [D_{ij}^k]$ where each row j is a destination and each column k is a neighbor. For D_i, the $\{j,k\}^{\text{th}}$ entry is distance of node i to node j, if the next hop is the node k. During execution of the DBF algorithm, every node fills its distance matrix. After the algorithm terminates, every node can deduce from its distance matrix, its next hop for each destination. The neighbor, i.e., the column in the distance matrix, that has the minimum distance to the given destination node is assigned as the next hop for that destination.

If the link qualities change, the nodes having this link, update their distance matrices and exchange the updated *routing vectors* with their neighborhood. In case of a link break between node i and j, these nodes delete each other from their *distance matrix*, i.e., node i deletes its j^{th} column and the node j deletes the i^{th} column of its *distance matrix*. Thereafter a re-routing, i.e., exchange of *routing vectors* and update of new distances in *distance matrix*, is triggered. If this link becomes available again, these deleted columns are inserted and a new re-routing is started.

The DBF algorithm converges whenever the graph is connected however it may converge very slowly as it suffers from the *bouncing effect*, where a node keeps on increasing its distance through a neighbor, which does not actually have a path to the specific destination. This situation can occur in case of link breaks. A worse situation occurs, when the node suffering the bouncing effect does not have any other alternative next hop for the specific destination. In this case, both nodes select each other as next hop and they keep on increasing their distances without any bound, which is called *count-to-infinity* problem. There are some extended protocols on DBF that avoid these problems using extra messaging and keeping more complex data structures than *distance matrix* of standard DBF [11].

2.4.1.1 *Performance*

The number of messages generated using DBF algorithm is bounded by the an exponential function of number of nodes in the network n, a polynomial function maximum node degree in the graph and a linear function of the number of changes in the network topology [12]. In case there is no *bouncing-effect* or *counting-to-infinity* problem, the time complexity of the multiple link failures/recoveries is $O(n)$ [11].

2.4.2 Distributed MST Algorithm

In the literature of distributed spanning tree algorithms, there has also been significant research to find distributed versions of MST algorithms. A distributed MST algorithm, which is based on the Borůvka's Algorithm (see section 2.2.2) is found by R.G Gallager et al. [13]. In this algorithm, the nodes of the sensor network generate the MST by message exchange. The algorithm assumes that the nodes now the IDs of the other nodes and the link qualities of their outgoing edges are known a priori, which could also be achieved by message exchange and using *Received Signal Strength Indication* (RSSI). All the nodes in the graph are initially in the *Sleeping* state. When they are turned on all of them exchange message with their neighbors to set up a single link. Every node makes a connection with the node that has the best possible link to it. The nodes that become connected after this first phase are called *fragments.* After this first step, the constructed fragments communicate with each other. Every fragment searches for its best outgoing edge[4]. The fragments sharing the best outgoing edge are merged through this edge and then they search for the new best outgoing edge. This procedure is applied by all fragments and in this manner fragments keep on merging with each other until there is only a single fragment left. At this point the algorithm terminates because there is no outgoing edges. This remaining single fragment is the MST itself.

2.4.2.1 *Performance*

The number of messages generated using given distributed MST algorithm is bounded by $O(n \log n + e)$, where n is the number of nodes in the network and e is the number of edges in the graph. The time complexity is proved to be $O(n \log n)$ [13]. However with some modifications, the algorithm's time complexity can be reduced to $O(n)$ [14], [15].

2.5 Spanning Tree Algorithm

This algorithm is discovered by the network engineer Radia Perlman in 1985 [16] and the *Spanning Tree Protocol* based on this algorithm was included in the IEEE Standard 802.11d. It is a distributed algorithm that is developed for the local are networks. The algorithm is computed by the bridges of the LANs to generate an acyclic spanning subset of the network which is a routing tree. For the algorithm to run, the only a priori knowledge a bridge must have is its unique ID. Originally the algorithm constructs a shortest path spanning tree in terms of hops however it can be easily extended to perform on the networks with weighted communication links.

The algorithm uses periodic broadcast messages, called "Hello Messages" to generate the spanning tree. In the original version of the algorithm, the root bridge is not unique and it is selected dynamically. Hence at first, a root bridge is selected, which is the node with the least ID. Then other bridges calculate their distance to this root and select the link that minimizes their hop distance to the root. When a particular spanning tree topology is constructed, the "Hello Message" is only sent by the root bridge and forwarded by other bridges to every one of them applying the algorithm. With this periodical messages, the bridges can keep track if there is a problem such as a link or bridge break. If one of the "Hello" Message cannot reach to some of the bridges, they generate new "Hello" Messages and with the interaction of other bridges (a bridge receiving a "Hello" Message prepares its own "Hello" Message and send it as broadcast) they realize re-routing. Another incident that triggers the re-routing is when a new LAN is introduced to the network with new bridges. In this case, again the new bridges

[4] An outgoing edge of a fragment is an edge between two nodes, one of which is already contained in the fragment and the other one is outside of that fragment.

send “Hello” Messages that initiate a re-routing. During the spanning tree is constructed, the bridges keep a link state data base per link. With this they can decide to forward or to backup the data packets they receive from some source on some link to a specific destination to prevent loops.
The *Spanning Tree Algorithm* preserves the spanning tree between the bridges even in case of link and bridge breaks. Hence, it prevents data packets entering into loops and introducing extra overhead. Another important property of the algorithm is that it is deterministic and for a given topology always the same spanning tree is constructed. The algorithm is also for wireless sensor network applicable where the sink has to have the smallest ID among all other nodes and other sensor nodes replace the bridges applying the algorithm.

2.5.1.1 *Performance*

The algorithm has a time complexity proportional to diameter of the network[5] [16]. As the algorithm is fully distributed the required bandwidth and memory per bridge is independent of the total number of bridges and links in the network.

2.6 Discussion of the Algorithms

In the sections 2.2 -2.4 , we introduced some of the important global and distributed algorithms concerning spanning trees. In this section, we will discuss these algorithms according to our problem statement and needs (see section 1.2). A candidate algorithm should be locally implementable and if it is the case, the message exchange or communication complexity must be as low as possible. In case of the topology changes, e.g., link failures, the algorithm should not generate many messages and thus cause a high power consumption.
Among the introduced MST algorithms, *Kruskal’s Algorithm* is not suitable for local implementation because in order to sort all the edges in terms of their costs, all the edge weights must be known. *Borůvka’s Algorithm* is a MST algorithm which is more appropriate for a distributed implementation and its main idea is already utilized in the literature for distributed spanning tree algorithms (see section 2.4.2). However the algorithm needs re-routing in case of link failures or topology changes. The same problem arises with *Bellman-Ford Algorithm*. Although distributed versions of *Bellman-Ford Algorithm* exists, they all need to re-route the graph in case of link breaks and or link recoveries. The *Spanning Tree Algorithm* guarantees a shortest path spanning tree even under dynamic environment and it is implemented in a distributed fashion. However, it needs periodic messages to keep track of the dynamic topology hence adds extra power consumption even if there is no change in the topology. In CWSN, the reactivity of the network is very important, hence the period of the broadcast messages to monitor the network should be high in case of re-routings. Thus, the larger amount of power must be reserved for monitoring.
Prim’s and *Dijkstra’s Algorithms* are very similar in their structure where a partial spanning tree is growing at each iteration and become the aimed spanning tree finally. In order to apply them, all the edge-weights of the links to the partial tree must be known by the nodes in the partial spanning tree. Hence they are not ready for local implementation in their original form. However, with some modifications and further processing these algorithms can be altered to be locally implementable. The important properties that could enable a local implementation are the followings:

[5] Diameter of a network: the average number of hops between every pair of nodes in the communication network.

- For both of these algorithms, a partial spanning tree is initialized, i.e., it only contains the sink initially and then in each iteration this partial spanning tree is let grow. Hence, in practice only the links which are between the nodes in the partial spanning tree and the ones closer to it matter. The quality of these links can be easily found by message exchanges.

- An other important point is that the nodes are added to the current partial spanning tree in an hierarchical order. First the parents are added and then children of them. Hence the nodes do not have to organize themselves to determine the direction to the sink.

- As the nodes are attached to the current tree in an hierarchical order, the new nodes added to the tree at an iteration step might learn their alternative links. This could help to find alternative paths in case of link failures without re-routing.

Due to these observations we will focus on these algorithms and use their basic idea of a growing partial spanning tree to develop new routing algorithms. An overview of the discussion about suitability of the algorithms are given in Table 2.

Algorithm Name	Local Implementability	Static Re-Routing Possibility
Kruskal's Algorithm	-	-
Borůvka's Algorithm	++	-
Prim's Algorithm	+	+
Bellman-Ford Algorithm	++	-
Dijkstra's Algorithm	+	+
Spanning Tree Algorithm	++	-

Table 2: Comparison of the algorithms

3. New Routing Algorithms

3.1 Pairing Algorithm: Localization of Prim's and Dijkstra's Algorithms

We developed the so-called *Pairing Algorithm* to localize the *Prim's* and *Dijkstra's Algorithm*. As we have seen in section 2.6 , these two algorithms are in their original forms not suitable to be implemented locally. However with *Pairing Algorithm*, although the resulting trees might be suboptimal, a localization of these algorithms is possible.
The algorithm is based on the idea of the growing partial spanning tree that is utilized by *Prim's* and *Dijksta's Algorithms* (see sections 2.2.1 and 2.3.1). The given communication network is a graph $G(V,E)$ with nodes V and links E. There is a sink node among V and an edge $\{i,j\}$ has the weight w_{ij}. Initially, the partial spanning tree only contains the sink. During the execution of the algorithm new links are added to the partial spanning tree and at the end a spanning tree is constructed. At each iteration of the algorithm, the basic operations that are executed are as follows:

1. Each node in the current partial spanning tree selects a *favorite child* node (if any) from the nodes that are not added to the current tree yet.

2. Each node of the graph that is not a member of the current tree selects a *favorite parent* (if any) among the nodes in the current tree.

3. The nodes in the current tree which are *favorite parent* of their *favorite child* set up a link with these nodes.

An algorithm iteration at an intermediate step is illustrated in Figure 1. The algorithm is locally, i.e., in the node level, implementable. In order to setup a link, the nodes in the current partial spanning tree do not need global information. Information from their neighborhood would be enough to decide which node is their *favorite child* and the same holds for other nodes in the graph and their *favorite parent*. The necessary synchronization could be realized by a flood message from the sink. A possible protocol realizing *Pairing Algorithm* at node level using messages and timers is given in section 3.3 .

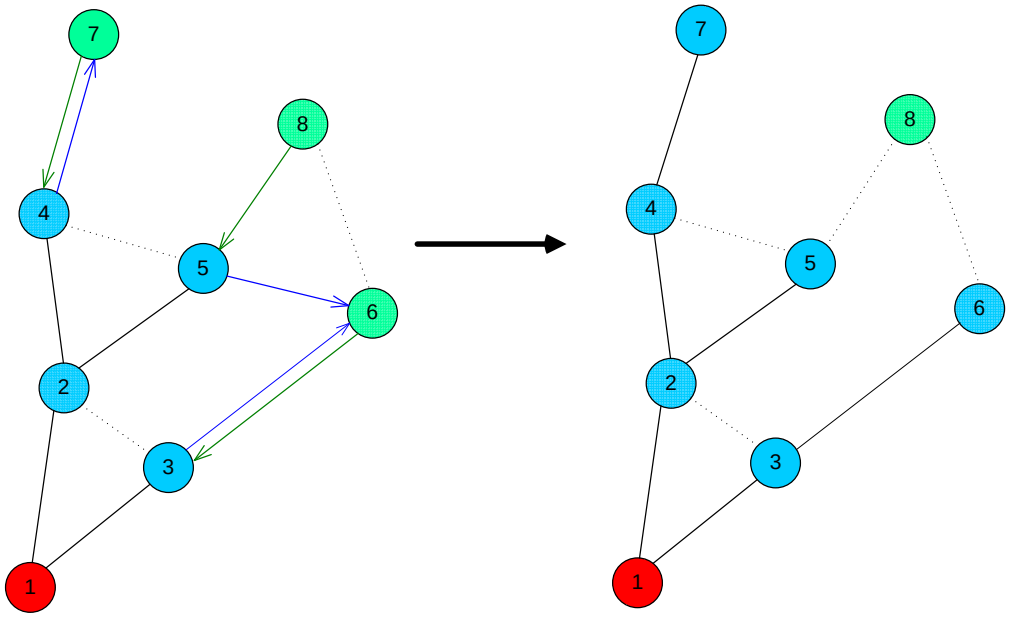


Figure 1: An illustrative example of a link setup. The sink node is in red and the blue nodes with numbers 2-5 are other nodes existing in the current tree. The solid black lines are the links of the current tree and the dotted black lines are the possible links. The nodes 7,8 and 9 are potential child nodes, which are not added to the tree yet (on the left). The node 7 selects node 4 as its favorite parent (green arrow), whereas the node 4 selects node 7 as favorite child (blue arrow). The same happens with node 6 and 3 as well. Node 5 selects node 6 as its favorite child and node 8 selects node 5 as its favorite parent. After links are constructed with one *Pairing Algorithm* iteration, we obtain the partial spanning tree on the right. Node 6 and 7 are connected to the tree whereas the node 8 could not be added to the tree because it was not selected by its favorite parent.

An important property of the *Pairing Algorithm* is that it is generic in terms of the metrics that it optimizes. The selection of the *favorite parents* and *favorite children* depends solely of the metrics. For MST optimization, a parent node selects the node that has the best link to it as its *favorite child*, the same holds for the child node for its *favorite parent*. For SPT optimization, the parents select the node with the best possible link to them as *favorite child* whereas the children select the parent that would minimize their total distance to the sink as their *favorite parent*. For optimization metrics other than SPT and MST, various parameters such as number of hops, number of node branches can be used for the selection of the *favorite parents* and *favorite children*.

If for any optimization metrics, there are more than one possible favorite child or parent candidate, the one with the smaller ID is selected. In this way, the uniqueness of the favorite children and parents is guaranteed. The algorithm is also deterministic and hence reproducible.

3.1.1 *Algorithm 5: Pairing Algorithm*

Inputs:

E : Set of edges in the graph
W : Set of edge-weights
V : Set of vertices
Cost(*i*,*j*) : Cost function according to some metric (Prim, Dijkstra, etc.) that outputs the cost of

the connection between parent node *i* and child node *j*.

Outputs:

E_T : Edges of the current tree

Temporary Variables:

T : Set of the vertices added to the routing tree

T' : Set of the vertices not added to the routing tree yet

FP_j : Favorite Parent of node *j*. It is comprised of 2 entries, favorite parent ID *i* and the cost of the link between the nodes *j* and *i*, e.g., $FP_j(1)=i$, $FP_j(2)=\text{Cost}(i,j)$.

FC_i : Favorite Child of node *i*. It is comprised of 2 entries, favorite child ID *j* and the cost of the link between the nodes *i* and *j*, e.g., $FC_i(1)=j$, $FC_i(2)=\text{Cost}(i,j)$

Initialization:

$T := \{Sink\}$

$T' := V / \{Sink\}$

$E_T := \{\}$

$FP_j := \{\text{Null, INF}\} \ \forall j \in V$

$FC_i := \{\text{Null, INF}\} \ \forall i \in V$

Main Algorithm:

While T != V

  For $\forall i \in T$

    For $\forall j \in T'$

      If (Cost(i,j) < $FC_j(2)$)

        $FC_i(2) := \text{Cost}(i,j)$

        $FC_i(1) := j$

      End

      If (Cost(i,j) < $FP_j(2)$)

        $FP_j(2) := \text{Cost}(i,j)$

        $FP_j(1) := i$

      End

    End

  End

  For $\forall i \in \boldsymbol{T}$

    $j := FC_i(1)$

    If ($i=FP_j(1)$)

      $T := T \cup \{i\}$

      $T' := T' \setminus \{i\}$

      $E_T := E_T \cup \{\{i, j\}\}$

    End

  End

  $FP_j := \{\text{Null, INF}\} \ \forall j \in V$

  $FC_i := \{\text{Null, INF}\} \ \forall i \in V$

End

3.1.2 Block Diagram of the Pairing Algorithm

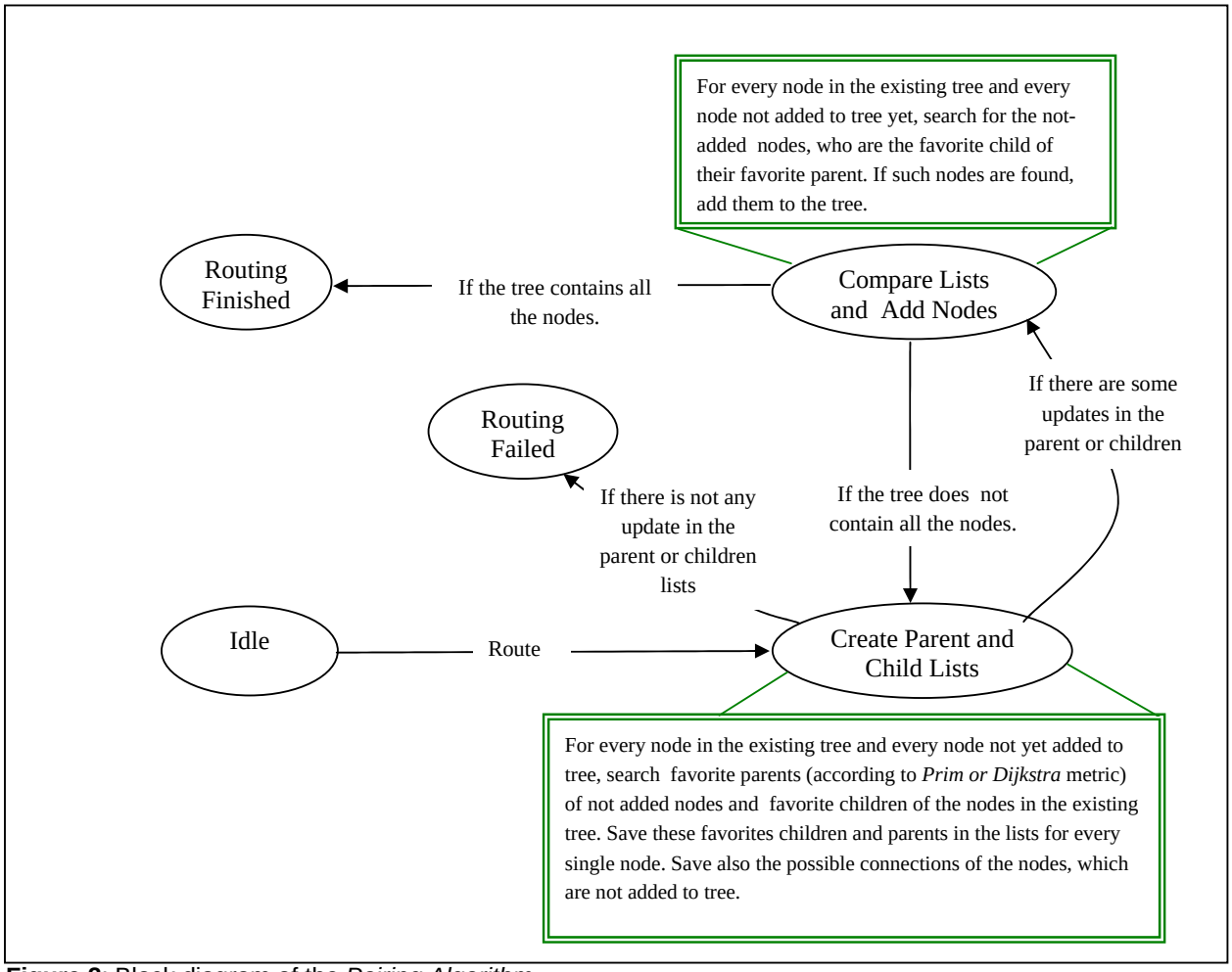


Figure 2: Block diagram of the *Pairing Algorithm*

3.1.3 Complexity[6]

The *Pairing Algorithm* has $O(n)$ iterations, where n is the number of nodes in the network. At each iteration, every node considers only its neighborhood. Hence, each iteration takes $O(d)$ time, where d is the *node degree*. Hence the overall time complexity of the algorithm becomes $O(n \cdot d)$.

3.1.4 Convergence Issues

Definitions

$G(V,E)$: A Graph with vertices V and edges E.
$T(V_T,E_T)$: Current tree with vertices V_T and edges E_T.
Cost(i,j) : Cost function according to some metric (Prim, Dijkstra, etc.) that outputs the cost of the connection between parent node i and child node j.
i , j : Node IDs

[6] Complexity refers to time complexity for the off-line computation of the spanning tree with the *Pairing Algorithm*.

Theorem
For a single-source, undirected graph $G(V,E)$ with the set of nodes V, edges E, the *Pairing Algorithm* always converges and finds a routing tree provided the graph $G(V,E)$ is *connected*.

Proof
Proving that at least one new link is added to the current tree at each algorithm iteration is enough to guarantee that the algorithm converges in case the graph is connected. We consider the cost minimizing link $\{i^*,j^*\}$ at some iteration step of the algorithm, i.e., V_T being the set of the vertices in the existing tree and $V_T' = V \setminus V_T$ the set of the vertices that are not added to the existing tree, $\{i^*.j^*\} = \underset{\forall i \in V_T, \forall j \in V_T'}{\arg\min} \left(\text{Cost}(i, j)\right)$. Such a pair can be definitely found because the graph is connected. Nevertheless, the cost minimizing pairs might not be unique. In this case, as the algorithm always chooses the link with minimum node ID and the node IDs are unique, a unique cost minimizing pair $\{i^*,j^*\}$ can be found. The fact that the algorithm allows parallel link additions to the current tree, does not violate this situation as at least the best possible link for that current tree is definitely added. Hence we can conclude that for connected graphs the current tree grows at each step until the last node is added, which guarantees convergence of the *Pairing Algorithm*.

3.2 N-SafeLinks Algorithm

3.2.1 Main Idea: *N-SafeLinks*

The *N-SafeLinks* algorithm is based on the assumption that every node in a graph has N possible links to other nodes to reach the sink node and at most N-1 of these link can break at the same time. For this algorithm, we utilize again the growing partial spanning tree concept as we did for the *Pairing Algorithm* (see section 2.2.1). In the *N-SafeLinks* algorithm, a node is only eligible to be added to the current tree if it has at least N possible links to the current tree. The ineligible nodes are ignored until they have N possible links to the current tree. The *N-SafeLinks Algorithm* is generic like the *Pairing Algorithm* as the links to be added to the current tree can be selected with any arbitrary metric. However, there are two different variants for selecting the best possible link:

1. Only the best link of the eligible node is considered.

2. The average of the best possible N links of a node is considered.

We carried out all the tests with the second variant because for a varying environment where alternative links are frequently needed, the overall performance is more important than the best case. In the next two sections we introduce two versions of the *N-SafeLinks Algorithm* and examine them in detail.

3.2.2 N-SafeLinks Algorithm without Pairing

In its simple version, the *N-SafeLinks Algorithm* follows the given steps at each iteration in the intermediate phase

1. The list of the eligible nodes, i.e., the nodes having at least *N* possible links to the current tree is updated

2. The best possible link[7] from the current tree to the eligible nodes is found and added to the current partial spanning tree

There are two other phases of the algorithm that differ from the intermediate phase. The first one is termed as *initial phase* and is carried out at the beginning of the algorithm. In this phase, an initial tree is constructed with the sink and *N* nodes, where all of these nodes have possible connections to each other. The latter phase is the *final phase*, where every node lists their best *N* possible links and the belonging parents. Every node saves its list on a table that is termed *Parent Table*. These tables are used in case of link breaks to switch to alternative parents in the normal operation. The algorithm for this simple version of the *N-SafeLinks Algorithm* is given in section 3.2.2.1 . The simple version of the *N-SafeLinks Algorithm* has the problem that it cannot be implemented locally. The reason is that for each algorithm iteration the weights of the all outgoing links from the current tree have to be known.
At this point, we utilized the fact that both *N-SafeLinks* and *Pairing Algorithm* use the same concept of the growing partial spanning tree. So we merged the *N-SafeLinks* and *Pairing Algorithms*, which is introduced in section 3.2.3

3.2.2.1 *Algorithm 6: N-SafeLinks Algorithm without Pairing (for N=2)*

Inputs:
E : Set of edges in the graph
W : Set of edge-weights
V : Set of vertices
M : Number of vertices
Cost(i,j) : Cost function according some metric (Prim, Dijkstra, etc.) that outputs the cost of the connection between node i and j.

Outputs:
E_T : Edges of the current tree
ParentTable : It is a matrix $ParentTable^{\{Mx2\}}$. Every node ID refers to a row number. First and second entries in a row denote the main and alternative parents, respectively.

Temporary Variables:
T : Set of the vertices added to the routing tree
T' : Set of the vertices not added to the routing tree yet
$Parent(i)$: Parent node or predecessor of node i
PPL_i : Possible links list of the node i. It is comprised of concatenated node and cost pairs, e.g., PPL_i=({$Node_m$,Cost(i,m)},{$Node_n$,Cost(i,n)}, …)

Initialization:
$T := \{Sink\}$
$T' := V / \{Sink\}$

[7] Best possible link is not the link with the minimum edge weight but it is the optimum link for the given optimization metric, e.g., MST,SPT etc. The best link might be selected as the single best link of the node (variant 1) or the average best link of the node (variant 2).

E_T:= {}
FirstNode := {}
SecondNode := {}
BestLink :={}
BestCost := INF
$PLL_i := \{\} \ \forall i \in V$

Main Algorithm:

Part – 1 (Initial Phase: Adding first node):

For $\forall i \in T$
  If (Cost(*Sink*,*i*) < *BestCost*)
    FirstNode:=*i*
    BestCost := Cost(*Sink*,*i*)
  End
End
BestCost := INF
PLL_i:= ({*Sink*, Cost(*Sink*,*i*)})
$T := T \cup \{i\}, T' := T'/\{i\}$
$E_T := E_T \cup \{\{Sink, i\}\}$

Part – 2 (Initial Phase: Adding second node):

For $\forall i \in T$
  If (Cost(*Sink*,*i*) < *BestCost* **And** $\{i, FirstNode\} \in E$)
    SecondNode:=*i*
    BestCost := Cost(*Sink*,*i*)
  End
End
BestCost := INF
PLL_i:= ({*Sink*, Cost(*Sink*,*i*)})
$T := T \cup \{i\}, T' := T'/\{i\}$
$E_T := E_T \cup \{\{Sink, i\}\}$

Part – 3 (Intermediate Phase: Adding remaining nodes):

While T != V
  For $\forall i \in T$
    For $\forall j \in T'$
      If $\{i, j\} \in E$
        $PLL_i := PLL_i \cup \{\{j, \text{Cost}(i, j)\}\}$
      End
      If (Cost(*i*,*j*) < *BestCost* **And** $|PLL_i| \geq N$)
        BestCost := Cost(*i*,*j*)
        BestLink := {*i*,*j*}
      End
    End
  End
  $T := T \cup \{BestLink(2)\}$

$T' := T' \setminus \{BestLink(2)\}$

$E_T := E_T \cup \{BestLink\}$

$Parent(BestLink(2)) := BestLink(1)$

$BestCost := INF$

$BestLink := \{\}$

End

Part – 4 (Final Phase: Generate the parent tables):

For $\forall i \in \boldsymbol{T} \setminus \{Sink, FirstNode, SecondNode\}$

$PLL_i :=$ **SortInAscendingOrder (**PLL_i**)**

If $PLL_i(1,1) = Parent(i)$

$ParentTable(i) := \{Parent(i), PLL_i(2,1)\}$

Else

$ParentTable(i) := \{Parent(i), PLL_i(1,1)\}$

End

End

Sub-Routines:

SortInAscendingOrder(ListOfPairs V**) :** It is a sub-routine that sorts the pairs in the list of pairs according to second entry in ascending order, e.g., given V={{1,2},{3,6},{5,1}}, **SortInAscendingOrder**(V) →{{5,1},{1,2},{3,6}}

3.2.3 N-SafeLinks-Pairing Algorithm

The *N-SafeLinks-Pairing Algorithm* uses the same neighborhood selection procedure of the *Pairing Algorithm*. It applies at the same time the eligibility principal of *N-SafeLinks Algorithm*, i.e., the nodes with less than *N* possible links to the current tree are not eligible to be added. The basic steps of the algorithm at some intermediate iteration are as follows

1. The list of the eligible nodes, i.e., the nodes having at least *N* possible links to the current tree is updated (from *N-SafeLinks Algorithm*).

2. Each node in the current partial spanning tree selects a *favorite child* node (if any) from the eligible nodes that are not added to the current tree yet (from *Pairing Algorithm*).

3. Each node existing in the graph but not a member of the current tree selects a *favorite parent* (if any) among the nodes in the current tree (from *Pairing Algorithm*).

4. The nodes in the current tree, which are *favorite parent* of their *favorite child*, set up a link with these nodes (from *Pairing Algorithm*).

An illustrative example of an algorithm step is shown in Figure 3. As only this version of the *N-SafeLinks Algorithm* is of concern for a local implementation, in the rest of the report, the term *N-SafeLinks Algorithm* will denote the *N-SafeLinks-Pairing Algorithm*. A possible local implementation of the *N-SafeLinks Algorithm* is given in section 3.3 .

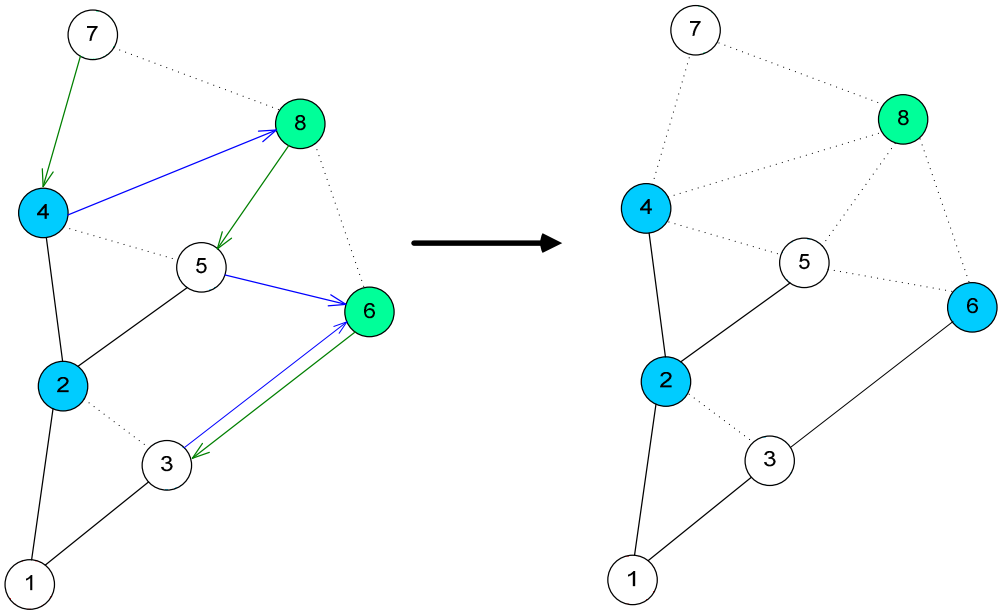


Figure 3: An example of an iteration using the *2-SafeLinks-Pairing Algorithm* for the same current tree from Figure 1. The sink node is in red and the blue nodes with numbers 2-5 are other nodes existing in the current tree. The solid-black lines are the links of the current tree and dotted lines are possible links. The nodes 7,8 and 9 are potential child nodes, which are not added to tree yet (on the left). The node 7 selects node 4 as its favorite parent (green arrow), however node 7 is not *eligible* as it has only one link to the current tree, hence the node 4 selects another node as favorite child (node 8 with blue line) . The node 6 is eligible as it has 2 possible connections to 3 and 5. Node 3 and node 5 selects node 6 as its favorite child and node 8 selects node 5 as its favorite parent. After links are constructed with one algorithm iteration, we obtain the partial spanning tree on the right. Only 6 is connected to the current tree whereas the nodes 7 and 8 could not be added to the tree because they were not selected by their favorite parents.

3.2.3.1 *Algorithm 7: N-Safe-Links Algorithm with Pairing (N=2)*

Inputs:

E : Set of edges in the graph
W : Set of edge-weights
V : Set of vertices
M : Number of vertices
Cost(i,j) : Cost function that outputs the cost of the connection between node i and j.

Outputs:

E_T : Edges of the current tree
ParentTable : It is a matrix $ParentTable^{\{Mx2\}}$. Every node ID refers to a row number. First and second entries in a row denote the main and alternative parents, respectively.

Temporary Variables:

T : Set of the vertices added to the routing tree
T' : Set of the vertices not added to the routing tree yet
Parent(i) : Parent node or predecessor of node i
FP_j : Favorite Parent of node j. It is comprised of 2 entries, favorite parent ID i and the cost of the link between the nodes j and i, e.g., $FP_j(1)=i$, $FP_j(2)=\text{Cost}(i,j)$.
FC_i : Favorite Child of node i. It is comprised of 2 entries, favorite child ID j and the cost of the link between the nodes i and j, e.g., $FC_i(1)=j$, $FC_i(2)=\text{Cost}(i,j)$
PPL_i : Possible links list of the node i. It is comprised of concatenated node and cost pairs, e.g., $PPL_i=(\{Node_m,\text{Cost}(i,m)\},\{Node_n,\text{Cost}(i,n)\},\ldots)$

Initialization:

$T := \{Sink\}$
$T' := V / \{Sink\}$
$E_T := \{\}$
FirstNode := {}
SecondNode := {}
BestCost := INF
FP_j := {Null, INF} $\forall j \in V$
FC_i := {Null, INF} $\forall i \in V$
$PLL_i := \{\}$ $\forall i \in V$

Main Algorithm:

Part – 1 (Initial Phase: Adding first node):

For $\forall i \in T$
  If (Cost(*Sink*,*i*) < *BestCost*)
    FirstNode:=*i*
    BestCost := Cost(*Sink*,*i*)
  End
End
BestCost := INF
PLL_i:= ({*Sink*, Cost(*Sink*,*i*)})
$T := T \cup \{i\}, T' := T'/\{i\}$
$E_T := E_T \cup \{\{Sink, i\}\}$

Part – 2 (Initial Phase: Adding second node):

For $\forall i \in T$
  If (Cost(*Sink*,*i*) < *BestCost* **And** $\{i, FirstNode\} \in E$)
    SecondNode:=*i*
    BestCost := Cost(*Sink*,*i*)
  End
End
BestCost := INF
PLL_i:= ({*Sink*, Cost(*Sink*,*i*)})
$T := T \cup \{i\}, T' := T'/\{i\}$
$E_T := E_T \cup \{\{Sink, i\}\}$
ParenTable(*FirstNode*) := {Sink,SecondNode}
ParenTable(*SecondNode*) := {Sink, FirstNode}

Part – 3 (Intermediate Phase: Adding remaining nodes):

While T != V
  For $\forall i \in T$
    For $\forall j \in T'$
      If $\{i, j\} \in E$
        $PLL_i := PLL_i \cup \{\{j, \text{Cost}(i, j)\}\}$
      End
      If (Cost(i,j) < $FC_i(2)$)
        $FC_i(2)$:= Cost(i,j)
        $FC_i(1) := j$
      End
      If (Cost(i,j) < $FP_j(2)$)
        $FP_j(2)$:= Cost(i,j)
        $FP_j(1) := i$
      End
    End
  End

  For $\forall i \in \boldsymbol{T}$
    $j := FC_i(1)$
    If (i=$FP_j(1)$ **And** $|PLL_i| \geq N$)
      $T := T \cup \{i\}$
      $T' := T \setminus \{i\}$
      $E_T := E_T \cup \{\{i, j\}\}$
      $Parent(j) := i$
    End
  End
  FP_j := {Null, INF} $\forall j \in V$
  FC_i := {Null, INF} $\forall i \in V$
End

Part – 4 (Final Phase: Generate the parent tables):

For $\forall i \in \boldsymbol{T} \setminus \{Sink, FirstNode, SecondNode\}$
  PLL_i := **SortInAscendingOrder (**PLL_i**)**
  If $PLL_i(1,1) = Parent(i)$
    $ParentTable(i)$:= {$Parent(i)$,$PLL_i(2,1)$}
  Else
    $ParentTable(i)$:= {$Parent(i)$,$PLL_i(1,1)$}
  End
End

Sub-Routines:

SortInAscendingOrder(ListOfPairs V**) :** It is a sub-routine that sorts the pairs in the list of pairs according to second entry in ascending order , e.g., given V={{1,2},{3,6},{5,1}}, **SortInAscendingOrder**(V) →{{5,1},{1,2},{3,6}}.

3.2.4 Block Diagram of the *N-Safe-Links Algorithm*

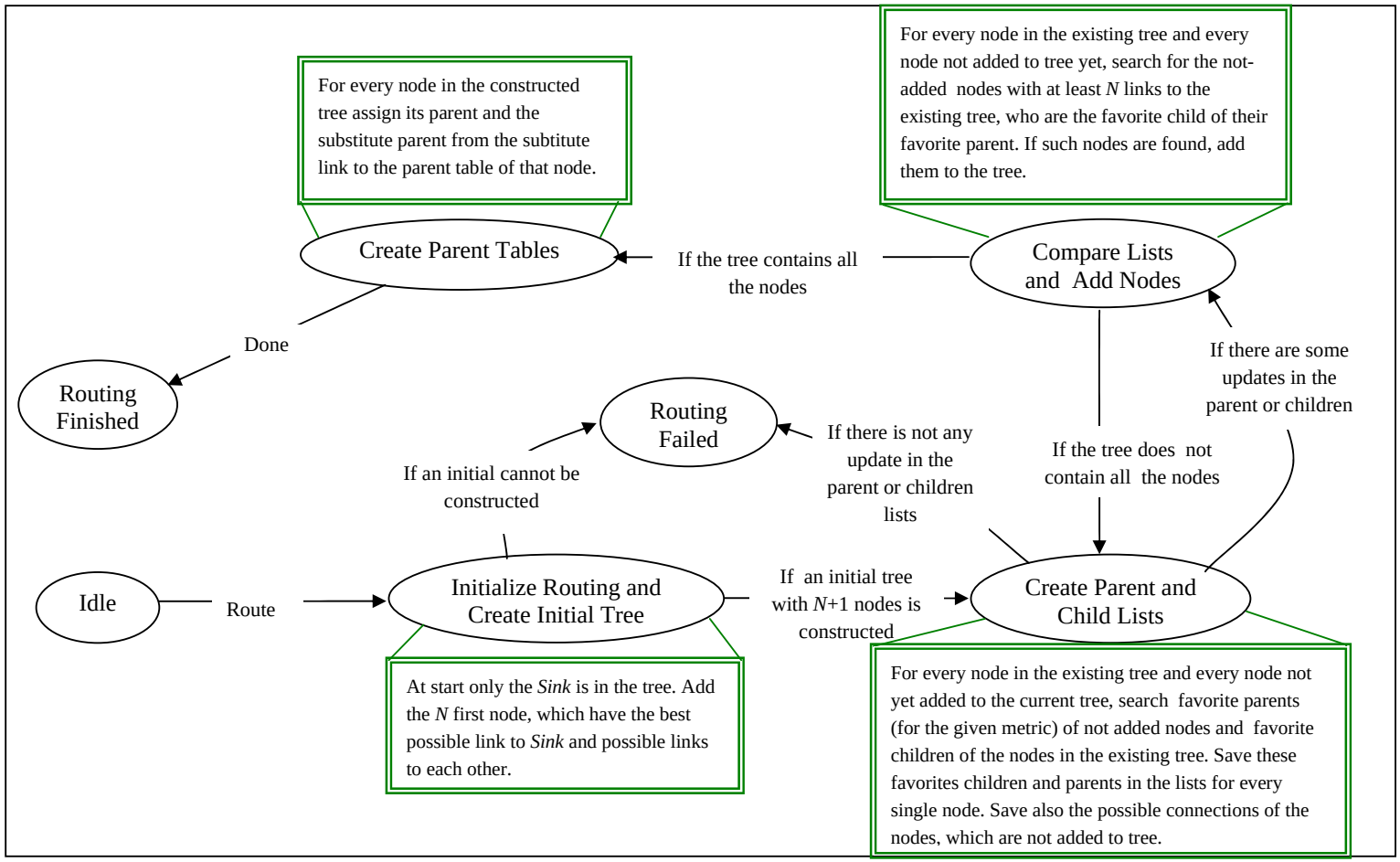


Figure 4: Block Diagram of the *N-Safe-Links Algorithm*

3.2.5 Complexity[9]

The *N-SafeLinks Algorithm* has $O(n)$ iterations, where n is the number of nodes in the network. Each iteration to construct the spanning tree takes $O(d)$ time, where d is the *node degree* and when the tree is constructed, for every node to create its *parent table* needs $O(m)$ time. Hence the overall time complexity of the algorithm becomes $O(n \cdot d) + O(m \cdot d)$.

3.2.6 Convergence Issues

3.2.6.1 *Definitions*

$G(V,E)$: A Graph with vertices V and edges E.
$T(V_T,E_T)$: Current tree with vertices V_T and edges E_T.
j : Node IDs

3.2.6.2 *Convergence Condition*

For a single-source, undirected graph $G(V,E)$ with the set of nodes V, edges E, the following condition must hold to guarantee the convergence of the *N-SafeLinks Algorithm*:
At any algorithm step s, there must be at least one node $j \in V \setminus V_T$, that has at least N possible links to the nodes V_T of the current tree T. If for each algorithm step such a node can be found the algorithm adds this node to the current tree and the current tree grows at every iteration until the last node $j \in V$ is added. Otherwise, at some iteration, the algorithm enters into a dead-lock and no new node can be attached, hence no spanning tree can be constructed.

3.2.7 Proof of Loop-Freeness

3.2.7.1 *Definitions*

T_i : Set of nodes in the current tree just before node i is added
L_i : Set of the nodes appearing in the parent table of node i
S_i : Set of nodes appearing in the subtree[10] of node i
N^s : Set of nodes added to the tree after the step s
i,j,k : Node indices

3.2.7.2 *Lemmas*

Lemma 1: The set L_i containing the nodes appearing in the parent table of node i is a subset of the nodes in the current tree just before node i is added, meaning

$$T_i \supset L_i \ .$$

[9] Complexity refers to time complexity of the off-line computation of the spanning tree with the *N-SafeLinks Algorithm*.

[10] The subtree of a node i, is the partial tree with the root node i. The subtree of node i is regarded only then when the routing is finalized and the complete tree is generated.

This is true because the nodes in L_i are selected among the nodes from the current tree.

Lemma 2: If a node *i* appears in the current tree just before another node *j* is added and node *j* is in the current tree just before the node *k* is added then the node *i* must be in the current tree just before node *k* is added. This can be stated rigorously as

$$\forall i \in T_j, j \in T_k \Rightarrow i \in T_k \quad .$$

This is due to the fact that the algorithm lets the tree grow at every step. A node added to the current tree at any step remains permanently there.

Lemma 3: Any node *i* appearing in the current tree of node *j* cannot appear in this node's subtree at the same time, i.e.,

$$\forall i \in T_j \Rightarrow i \notin S_j \quad .$$

This is obvious as a node can be added to the tree only once. A node *i* which is added to the tree before the node *j* cannot be in the set of nodes that are added after *j*.

3.2.7.3 *Theorem : Loop Freeness of N-SafeLinks Algorithm*

For a tree constructed by the *N-SafeLinks Algorithm*, irrespective of the link, i.e., primary link or alternative link, that a node uses, the routing graph remains as a tree, i.e., no loops can occur.
The loop-freeness (acyclic) property can be stated formally as follows: At algorithm step *s*, for $\forall i \in N^s$, $\forall k \in L_j$ and $\forall j \in L_i$ the following always holds $k \notin S_i$. The proof is as follows

From *Lemma 1*, we know that if the node *k* is in the parent table of node *j*, then it must exist also in the current tree just before node *j* is added and because of the same reason the node *j* must be in the current tree just before node *i* is added, i.e.,

$$\forall k \in L_j \Rightarrow k \in T_j \text{ and } \forall j \in L_i \Rightarrow j \in T_i \quad .$$

From *Lemma 2*, we conclude if the node *k* is in the current tree of node *j* and node *j* is in the current tree of node *i*, then node *k* must be in the current tree of node *i*. This can be stated mathematically as follows

$$\forall k \in T_j, j \in T_i \Rightarrow k \in T_i \quad .$$

At last, directly from *Lemma 3*, we deduce that the node *k* cannot be in the subtree of the node *i* as it is present in the current tree before node *k* is added.

$$k \in T_i \Rightarrow k \notin S_i \quad .$$

Thus we have proven that for any selection of the links of the nodes (link to main or alternative parent), the resulting graph does not have cycles and the tree structure is preserved.

3.3 An Initial Routing Procedure using 2-SafeLinks Algorithm

In this section, we describe a possible procedure that can be utilized to implement the *2-SafeLinks Algorithm* in the node level. This routing procedure implements the algorithm in the initial phase of the WSN application. After the routing tree is generated and the nodes acquire their *parent tables* the normal operation starts. The procedure is based on message exchange between the nodes and timers to trigger necessary events. We will first explain the necessary messages and the timing blocks of the routing procedure. Afterwards the states of the implemented algorithm is given. At last the state diagram and timing scheme will be presented.

3.3.1 Messages needed for the execution of 2-SafeLinks Algorithm

- **M_1 :** This message is sent from the nodes in the existing tree to search new children.
  Payload: {*Message Type, Source Address*}

- **M_2:** This message is generated by the nodes, which are not added to the tree, yet. It indicates the favorite parent.
  Payload: {*Message Type, Source Address, Favorite Parent Address, Link Quality*}

- **M_3:** This message is generated by the nodes in the existing tree to inform the nodes searching a parent, which is selected by this parent.
  Payload: {*Message Type, Source Address, Favorite Child Address*}

- **M_4:** This message is an acknowledge message generated by the node who found his favorite parent and agreed to setup a connection.
  Payload: {*Message Type, Source Address, Favorite Child Address*}

- **M_5:** These messages are the unicast messages from children to parents until Sink to inform the Sink about the newly added child.
  Payload: {*Message Type, Source Address, Parent Address*}

- **M_6** (reserved): Message that is first generated by the sink, sent from parent to children, declaring that the routing is finished and the network is ready for operation.
  Payload: {*Message Type, Source Address, Parent Address*}

3.3.2 Special messages needed for the initialization

- **M_1^* :** Message from the sink indicating that *FirstNode* is searched after sink to be added to the tree.
  Payload: {*Message Type, Source Address*}

- **M_2^* :** Message from the nodes getting the M_1^* from the sink. It is a message for other neighbors.
  Payload: {*Message Type, Source Address*}

- **M_3^* :** Message from the nodes competing to be part of the initial tree to the Sink. It contains the link quality to sink and the list of neighbors, to which a link above a certain threshold exists.
  Payload: {*Message Type, Source Address, Link Quality,* {*neighbor1,neighbor2,…*}}

- **M_{SINK}^* :** Message from the Sink to a certain node to indicate that this node is selected as *FirstNode* and its substitute parent is *SecondNode*.
  Payload: {*Message Type, Sink Address, FirstNode Address, SecondNode Address*}

- **M_{SINK}^{**} :** Message from the Sink to a certain node to indicate that this node is selected as *SecondNode* and its substitute parent is *FirstNode*.
  Payload: {*Message Type, Sink Address, SecondNode Address, FirstNode Address*}

3.3.3 Timing of Local Implementation

The communication during the routing starts with a flood from Sink. With this flood the nodes set their clock to the global clock of the sink. Afterwards during a time-block, the initial tree is constructed.

3.3.3.1 *Initialization time-block*

The Initialization Time-Block starts with the message M_1^* from the sink. The nodes getting this message, generate the message M_2^* and send it while listening also to the medium (Phase1). This phase finishes when First-Phase-Timeout occurs. In the next phase of Initialization Block, the nodes, which got M_1^* and M_2^* send a status message about their neighbors and link quality level to sink (Phase2). After processing these messages the sink sends back two different messages M_{SINK}^* and M_{SINK}^{**}, which tells to the receiving nodes who are the first and second nodes in the initial tree (Phase3). In an optional fourth phase these nodes might send back acknowledge messages. The Initialization Time-Block stops with Communication-Block-Timeout, which also make Communication-Time-Block and hence the *2-SafeLinks Algorithm* to start.

3.3.3.2 *Communication time-block*

Communication takes place in blocks, where each time, a group of nodes are added to the existing tree (one algorithm iteration). These blocks start with a sub-block, where newly added nodes send M_1 (Block 1), in the second sub-block the potential children who chose a favorite parent generate M_2 (Block 2),. In the third sub-block, the parents, who have chosen a best child, that demands a communication with them, send messages M_3 (Block 3). In the next block, the children who got M_3, sends an acknowledge to the parent and become a member of the tree (Block 4). Afterwards parents, who acquired a new child send this child's address to their parents. These addresses are transmitted from children to parents all the way until the sink (Block 5),. In the reserved last block, if the routing is over, sink sends a message M_6 that is transmitted from the parents to the children. This message indicates that the routing is accomplished (Block 6). During the routing process, the communication block repeats whenever a Communication-Block-Timeout comes.

3.3.3.3 *Important notes about timing*

There are two types of timeout:

1- Initial Phase Timeout: This timeout is used in the Initialization Time-Block. It means the exchange of the messages M_2^* should be finished. After this timeout, the nodes, who are competing to become the first and second of initial tree send M_3^* to the sink.

2- Communication Block Timeout: This timeout is used to indicate that one iteration of *2-SafeLinks Algorithm* is finished. After this timeout either a new block starts or a normal operation should start if the algorithm is terminated and the tree is constructed.

The timeouts are triggered locally at every node. The sub-block lengths should be defined appropriately long to allow all nodes to find time to send their messages but in the same time, not too long, which might cause the clock drifts impossible to be tracked the whole routing process long.

3.3.4 Description of States

Idle: The initial state of the nodes after installation or a start of the network. The nodes leave this state, if they receive either the message M_1^* from the sink or M_1 from a node in the tree.

Woken Up for Initial Tree: The nodes, after receiving M_1^* from the sink transit to this state. It is the first state of the process, where the initial tree comprised of 3 nodes including sink is constructed. The nodes entering this state create a message (M_2^*) and send it to their neighborhood. Meanwhile, they listen to the medium and try to receive M_2^* from other nodes in their neighborhoods. If they receive any M_2^* from an external node in the neighborhood, they change their state to the *Neighbors Found*.

Neighbors Found: In this state, the nodes try to catch other M_2^* messages from the neighborhood. The owners of message, that come over a certain threshold are saved to a buffer to be used later. No message is generated by the node during this state.

Inform Sink about Status: The transition to this stage is triggered by a special timeout, namely Initial-Phase-Timeout. The nodes in this state generate a message M_3^*, which contains their neighborhood nodes (with a certain link quality above a predefined threshold) and the link quality to the sink, which they measure from M_1^*. A node exits from this state in the case of getting the message M_{SINK}^* or M_{SINK}^{**}. If no message is received, the node exits this state when the Communication-Block-Timeout occurs.

Become FirstNode: This state can be transited, if a node get a message M_{SINK}^* while it was in the *Inform Sink about Status* state. This message from the sink means that the sink made a processing among M_3^* messages and selected this node as the first node to be added to tree, because it had the best possible link to the sink and a link above a threshold to the *SecondNode* of the initial tree. When the node gets a Communication-Block-Timeout, it switches over the *Connected* state.

Become SecondNode: This state is very similar to the state Become FirstNode and it can be entered, if a node gets a message M_{SINK}^{**} while it was in the *Inform Sink about Status* state. When the node in this state gets a Communication-Block-Timeout, it switches over the *Connected* state.

One Possible Connection: The nodes, who were in the neighborhood of sink but node selected neither as first nor as second node, transits to this state, when they got the first Communication Block Timeout, which indicates that the initialization block is over and the Local Modified Prim algorithm starts. Another option to enter this state is for the nodes, who gets an M_1 message for the first time. In One Possible Connection state, nodes do not generate messages but they wake up in the first sub-block of Communication-Time-Block and they try to catch other M_1 messages

Qualified To Connect: A node enters to this state if it was in the *One Possible Connection* state and received a new M_1 message. The nodes in this state search for parents to set up connection, hence they send M_2 messages in the second sub-block of the Communication-Time-Block. M_2 contains the favorite parent and the link quality to it. If a parent gets this message and decides that this node is its favorite node, he sends a message M_3 declaring its wish for connection to this node in the 3rd sub-block of Communication-Time-Block. With this arriving message M_3, a state transition occurs and the node enters the state *Connected*.

Connected: One node can enter to this state from initialization phase if it is the first or second node of the tree. In this case the transition occurs after the first Communication-Time-Block. The other option to enter this state is the case when the node was in *Qualified to Connect* state and receives the message M_3 from its favorite parent. From this state on, the node is a member of the tree. In this state, the first thing the node does is to send an acknowledge to its parent (M_4) in the 4^{th} sub-block. After that it waits for the first sub-block to send M_1 for the nodes not yet added to tree and searching for a parent. For the transition from this state, a node waits for the message M_2 or M_6. With the first M_2 message that arrives, it enters to *Search Nodes* state. In case of M_6 the *Normal Operation* sate is entered.

Search Nodes: In this state, the node is in the tree and it searches for new children. In sub-block 2, if it receives M_2 messages from the nodes wanting to connect to it, these messages are processed and if the node's favorite child has chosen this node as its favorite parent, a connection should be established. Hence, the node in this present state, sends the message M_3 informing the child node during the sub-block 3, a connection is established and this new node becomes a new member of the tree. The node exits this state either if it gets an acknowledgement (M_4) from the child or an algorithm-termination message (M_6) is acquired.

Child Found: A node enters into this state, when it was in *Search Nodes* state and it catches the message M_4. The mission for the node in this state is to send its parent the information that a new node is added to the tree and the address of this new node (M_5). The message is sent during the 5^{th} sub-block. In this sub-block the node should also listen to its other children (if any) that might also be sending M_5 and if one of this messages is caught, the node sends it to its parent. . In this way, this information should go until the sink. The node exits from this state, if either a Communication-Block-Timeout occurs or an algorithm-termination message (M_6) is received.

Normal Operation: It is the final state, to which all nodes enter in the same interval (within the 6^{th} sub-block of the last Communication Block). This state is entered when a message M_6 is received from the parent, while the node is in *Search Nodes*, *Child Found* or *Connected* state. In this state, the node sends M_6 to its children, which will put them to this *Normal Operation* state. After delivering the M_6 to its children, the node is done with routing and it is ready for the normal operation

3.3.5 State-Machine Diagram for initial routing by the 2-SafeLinks Algorithm

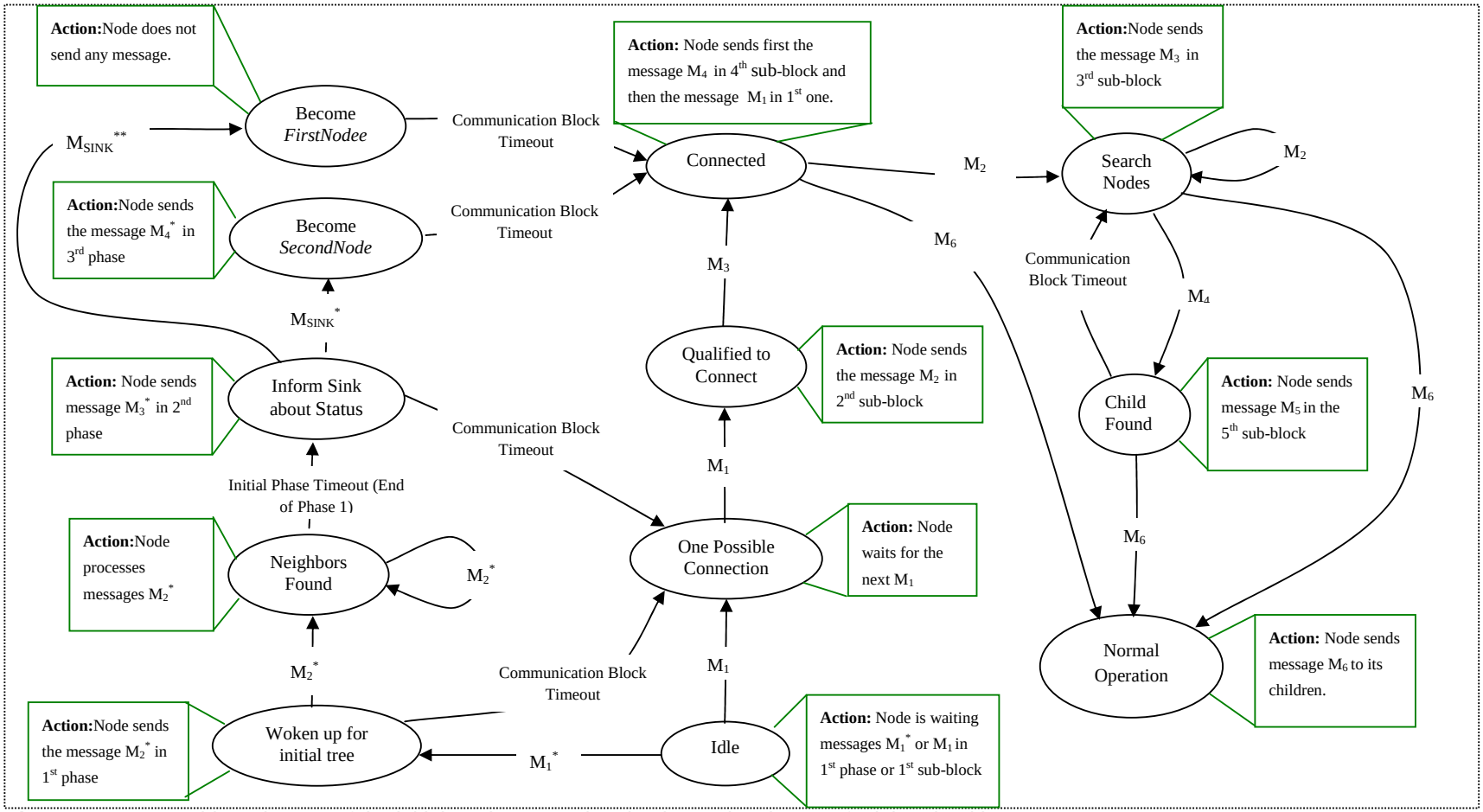


Figure 5: State-Machine Diagram of the initial routing procedure implementing the 2-SafeLinks Algorithm.

3.3.6 Timing for the initial routing in Glomosim

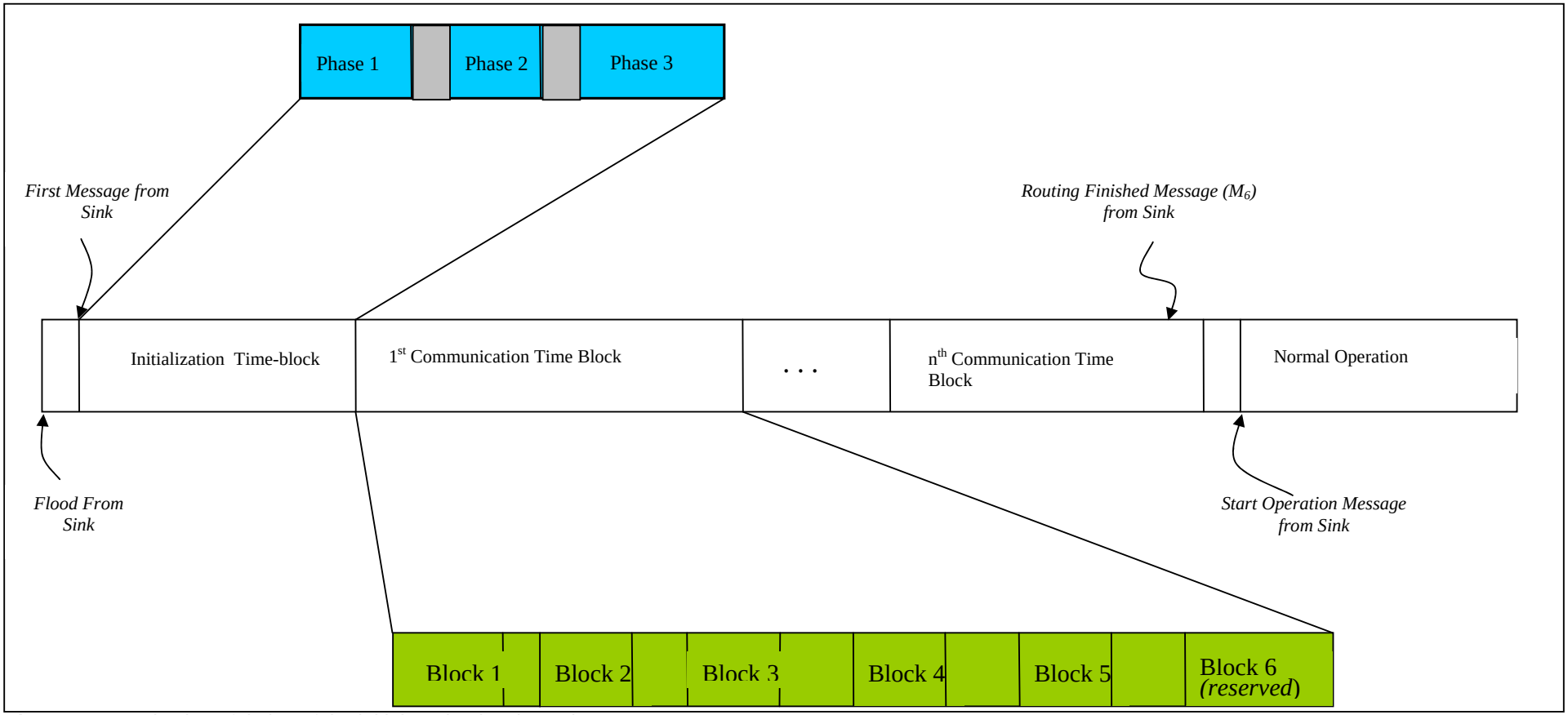


Figure 6: Organization of timing of the initial routing in Glomosim.

4. Double Tree and Local Sinks Concepts

CWSN Application is developed for the fire-detection wireless sensor networks. In these multi-hop networks, the sensors send periodic monitoring messages towards the sink. This messages are accumulated at every hop and forwarded to the next hop. In case of a fire event the sensor detecting this event sends an alarm message to the sink over the hops. The latency of the fire alarm cannot exceed τ_{Fire} specified in [1]. The network must also be able to inform the sink in case of node crashes within a certain latency τ_{Fail}. The network is most of the time in the normal operation, where only monitoring message are sent and received. Hence the network should be organized to minimize the power consumption in the normal operation. However, the network must also be reactive enough in case of fire events or node crashes. The latencies and power consumption must be kept low even if the environment is varying, where some links may fail.

4.1 Dynamic and Static Routing

We term a routing algorithm dynamic if in case of link breaks re-routing and routing traffic is required. A static algorithm on the other hand uses fix routing tables generated during the initialization phase to react on link breaks.
Dynamic routing has the advantage to re-generate optimal routing trees, when some links break permanently. However, as it needs message exchange, whenever re-routing is needed, it can lead to significant extra power consumption. Also it might lead to instable states: the quality of wireless links may fluctuate over time which may cause permanent re-routing activity and traffic.
Static routing on the other hand, needs an initial phase where the routing tables are generated. All message exchange takes place during this phase. For the given initial link qualities an optimal routing tree can be determined (If during normal operation updates of the link qualities are received the routing tables might be resorted though).
Due to the low power requirements of the CWSN application it was decided to concentrate on static routing algorithms. However to utilize static routing, a new way to create the routing tables was needed, which guarantees that any link breaks that might occur in the net would not break the connection to the sink. Assuming that up to *N*-1 links per node may break, the algorithm termed *N-SafeLinks Algorithm* was developed (see section 3.2) that still guarantees connectivity and preserved tree structure.

4.2 Double Tree Concept

4.2.1 Main Idea: MST for Monitoring and SPT for Alarms

Since for the given radio the power consumed by sampling or listening $\hat{P}_R$ is almost as large as the power consumed by sending data $\hat{P}_T$ it was essential to increase the sampling period T_w. As discussed in section 4.4 the wakeup period T_w is constrained by the maximal number of hops in the tree for the fire alarms. Hence for the alarms it is required to have a tree with a minimal hop count. On the other hand for the cost of monitoring it is required to have a tree with only few children per parent (see section 4.4.5). The double tree concept uses simultaneously two different trees to meet both requirements:

- A SPT[11] for alarms: . The SPT algorithms have the tendency to generate graphs with numerous branches but small number of hops. Minimizing hop count maximizes T_w for the given allowed latency τ_{Fire} . Since alarm messages are not considered in the power balance the many branches per node are irrelevant.
- A MST[12] for monitoring: MST algorithms tend to produce routing trees with less number of brunches but a large number of hops. Moreover optimal link qualities avoid retransmissions. A minimal number of children per node minimize the power consumption.

For the SPT and MST, *2-SafeLinks* Algorithms is utilized. Using the *2-SafeLinks Algorithm* every node has a total of four potential links, two for the alarms and two for monitoring.
In Figure 7 and Figure 8 the alarm and monitoring trees of a randomly generated network with 81 nodes are shown.

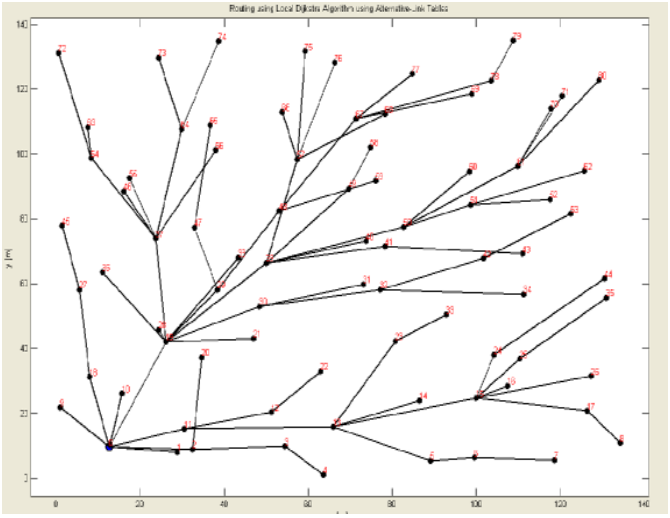


Figure 7: Alarm tree of a randomly generated network with 81 nodes generated by the *2-SafeLinks Dijkstra Algorithm*. The maximum number of branches in the tree is $K_{\max} = 8$ whereas the maximum number hops is only $h_{\max}^{alarm} = 6$.

[11] SPT is found using *2-SafeLinks Algorithm* using *Dijkstra's Algorithm* and hence it is not an optimal shortest path tree but a sub-optimal one. To see the difference of the SPT metrics with *Dijkstra's Algorithm* and *2-SafeLinks Algorithm* using *Dijkstra's Algorithm* refer to section 5.3.1

[12] MST is found using *2-SafeLinks Algorithm* using *Prim's Algorithm* and hence it is not an optimal shortest path tree but a sub-optimal one. To see the difference of the MST metrics with *Prim's Algorithm* and *2-SafeLinks Algorithm* using *Prim's Algorithm* refer to section 5.3.1

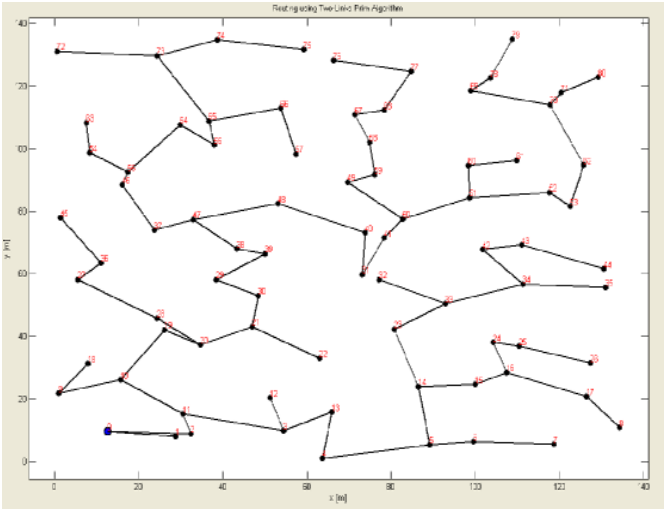


Figure 8: Monitoring tree of the same network with 81 nodes generated by the *2-SafeLinks Prim Algorithm*. The maximum number of branches is only $K_{\max} = 2$ whereas the maximum number of hops is $h_{\max}^{mon} = 24$.

4.3 Local Sinks: A Clustering and Supervising Concept

From the analysis discussed in section 4.4 it is obvious that the power consumption depends on the monitoring interval Δ_M. It must be selected as large as possible for low power consumption. However because of the maximum allowed latency for node failures τ_{Fail} must not be exceeded in order to increase Δ_M, the hopcount or the maximum number of hops in the monitoring tree should be reduced. Since this is not possible without increasing the number of children per node and hence a higher power consumption, a *Local Sinks Concept* was developed. This concept is based on partitioning the routing tree into clusters which are monitored by *Local Sinks*. At the moment a local sink detects a node failure in its cluster or subtree, it generates a *Technical Alarm Massage*. This message is sent over the alarm tree with low latency. Thus the maximum number of hops in the monitoring tree $h_{\max}^{mon}$ does not limit monitoring interval Δ_M and the power consumption can be reduced compared to using the standard monitoring tree that is generated using *Prim Based 2-SafeLinks Algorithm*. The *Local Sinks Concept*, which will be explained in section 4.3.1 , is an extension of the *N-SafeLinks Algorithm*.

4.3.1 The Concept: Parallelization of the N-Safe-Links Algorithm

In the standard *N-SafeLinks Algorithm* all nodes except the sink are treated commonly whereas with the *Local Sink Concept* it is allowed that some nodes act like a sink and they construct their own clusters or subtrees, i.e., the tree that has the *local sink* as its root node. Nevertheless it should be guaranteed that the alternative links of the nodes in a subtree will always remain in the subtree and the nodes in the subtree of a specific *local sink* cannot make any connection with nodes from other subtrees. This is achieved by reformulating the *definition of eligibility* in the *N-SafeLinks Algorithm*. In the new formulation, a node *j* that is not added to the current tree is *eligible* to make a connection with the node *i*, which is already in the current tree if node *j* has at least *N* possible links to the subtree holding node *i*.

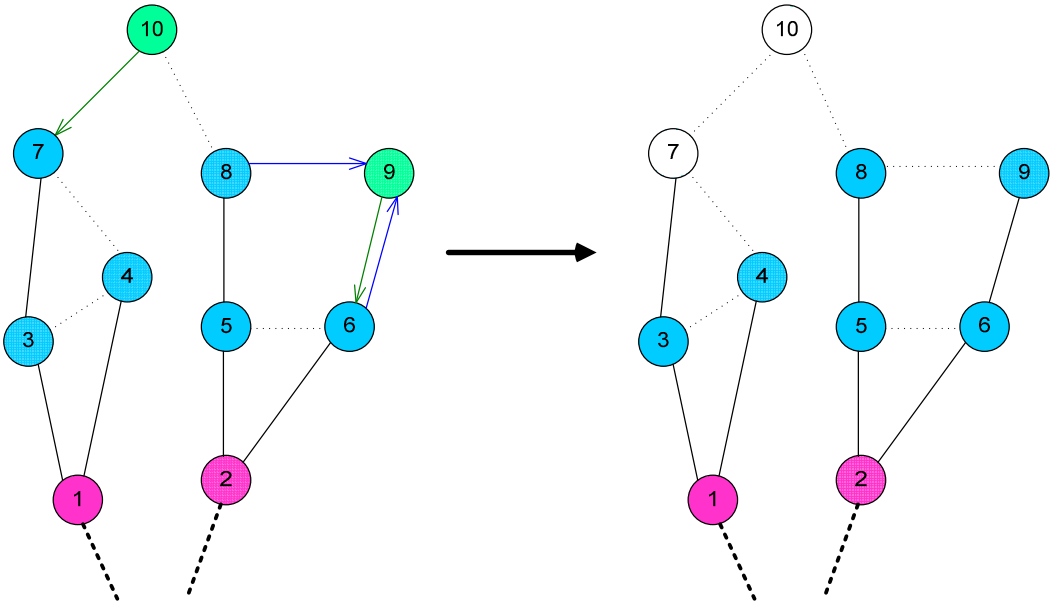


Figure 9: An iteration of the *2-SafeLinks Algorithm* with *Local Sinks Concept.* This example illustrates the new *eligibility* definition for a node to be added to the current tree. Nodes 1 and 2 are the *local sinks* shown in pink, which are connected to the current tree with the thick dotted lines. The blue nodes with numbers 3-8 are other nodes existing in the current tree. The nodes 9 and 10 are potential child nodes, which are not added to tree yet (on the left). The solid black lines are the links of the current tree and light dotted lines are the possible links of the nodes. On the left the node 9 selects node 6 as its favorite parent (green arrow), whereas the node 6 selects node 9 as favorite child (blue arrow). The node 10 selects the node 7 as its favorite parent, whereas the node 8 selects node 9 as favorite child. The node 9 has two possible connections to the subtree of node 2 and hence it is an *eligible* node and is selected by node 6 and a link is setup between node 6 and 9. The node 10 selects node 7 as its favorite child but it has only one link to the subtree of the *local sink* 1 and therefore it is not an *eligible* node and is neglected and not selected by node 7.

The *Local-Sinks Concept* can also be seen as a parallelization of the *2-SafeLinks Algorithm* as the different local sinks appearing in different places of the graph can construct their subtrees in parallel. The maximal allowed hopcount in a subtree is $h_{\max}^{subtree}$. A node at hop $h_{\max}^{subtree}$ from its subtree becomes a *local sink* if it is possible and builds its own subtree. Every *local sink* is also a normal node in another subtree. However being $h_{\max}^{subtree}$ away from its *local sink* is not enough for a node to become a local sink. If this node can finalize the initial phase[13] of the *N-SafeLinks Algorithm* then the node is declared as a *local sink*. If the topology does not permit this initial phase, the node will be declared as a *leaf node* and it does not have the permission to make any further links with other nodes. The iteration steps of the *Local Sinks Concept* with *N-SafeLinks Algorithm* are given as

1. The list of the eligible nodes for each subtree of the current tree is updated

[13] The initial phase of the *N-SafeLinks Algorithm* is generating an initial tree with *N* nodes. See section 3.2.2.1 for details.

2. Each node in the current partial spanning tree selects a *favorite child* node (if any) from the eligible nodes for its subtree.

3. Each node existing in the graph but not a member of the current tree selects a *favorite parent* (if any) among the nodes in the current tree.

4. The nodes in the current tree, which are *favorite parent* of their *favorite child*, set up a link with these nodes.

5. Among the nodes that are newly added to the current tree, the ones $h_{\max}^{subtree}$ hops away from their *local sink*, try to generate their initial tree. The ones achieving this become new *local sinks*. The ones, which fail to generate an initial tree, become *leaf nodes*.

An example of an algorithm iteration is illustrated in Figure 10. As the *Local Sinks Concept* is an extension of the *N-SafeLinks Algorithm*, the metric to be optimized can be selected freely. The *Local Sinks Concept* can be easily applied to *Prim-Based* or *Dijkstra-Based N-SafeLinks Algorithm*. The selection of the favorite parents and children is generic and it is carried out according to the given metrics, i.e., MST or SPT. It is important to mention that like the *N-SafeLinks Algorithm*, the *Local Sinks Concept* with *N-SafeLinks Algorithm* does not find the optimal MST and SPT but a suboptimal variant of them because of the restrictions during the construction of the spanning tree. The tests and simulation results for the difference from the optimal MST and SPT when using *Local Sinks Concept* is given in section 5.3.1 .

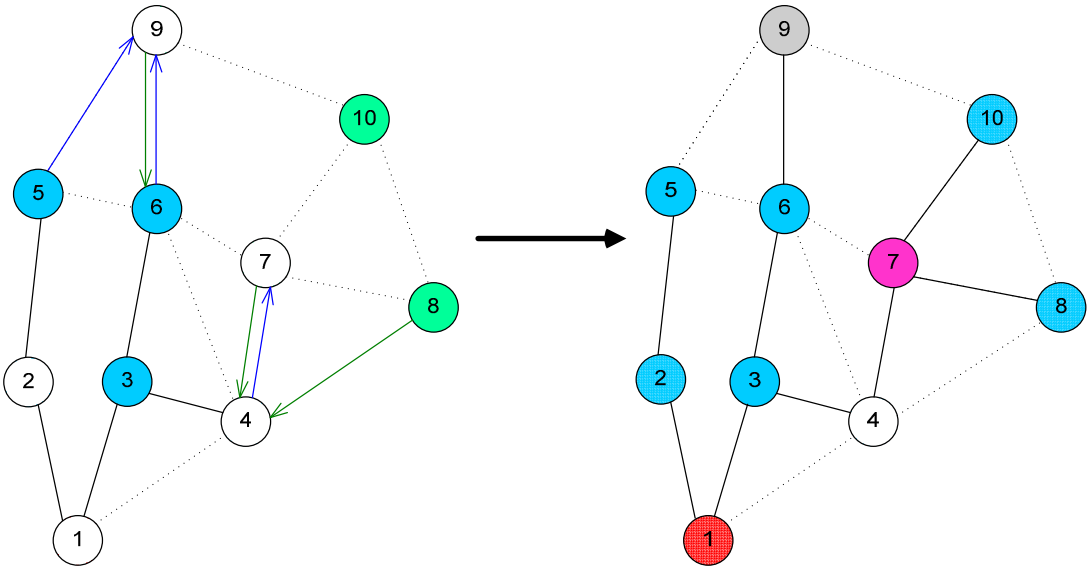


Figure 10: An iteration of *2-SafeLinks Algorithm* with the *Local Sinks Concept* at an intermediate step. For this example $h_{\max}^{subtree}$ =3. The sink node is in red and the blue nodes with numbers 2-6 are other nodes existing in the current tree. The solid black lines are the links of the current tree and

dotted lines are the possible links of the nodes. The nodes 7, 8, 9 and 10 are potential child nodes, which are not added to tree yet (on the left). The node 9 selects node 6 as its favorite parent (green arrow), whereas the node 6 selects node 9 as favorite child (blue arrow). The same happens with node 7 and 4 as well. Node 5 selects node 9 as its favorite child and node 8 selects node 4 as its favorite parent (on the left). In this way, node 8 and 6 are connected to their favorite parents. However, both node 6 and 8 are 3 hops away from the sink and they should become *local sinks*. Node 7 has 2 possible connections to node 8 and 10, which also have a possible connection between them. Thus, node 7 becomes a *local sink* (shown in pink) and with nodes 8 and 10 it constructs its initial tree. The other added node 10 does not have 2 possible connections to other nodes and it cannot setup an initial tree. Because of that it becomes a *leaf node* (shown in grey), which is not eligible to setup new connections.

The differences in the resulting topology from the standard *Prim Based Two-SafeLinks Algorithm* and from its *Local-Sinks* extension are illustrated in Figure 11 and Figure 12.

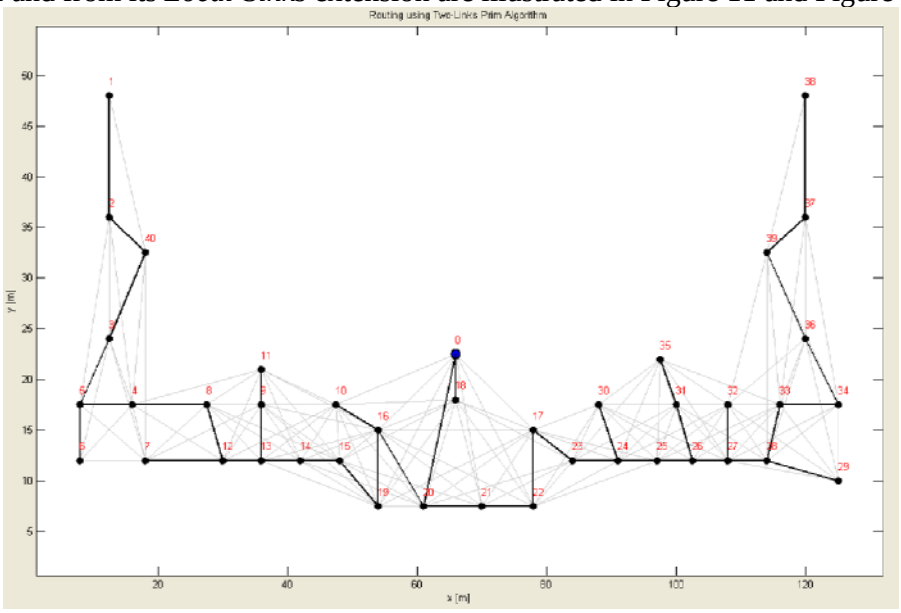


Figure 11: Routing tree for the Bundeshaus scenario using the 2-SafeLinks Algorithm with Prim's metric.

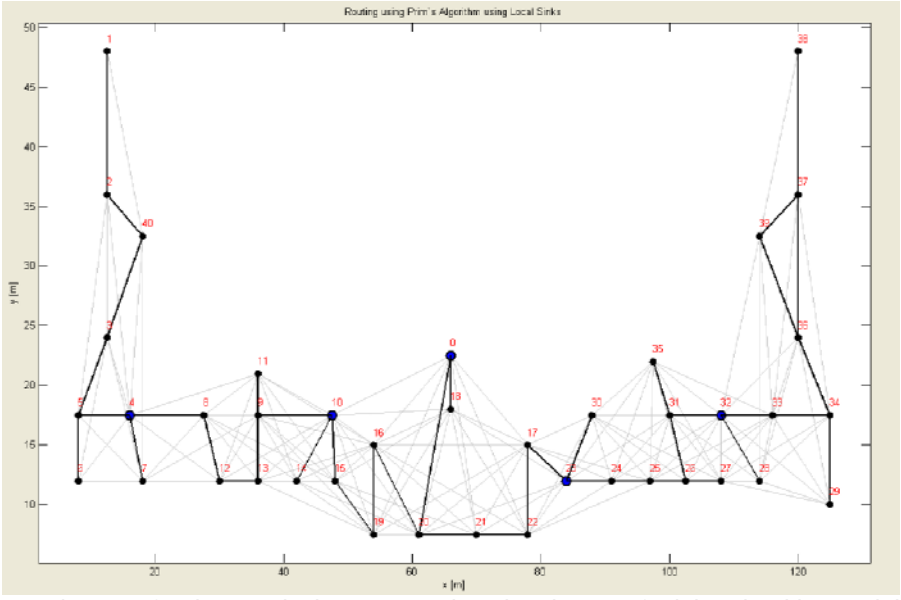


Figure 12: Routing tree for the Bundeshaus scenario using the 2-SafeLinks Algorithm and the *Local-Sinks Concept* with Prim's metric. Local sinks in blue.

For simplicity, in the rest of the thesis we term the *Prim-Based 2-SafeLinks Algorithm* with *Local Sinks* as *Local Sinks Algorithm*. The next section presents the detailed algorithm.

4.3.1.1 *Algorithm 8: Local Sinks Algorithm*

Inputs:

E : Set of edges in the graph
W : Set of edge-weights
V : Set of vertices
Cost(i,j) : Cost function that outputs the cost of the connection between node i and j.
$h_{\max}^{subtree}$: Maximum number of allowed hops in subtree

Outputs:

E_T : Edges of the current tree
LocalSinks : List of the local sinks
ParentTable :The matrix with size, "number of V's elements" X "2". The node ID refers to row number. First and second entries in a row refer to the main and alternative parents.

Temporary Variables:

T : Set of the vertices added to the routing tree
T' : Set of the vertices not added to the routing tree yet
Parent(i) : Parent node or predecessor of node i
Leafs : The list holding the leaf nodes
$NodesSink_i$: The address of the local or global sink of node i
FP_j : Favorite Parent of node j. It is comprised of 2 entries, favorite parent ID i and the cost of the link between the nodes j and i, e.g., FP_j={j,Cost(j,i)}
FC_i : Favorite Child of node i. It is comprised of 2 entries, favorite child ID j and the cost of the link between the nodes i and j, e.g., FC_i={i,Cost(i,j)}
$PPL_{i,k}$: Possible links list of the node i to the nodes with the local sink k. It is comprised of concatenated node IDs and costs of the link to these nodes pairs, e.g., PPL_{ik}=({$Node_m$,Cost(i,m)},{$Node_n$,Cost(i,n)}, …)
FirstNode : The first node of a subtree after the global or local sink
SecondNode : The second node of a subtree after the global or local sink
BestCost : At some iteration step the the link with best cost, not necessarily the least weight.

Initialization:

T := {*Sink*}
T':= V / {*Sink*}
E_T:= {}
FirstNode := {}
SecondNode := {}
BestCost := INF
FP_j := {Null, INF} $\forall j \in V$
FC_i := {Null, INF} $\forall i \in V$
$PLL_{i,j}$:= {} $\forall i, j \in V$
$NodesSinks_i$:= {} $\forall i \in V$

Leafs := {}
LocalSinks := {}

Main Algorithm:

AddFirstNode(*Sink*)
AddSecondNode(*Sink*,*FirstNode*)
While T != V
  For $\forall i \in T \setminus Leafs$
    For $\forall j \in T'$
      If $\{i, j\} \in E$
        $PLL_{j,NodesSink_i} := PLL_{j,NodesSink_i} \cup \{\{j, \text{Cost}(i, j)\}\}$

        If (Cost(*i*,*j*) < $FC_i(2)$)
          $FC_i(2)$:= Cost(*i*,*j*)
          $FC_i(1) := j$
        End
        If (Cost(*i*,*j*) < $FP_j(2)$)
          $FP_j(2)$:= Cost(*i*,*j*)
          $FP_j(1) := i$
        End
      End
    End
  End

  For $\forall i \in \boldsymbol{T}$
    $j := FC_i(1)$
    If (i=$FP_j(1)$ **And** $\left|PLL_{j,NodesSink_i}\right| \geq 2$)
      $T := T \cup \{i\}$
      $T' := T \setminus \{i\}$
      $E_T := E_T \cup \{\{i, j\}\}$
      Parent(*j*) :=*i*
      If FindHopsToLocalSink (*j*) == $h_{\max}^{subtree}$
        AddFirstNode(*j*)
        AddSecondNode(*j*,*FirstNode*)
        If *SecondNode* != {}
          $LocalSinks := LocalSinks \cup \{j\}$
        Else
          $Leafs := Leafs \cup \{j\}$
        End
      End
    End
  End
  FP_j := {Null, INF} $\forall j \in V$
  FC_i := {Null, INF} $\forall i \in V$
End

Sub-Routines:

AddFirstNode(SinkAddress i):

For $\forall j \in \boldsymbol{T}$
If (Cost(*i*,*j*) < *BestCost*)
FirstNode:=*j*
BestCost := Cost(*i*,*j*)
End
End
BestCost := INF
$PLL_{FirstNode,i}$:= ({*i,* Cost(*i*,*FirstNode*)})

AddSecondNode(SinkAddress *i*, FirstNodeAddress *k*):

For $\forall j \in \boldsymbol{T}$
If (Cost(*i*,*j*) < *BestCost*)
FirstNode:=*j*
BestCost := Cost(*i*,*j*)
End
End
BestCost := INF
$PLL_{SecondNode,i}$:= ({*i,* Cost(*i, SecondNode*)})
NodesSink(*FirstNode*) := *i*
NodesSink(*SecondNode*) := *i*
$T := T \cup \{FirstNode, SecondNode\}, T' := T'/\{FirstNode, SecondNode\}$
$E_T := E_T \cup \{\{i, FirstNode\}, \{i, SecondNode\}\}$
ParenTable(*FirstNode*) := {Sink, SecondNode}
ParenTable(*SecondNode*) := {*Sink, FirstNode*}

GeneratedTables:

For $\forall i \in \boldsymbol{T} \setminus \{Sink\}$
If *ParentTable*{i} == {}
PLL_i := **SortInAscendingOrder (**PLL_i**)**
If $PLL_i(1,1)$ = *Parent*(*i*)
ParentTable(*i*) := {*Parent*(*i*),$PLL_i(2,1)$}
Else
ParentTable(*i*) := {*Parent*(*i*),$PLL_i(1,1)$}
End
End
End

FindHopsToLocalSink(node *i***):** It is a subroutine that finds the number of hops of the input node *i* to its local or global sink.

4.3.2 Block Diagram of Local Sinks Algorithm

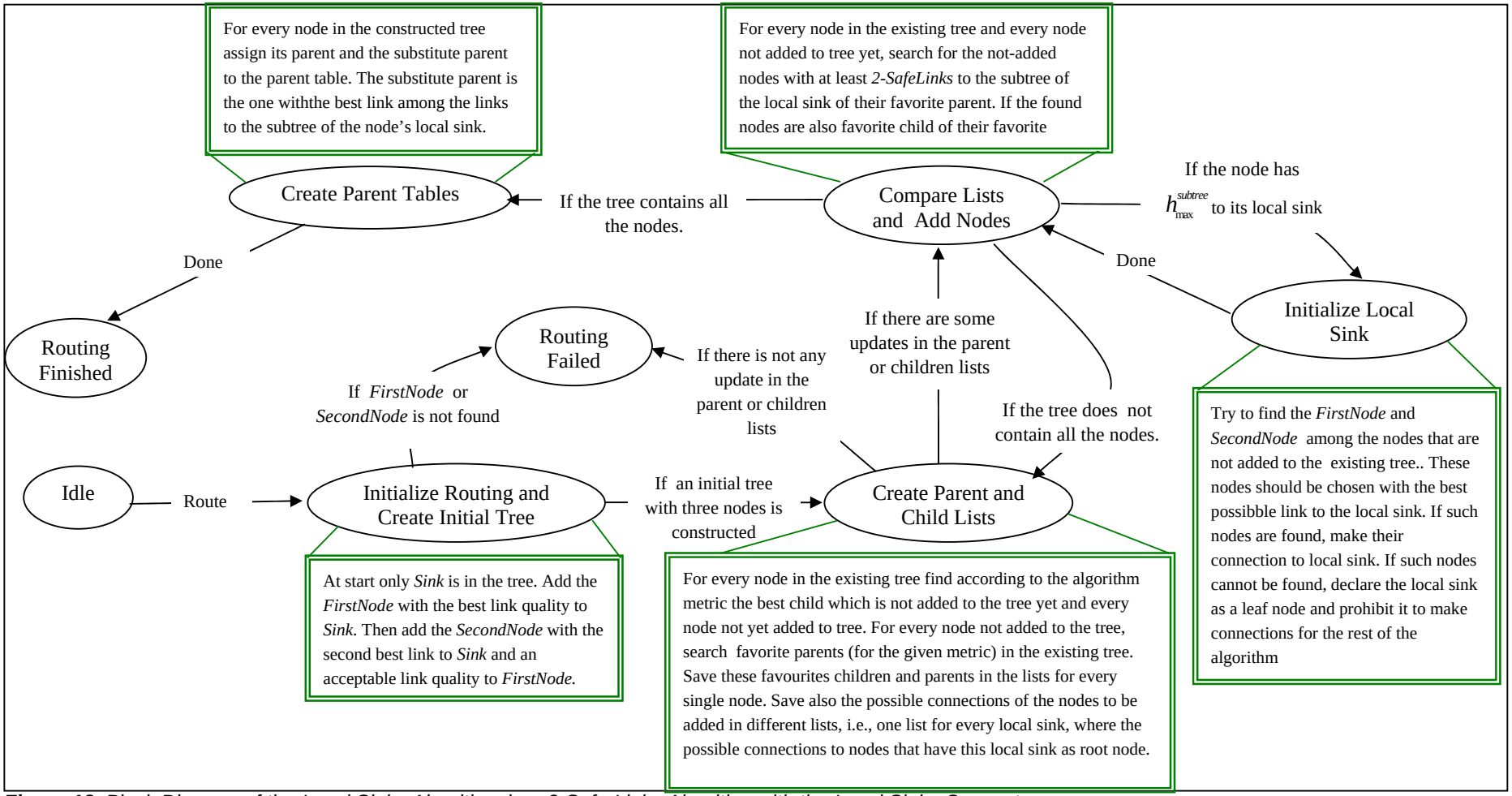


Figure 13: Block Diagram of the *Local Sinks Algorithm*, i.e., *2-Safe-Links Algorithm* with the *Local Sinks Concept*

4.3.2.1 *Complexity*[14]

The *N-SafeLinks Algorithm* with *Local Sinks Concept* has $O(n)$ iterations, where *n* is the number of nodes in the network. Each iteration to construct the spanning tree takes $O(d)$ time, where *d* is the *node degree* and when the tree is constructed, for every node to create its *parent table* needs $O(m)$ iterations. Hence the overall time complexity of the algorithm becomes $O(n \cdot (d + m))$.

4.3.3 Convergence Issues

As the *Local Sinks Concept* is an extension of the *N-SafeLinks* algorithm, the limitations for the convergence of the *N-SafeLinks* (see section 3.2.6) also holds for *Local Sinks Concept* extension. However for this new version the eligibility of the nodes are not determined by the number of the possible links they have to the current tree but it is determined by the number of possible links to each subtree in current tree. Because of this, it is expected that for the convergence the *Local Sink* extension needs a higher *node degree*[15] than the standard *N-SafeLinks Algorithm.* An other important factor for the convergence is the maximum allowed subtree hops $h_{\max}^{subtree}$. The less this number is the more subtrees the generated graph has and the more links a node should have to be eligible. In the extreme case when $h_{\max}^{subtree}$ is not bounded the algorithm becomes the standard *N-SafeLinks* algorithm. Hence it is presumable that for a larger $h_{\max}^{subtree}$ a smaller *node degree* is sufficient. The simulation results for the convergence are illustrated in section 5.3.2.0 .

4.4 Power Consumption

In this section, the power consumption of the relevant periodic protocol functions are analyzed (expressions are derived in [3]). For the calculations, it is assumed that double tree concept is utilized, i.e., one tree with less hops and numerous branches is used for the alarm and one tree with more hops but less branches is used for monitoring. The alarm tree has the number of maximum hops $h_{\max}^{alarm}$ and the maximum number of branches K_{Alarm}. The monitoring tree is constructed with *Local Sinks Concept* and has the maximum number of subtree hops $h_{\max}^{subtree}$ and maximum number of branches K_{Mon}. Before starting with the power calculations, we should introduce the relations between different protocol parameters and their posed constraints.

[14] Complexity refers to the time complexity of the off-line computation of the spanning tree with local sinks.

[15] Node degree of a graph is the mean connectivity of the nodes existing in that graph.

4.4.1 Protocol Parameter Dependencies and Constraints

4.4.2 Wake-Up Interval

The maximal allowed wake-up period is constrained by the latencies τ_{Fail} and τ_{Fire} where at the point of operation τ_{Fire} is the actual limiting factor. A worst case analysis gives (approximately, neglecting T_P, T_{DA}, T_T and T_S; see [3] for the exact term):

$$T_w = \frac{\tau_{Fire}}{(h_{\max}^{alarm} + N_{retries}^{alarm})} \quad . \tag{3}$$

Worst case, the time for next wakeup and hence the waiting time is T_w for every hop. In addition we add $N_{retries}^{alarm}$ to consider late messages.

If τ_{Fire} has to be met only with a certain probability, T_w can be larger. In average the waiting time per hop is $T_w / 2$. Thus, if T_w is chosen as

$$T_w = \frac{2\tau_{Fire}}{(h_{\max}^{alarm} + N_{retries}^{alarm})} \quad . \tag{4}$$

The average transmission time will be approximately τ_{Fail} (see [3] for exact term).

4.4.3 Monitoring Interval

If a node failure has to be detected with the maximum latency of τ_{Fail} and with a hop interval of Δ_h the maximal allowed monitoring interval is

$$\Delta_M = \tau_{Fail} - (h_{\max}^{subtree} + N_{retries}^{mon})\Delta_h - \tau_{Fire} \, , \tag{5}$$

where $h_{\max}^{subtree}$ is the number of hops to the local sink that supervises the node. In case some retransmissions are needed the hop count is increased by a margin of $N_{retries}^{mon}$. A technical alarm propagates to the local sink within τ_{Fire}.

4.4.4 Periodic Wake-Up

The power consumed by the periodic wake-up is inversely proportional to the wake-up interval T_w [3]:

$$P_{Sampling} = \frac{\hat{P}_S T_S + \hat{P}_R T_I}{T_w} \tag{6}$$

For the given radio $P_{Sampling}$ it is the largest portion in the overall power. Thus a maximization of T_w is crucial. The maximum of T_w is constrained by the allowed alarm latency τ_{Fail} **(3)**. Typical values of $P_{Sampling}$ are given in Table 3.

T_w [s]	$P_{Sampling}$ [μW]
0.5	184
0.95	100
1	92
1.2	80
1.5	62
2	46
4	23

Table 3: Power consumed by periodic wakeup.

4.4.5 Monitoring

The power P_{Mon} consumed by the monitoring activity is mainly determined by the monitoring interval Δ_M, the number of children K of a node and the preamble length T_P

$$P_{Mon} = \frac{K\left(\hat{P}_S T_S + \hat{P}_R\left(\frac{T_p}{2} + \frac{L_{DATA}}{B} + T_T\right) + \hat{P}_T\left(\frac{L_{ACK}}{B}\right)\right) + \hat{P}_S T_S + \hat{P}_T\left(T_p + \frac{L_{DATA}}{B}\right) + \hat{P}_R\left(T_T + \frac{L_{ACK}}{B}\right)}{\Delta_M}, \quad (7)$$

where

$$\Delta_M = \tau_{Fail} - (h_{\max}^{subtree} + N_{retries}^{mon})\Delta_h, \quad (8)$$

$$\Delta_h = N_S T_w, \quad (9)$$

and

$$T_P = 4\theta\Delta_M. \quad (10)$$

Typical values of P_{Mon} are given in Table 4.

K	$h_{\max}^{subtree}$	Δ_M [s]	T_w [s]	P_{Mon} [μW]	θ [ppm]
2	5	220	1.5	23	5
3	5	220	1.5	31	5
4	5	220	1.5	38	5
5	5	220	1.5	45	5
3	5	218	1.5	31	5
3	6	206	1.5	32	5
3	7	194	1.5	34	5
3	10	154	1.5	41	5
3	15	98	1.5	64	5
3	20	38	1.5	161	5
3	5	218	1.5	47	30
3	15	98	1.5	81	30

Table 4: Power consumed by periodic monitoring traffic using the OK-BIT, i.e., only one bit is used to indicate the presence of the source node instead of the all status-array. Parameters: $N_{retries}^{mon} = 1$ and $\Delta_h = 12s$.

4.4.6 Link Testing

With the given frequency schema [19], every node must be synchronized with the nodes to which it sends messages. This requires a node to periodically synchronize with its potential parents (two for monitoring M_1 and M_2, and two for alarms A_1 and A_2). Since only M_1 is used periodically for monitoring, a node has to send empty link testing packages to three nodes and receive the acknowledge messages. In addition if a node is a tested parent from K_{Mon} children on the M_2 monitoring tree and $2\,K_{Alarm}$ children on the alarm trees respectively the power consumed for link testing amounts to:

$$P_{Link} = \frac{(K_{Mon} + 2K_{Alarm})\left(\hat{P}_S T_S + \hat{P}_R\left(\frac{T_p}{2} + \frac{L_{Link}}{B} + T_T\right) + \hat{P}_T\left(\frac{L_{ACK}}{B}\right)\right) + 3\left(\hat{P}_T\left(T_p + \frac{L_{Link}}{B}\right) + \hat{P}_R\left(T_T + \frac{L_{ACK}}{B}\right)\right)}{\Delta_L} \quad (11)$$

Here Δ_L is the link testing interval and L_{Link} is the empty link testing packet length. The used preamble is now $T_P = 4\theta\Delta_L$

Typical values of P_{Link} are given in Table 5. The interval Δ_L has to be small enough to guarantee local synchronization, i.e., $T_P = 4\theta\Delta_L < T_w$. Local synchronization is kept with $\theta = 30$ ppm if $\Delta_L < 3.5$ h. It is obvious that good clock drift estimation below $\theta = 5$ ppm is required to keep P_{Link} low. The clock drift estimation process itself also poses an upper bound on Δ_L.

Δ_L [h]	P_{Link} [μW]	θ [ppm]	$T_P = 4\theta\Delta_L$ [ms]
1	20	5	72
2	17	5	144
3	16	5	216
10	14	5	720
1	88	30	432
2	84	30	864
3	83	30	1296
10	not possible	30	4320

Table 5: Power consumed by the testing of alternative links. Parameters: $K_{Mon} = 3$, $K_{Alarm} = 6$ and $T_w = 1.5$ s.

4.4.7 Total Power

The total power amounts to

$$P_{tot} = P_Z + P_{Sampling} + P_{Mon} + P_{Link}. \quad (12)$$

Typical values of P_{tot} are given in Table 6.

K	$h_{\max}^{subtree}$	$h_{\max}^{alarm}$	Δ_M [s]	Δ_L [h]	T_w [s]	θ [ppm]	P_{tot} [μW] without/with link testing
3	5	6	220	1	1.5	5	91/112
3	5	6	220	1	1.5	30	109/197

Table 6: Total power consumed in the protocol normal operation phase using the OK-BIT.

4.4.8 Lifetime of Batteries

As we pointed out before, the aimed power consumption for the communication module is $P_{target} \sim 100\mu W$. In this section, the typical batteries and their lifetime statistics are presented in Table 7 [2].

Lifetime [years]	Maximum Average Power [μW]		
	4* AA-Alkaline 1.5V(+20°C)	4* AA-Alkaline 1.5V(-10°C)	4* AA-Lithium 1.5V
3	350	150	340
4	240	90	230
5	170	60	170
6	120	-	120
7	-	-	90

Table 7: Operational lifetimes versus maximum average power at the transceiver for different battery types. The two columns for the AA-Alkaline battery refer to different temperatures (+20 and -10 °C) of the environment. The AA-Lithium batteries have almost the same lifetime statistics temperatures between -10 and +50°C . The cost of AA Alkaline battery set (four batteries) is 0.84€ whereas the AA-Lithium set costs 3.08€.

From the given table, we see that with 100 μW maximum transceiver power, lifetimes over 5 or even 6 (for lithium battery) can be reached. One should keep in mind though that the lithium battery costs almost four times more than alkaline batteries. On the other hand, the lifetime statistics of the alkaline battery (at +20°C) seem to be very close to the lithium but it drops drastically for low temperature. The nodes are planned to be installed inside the buildings and hence low temperatures are not expected in these environments. However, fluctuations in the temperature are expected also for indoor usage and hence the practical performance of alkaline batteries should be estimated worse than the performance at 20°C. Although, it is not declared which battery will be used for the CWSN application, because of cost advantages, AA-Alkaline batteries are the strongest candidate. From the given table and our observations, we can conclude that the power should be kept under $150\mu W$ (possibly close to $100\mu W$) for an operation time of 5 years or more.

5. Simulation and Evaluation of Algorithms

5.1 MATLAB Routing Graph Analyzer Tool

A routing graph analyzer was implemented to develop and test new routing algorithms, obtain statistics and to visualize the sensor networks and the routing graphs. The tool can also create *Routing Information Tables* for a given scenario which are then used by the GloMoSim implementation of the protocol. MATLAB was selected as the development environment because of its flexible and useful built-in functions and easily programmable GUI tool. The MATLAB routing graph analyzer looks as in Figure 14 when it is started.

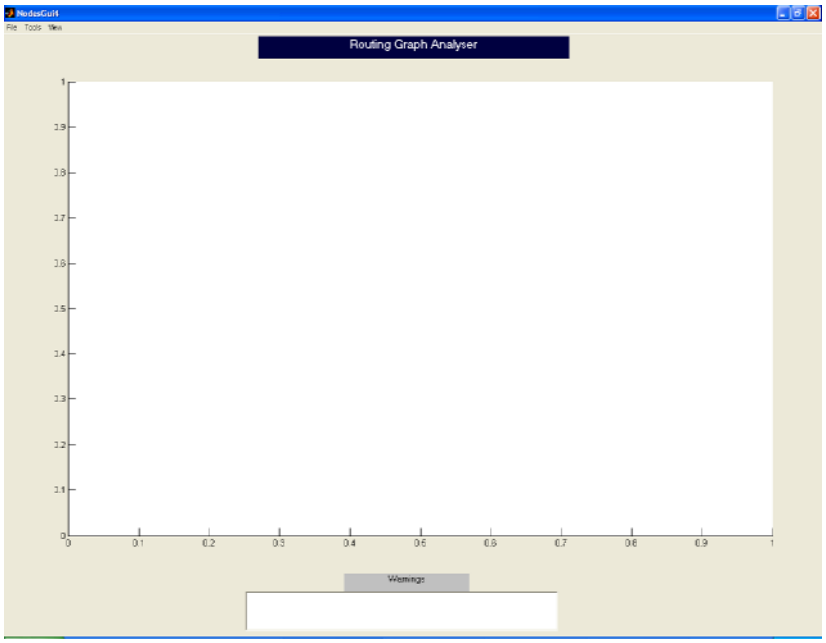


Figure 14: Start screen of the MATLAB Routing Graph Analyzer

During the progress of this work, the analyzer tool has been continuously developed. New features are added and the ones which became irrelevant to the project were removed. In its final version, there are three main menus: *File, Tools* and *View*.

5.1.1 Files

From the *File Menu*, the user can open different type of files. Randomly generated network files (with *.rnd extension) and obtain only the node positions. For this type of files the edge weights are calculated by the analyzer tool (see sections 5.2). The other option is to open one of the scenarios, which are provided by SBT[16]. To analyze the scenarios, two files must be opened, which have *.in extensions. One of them (nodes.in) holds the node positions whereas the other one (pathloss.in) contains the matrix, whose entries are the path losses between nodes in dBm. The analyzer tool again maps these values into RSSI values (see section 5.2.2). When a file is opened, the graph is displayed on the screen of the analyzer tool as in Figure 15.

[16] SBT: Siemens Building Technologies, Siemens Schweiz.

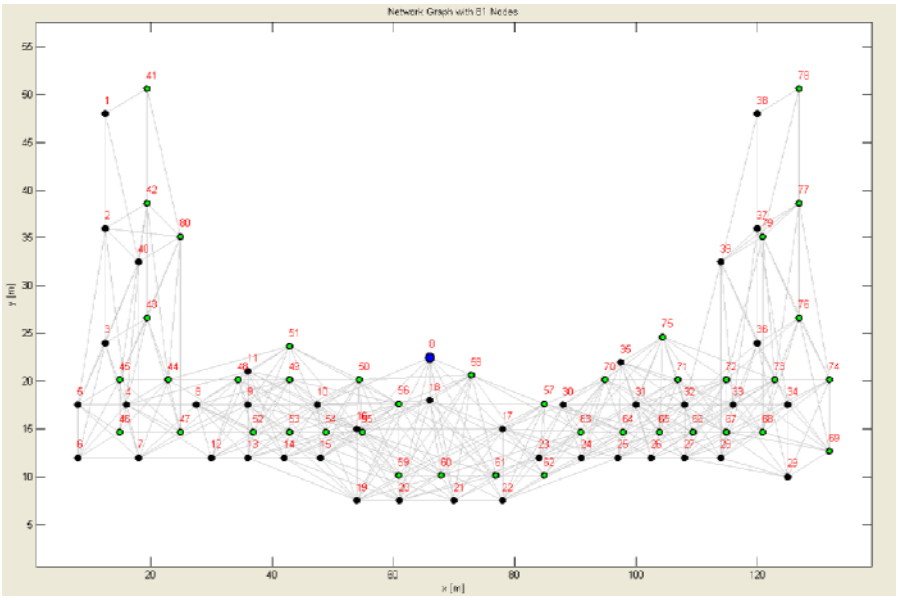


Figure 15: Display of routing graph comprised of 81 nodes with their physical links. The fire detection nodes in the first floor are displayed in black, the ones in the second or higher floors (if any) are displayed in green. The sink is displayed in blue if it is on the first floor, otherwise it is displayed in violet.

5.1.2 Tools

5.1.2.1 *Routing the graphs*

From the *Tools Menu*, the user can select the operation to be applied with the current graph. The most essential operation is routing the graph. In the final version of the tool, the list of the implemented SPT or SPT-based algorithms are

- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- Dijkstra Based Pairing Algorithm
- Dijkstra Based N-SafeLinks Algorithm

whereas the list of the MST or MST-based algorithms are the followings

- Prim's Algorithm
- Borůvka's Algorithm
- Prim based Pairing algorithm
- Prim based N-SafeLinks algorithm
- Prim-Based N-SafeLinks algorithm with Local Sinks Concept

In Figure 16, generated routing trees with different routing algorithms are illustrated.

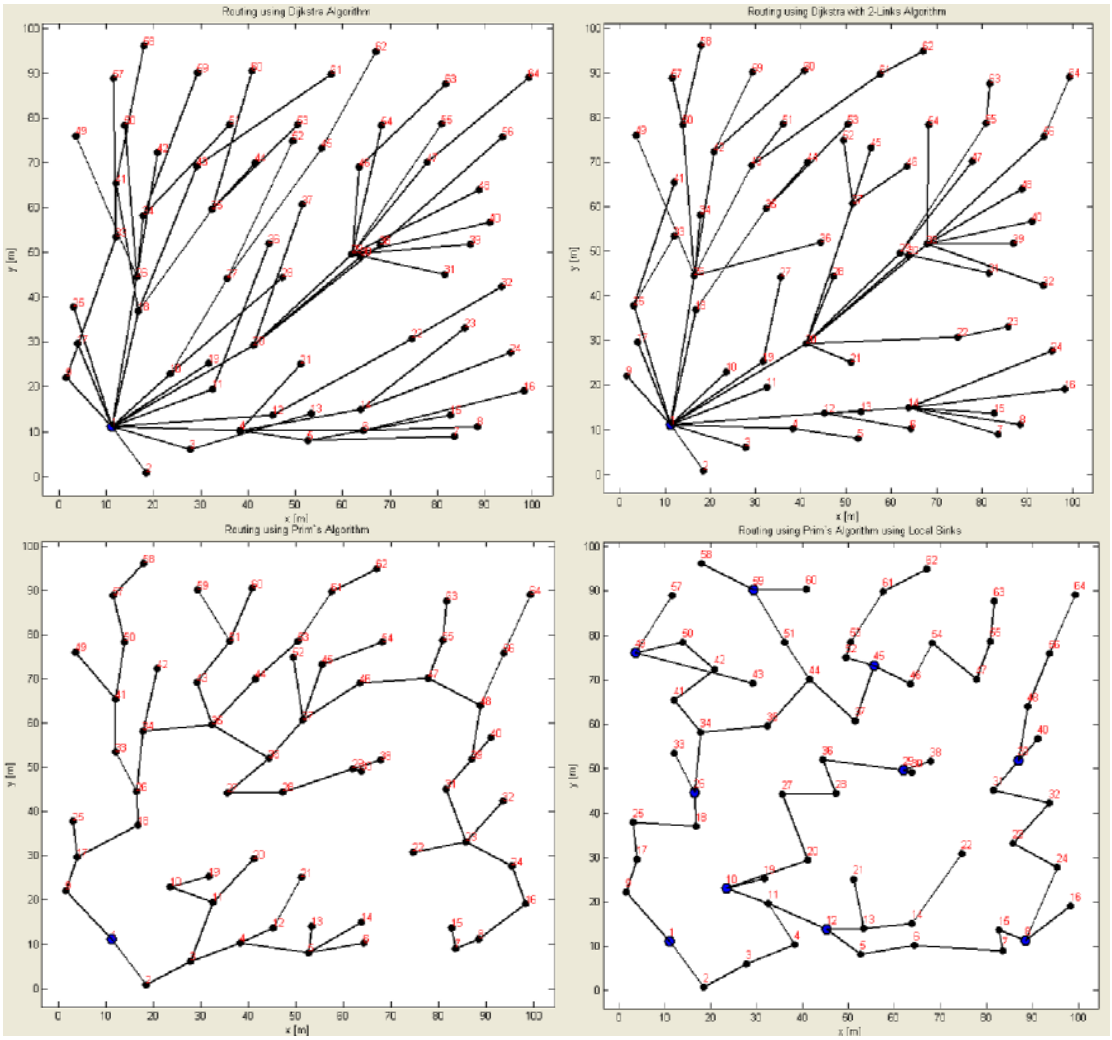


Figure 16: 4 different spanning tree generated from the same weighted graph comprised of 64 nodes. The utilized routing algorithms are *Dijkstra's Algorithm* (upper left), *Dijkstra Based 2-SafeLinks Algorithm* (upper right), *Prim's Algorithm* (lower left) and *Prim Based 2-SafeLinks Algorithm* with *Local Sinks Concept* (lower right).

5.1.2.2 *Simulation of Link Breaks*

From the *Tools Menu*, one can select the option *Simulate Link Breaks*. In this mode, if a link on the graph is clicked, the link is broken. If the algorithm is a parent table driven one, where the alternative parents are saved in lists, the algorithm directly changes to the alternative parent. If it is not case, the algorithm re-routes the graph without using the broken link (see Figure 17). If the *Simulate Link Breaks* option is not selected, for the algorithms that generate parent tables, these tables can be displayed by clicking on the nodes.

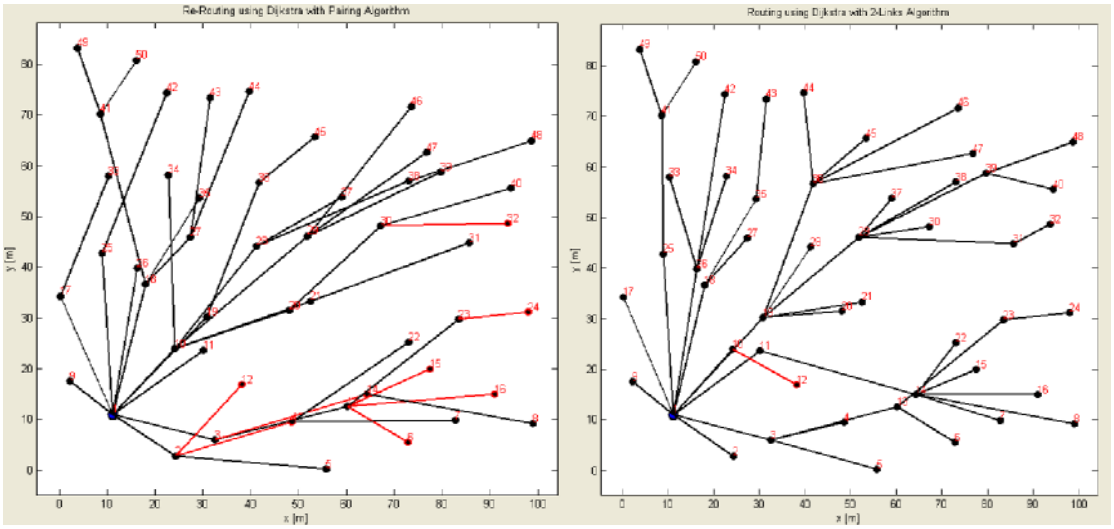


Figure 17: Simulation of link breaks. The link of node 12 towards the root node is broken. With *Dijkstra Based Pairing Algorithm*, a re-routing takes place and many nodes change their active links shown in red (on the left). On the other hand, as *2-SafeLinks Algorithm* is an algorithm that creates parent tables, only the node loosing the link (node 12) changes its active link, the rest of the links (in black) remain the same (on the right).

5.1.2.3 *Extraction of the Routing Information Tables*

The *Routing Information Table* is a text file that is designed to be read from the GloMoSim network simulator. The table has necessary routing information for every node. There are four lines for each node specifying the standard monitoring, alternative monitoring, standard alarm and alternative alarm parents.
A line specifies the node address, type of the node (0 for ordinary nodes, 1 for global sink and 2 for local sinks), parent address, address of the local sink, number of hops to the global sink, number of hops to the local sink and RSSI[17] of the link to this parent. An example of the routing information table is illustrated in Figure 18.

```
# ROUTING-INFORMATION-FILE
# FORMAT: NodeAddr NodeType ParentAddr LocalSink HopsToSink HopsToLocalSink RSSI
# NOTE: There are four lines for every node.
#       First line is for monitoring parent
#       Second Line is for substitute monitoring parent.
#       Third is for alarm and fourth is for substitute alarm parent
  0   1   0   0   0   0   0
  0   1   0   0   0   0   0
  0   1   0   0   0   0   0
  0   1   0   0   0   0   0
  1   0   0   0   1   1   9
  1   0   8   0   2   2   8
  1   0   0   0   1   1   9
  1   0   8   0   2   2   8
  2   0   1   0   2   2  10
  2   0   0   0   1   1   8
  2   0   0   0   1   1   8
  2   0   1   0   2   2  10
  3   0  11   0   4   4  10
  3   0   2   0   3   3   9
  3   0  11   0   2   2  10
  3   0   1   0   2   2   8
  4   2   3   0   5   5  10
  4   2  11   0   4   4   8
  4   0  11   0   2   2   8
  4   0   2   0   2   2   8
  5   0   4   4   6   1   9
  5   0  12   4   7   2   9
  5   0  12   0   3   3  10
  5   0   3   0   3   3   8
```

Figure 18: A part of the Routing Information Table.

[17] For the RSSI mapping refer to section 5.2.2 .

The Routing Information Tables can be generated by selecting *Extract Routing File* from the *Tools Menu* after a scenario is opened properly. The user is prompted to save the produced file with an '*.in*' extension. After that, the file is ready to be used in the GloMoSim without any further processing or formatting.

5.1.2.4 *Other Tools Menu Options*

The statistics of the routed graph can be displayed by selecting the *Show Statistics* option in the *Tools Menu*. The statistics contain distributions for the number of branches, number of hops, and link qualities of the graph as well as the power consumption per node based on the given parameters and constraints of the CWSN (see section 4.4). At last, the *Step by Step Routing* option in *Tools Menu* can be selected to illustrate the construction steps of the routing tree.

5.1.3 View Features

This menu is thought for the display preferences of the user. Objects such as grids, link qualities, available links and node IDs can be hidden or displayed (see Figure 19). Note that for scenarios, transmission range is not defined since link qualities are given by a *pathloss matrix* (in pathloss.in file). However, the transmission range can be displayed for randomly generated graphs (files with *.rnd extension). In the calculation of these ranges a *log-distance model* is used with a *decay-index* of 3.5 along with other transceiver and propagation parameters (see section 5.2.1). An other feature termed *View Menu* opens the current graph separately, which is helpful for comparing different routing algorithms. This action is triggered by selecting *Open Graph Separately* option.

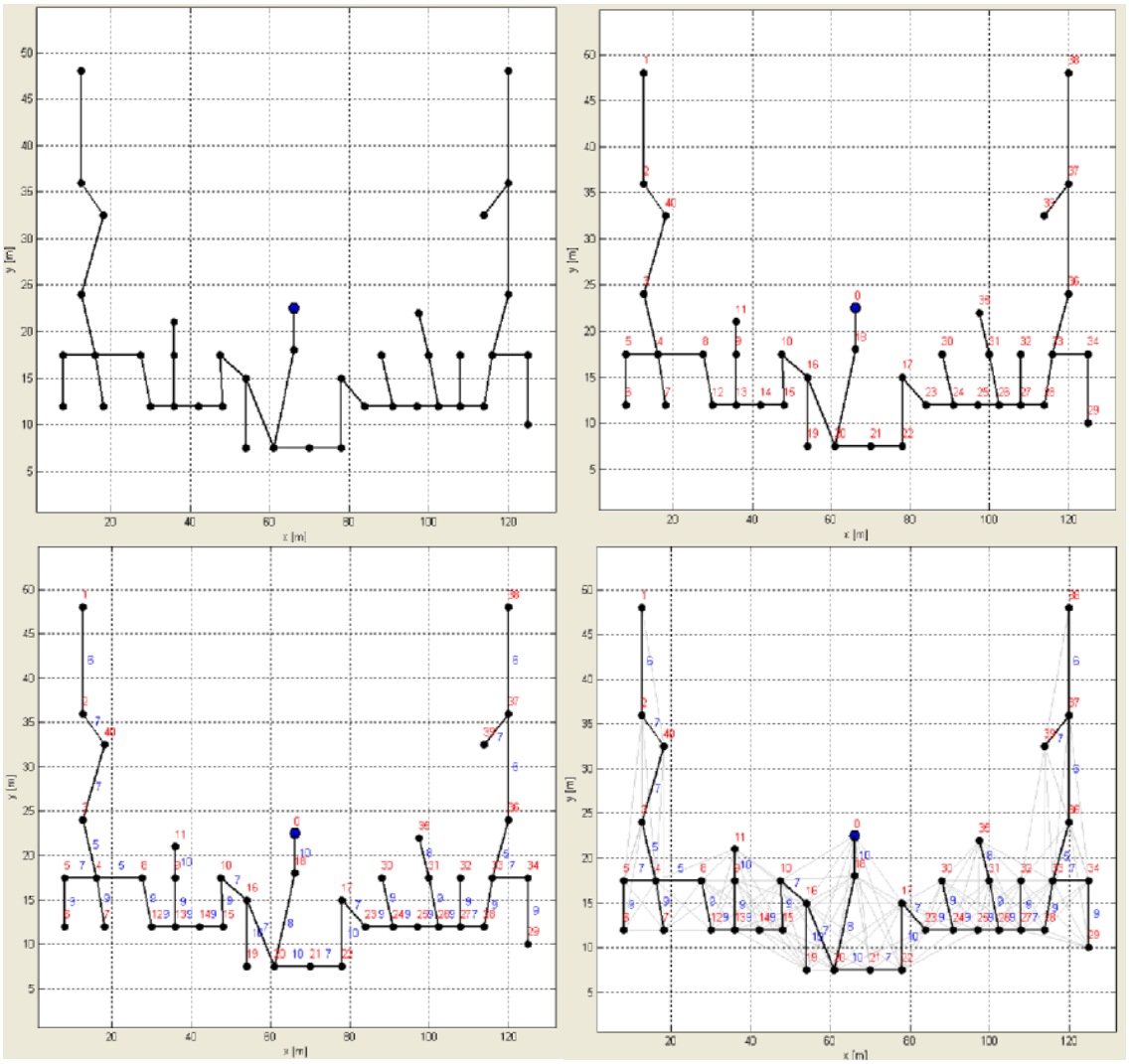


Figure 19: Graph only with grids (upper-left), with node numbers and grids (upper right), link qualities also attached to that (lower left), available links added to them (lower right).

5.2 Preliminaries for Testing

5.2.1 Determining the Qualities

We analyzed two different sources for testing, which are

1. The generated scenarios provided by SBT
2. Randomly generated graphs

For the first test source, we were given a *Path-Loss Matrix* that contains the simulated received signal strength for every link in the graph. For the second case, we calculated the received signal strengths according to the following parameters.

Radio TX-Power	8.5 dBm
Radio RX-Threshold	-81 dBm
Loss on Receiver	2 dBm
Loss on Transmitter	2 dBm

Table 8: The radio parameters used for the the simulation of link qualities

We used a *free-space* propagation model until 1 meter and *log-distance* propagation model with a decay index $\alpha = 3.5$, in order to simulate the indoor conditions. Using the radio parameters with this propagation model, a transmission radius of approximately 35m is obtained.

5.2.2 RSSI-Level Mapping

The routing algorithms use discrete RSSI (Received Signal Strength Indication) as an indicator for the link quality. Table 9 shows how RSSI levels are mapped from the received signal strength. We used 10 RSSI levels for our tests.

RSSI Level	**Signal Strength** P_R [dBm]
0	$P_R < -113$
1	$-113 < P_R < -108$
2	$-108 < P_R < -103$
3	$-103 < P_R < -98$
4	$-98 < P_R < -93$
5	$-93 < P_R < -88$
6	$-88 < P_R < -83$
7	$-83 < P_R < -78$
8	$-78 < P_R < -73$
9	$-73 < P_R < -68$
10	$-68 < P_R$

Table 9: Mapping of the received signal strength to the RSSI Levels.

5.2.3 Calculation of the Weights from the RSSI-Values

5.2.3.1 *Weights for the MST Metric*

From section 2.1.1 , we have seen that the objective function of MST is the sum of the link costs of the spanning tree (see equation (1)). MST algorithms minimize this objective function. However, as we use the RSSI mapping for the link weights, we obtain link qualities instead of costs for each link. For that reason we used the following mapping to find the cost of some link $\{i,j\}$ from its RSSI value as

$$\mathrm{Cost}_{\mathrm{MST}}(i,j) = \mathrm{RSSI}_{\max} - \mathrm{RSSI}_{ij} + 1 \quad , \tag{13}$$

where $\mathrm{RSSI}_{\max}$ is the maximum of the RSSI levels, RSSI_{ij} is the RSSI level of link $\{i,j\}$. A constant c (here c=1) is added to make every link cost positive. It could be shown that for any c the MST algorithms converge to the same results. The reason is that the MST algorithms select the subset of the links that is a spanning tree and the minimum in total cost. The number of the links in a spanning tree are fixed, which is one less from the number of nodes in the graph. Hence, the selection of c cannot make any difference between the objective function values of different spanning trees. We selected c=1 because of practical purposes for all the simulations and comparisons.

5.2.3.2 *Weights for the SPT Metric*

From section 2.1.1 , we have seen that the function to be minimized for SPT is the sum of the node distances to the sink in the spanning tree (see equation (2)). The distance of a node to the sink is given as

$$\mathrm{Dist}(j) = \sum_{\{i',j'\}\in P_j} \mathrm{Cost}_{\mathrm{SPT}}(i', j') \ , \tag{14}$$

where P_j is the path from sink to node j. As it was for MST we needed to map the link qualities in RSSI to link costs. For SPT algorithms we used the following mapping for an arbitrary link $\{i,j\}$

$$\mathrm{Cost}_{\mathrm{SPT}}(i, j) = \mathrm{RSSI}_{\max} - \mathrm{RSSI}_{ij} + 3 \ , \tag{15}$$

where $\mathrm{RSSI}_{\max}$ is the maximum of the RSSI levels, RSSI_{ij} is the RSSI level of link $\{i,j\}$. A constant c (in the case c=3) is added. The selection of this constant is made heuristically and it must be noted that in opposition to the MST, the selection of this constant change the behavior of the SPT algorithm. The reason is that the constant is added as many times as the number of total hops in the spanning tree, which can differ from one spanning tree to another. Apparently, the larger the constant c is selected the more hops cost. Because of this reason we selected a constant higher than 1 to reduce the number of the hops in the graph. From the simulations we carried out, we deduced that selecting higher constants do not reduce the number of hops significantly but just degrades the selected link qualities. Hence we carried out further experiments for c=3.

5.3 Evaluation and Analysis of the New Algorithms

5.3.1 MST and SPT Metrics

In this section, we examined the distance of the developed routing algorithms to the optimal MST and SPT algorithms in terms of the given MST and SPT metrics (see section 5.2.3). The effect of the factors such as node degree, link breaks, network size and terrain geometry on these distances are analyzed.

5.3.1.1 *Dependence on Node Degree*

In this section, the correlation of the SPT and MST metrics with node degree[18] is analyzed and evaluated for different developed algorithms. The algorithms are repeatedly run on random graphs with varying node degrees. For each network, we considered a square terrain. The terrains are divided into equal square regions. Within every region, one node is located randomly. The considered network had 64 nodes and by varying the size of the terrain dimensions (preserving the square geometry) we obtained graphs with different node degrees. The simulation results are shown in Figure 20 and summarized in Table 10.

[18] The node degree is the average number of potential links per node in a graph.

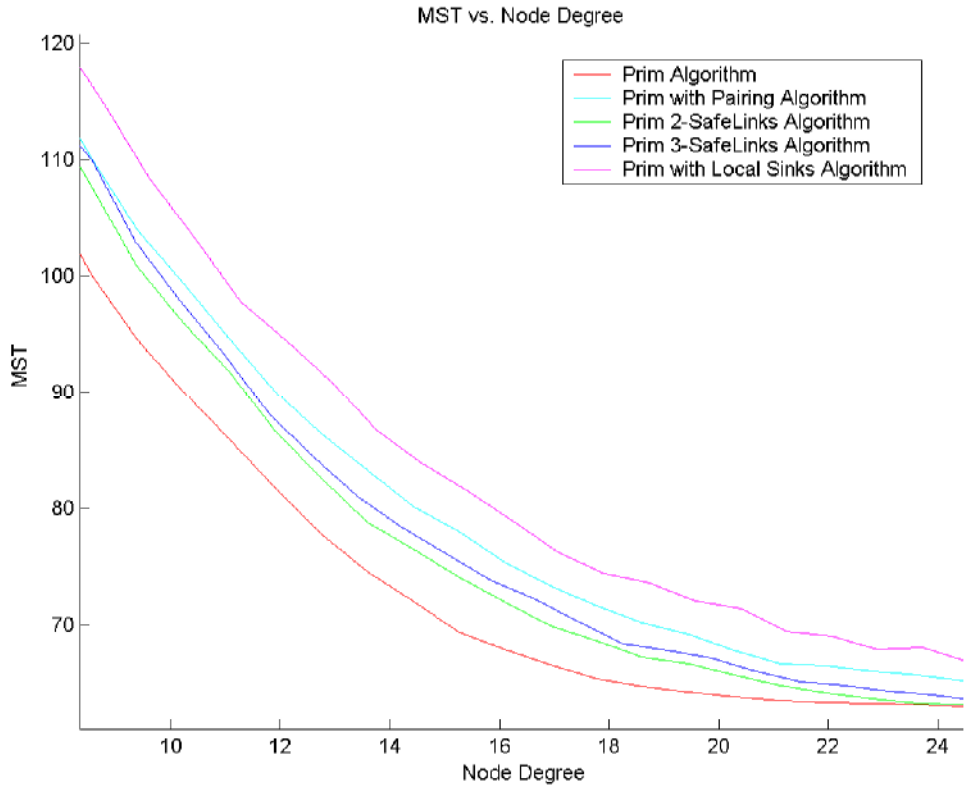


Figure 20: Resulting MST metric for the *Prim's*, *Prim-Based Pairing*, *Prim-Based 2-SafeLinks*, *Prim-Based 3-SafeLinks* and *Prim-Based Local Sinks* algorithms on numerous graphs with 64 nodes and varying node degrees.

Node Degree	Nominal MST metric [%]					Absolute MST metric				
	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks
10	100	109.9	106.8	108.3	114.3	93.3	102.5	99.6	101	106.6
15	100	111.4	106.0	107.4	116.3	71.3	79.4	75.6	76.6	82.9
20	100	106.6	102.8	103.9	111.9	64	68.2	65.8	66.5	71.6

Table 10: The comparison of the absolute and nominal MST metric for the developed algorithms. On the right, the absolute values of the MST metric are given. The numbers are obtained by averaging over 1000 simulation results. On the left, the nominal values of the MST metrics are given. *Prim's Algorithm* is an optimum algorithm for MST, therefore its nominal value is assigned as 100%.

From the simulation results that are illustrated in Figure 20 and summarized in Table 10, we can make the following observations:

- There is a negative correlation between node degree and MST metric. The reason is that with increasing node degree, more links with higher link qualities become available.

- With increasing node degree, the resulting metrics of the developed algorithms get closer to the MST (see Table 10). Again the reason is that more possible high quality links become available with increasing node degree in the graph.

- The *Prim-Based 2-SafeLinks* and *3-SafeLinks* algorithms are the closest algorithms to the optimum *Prim's Algorithm*, which finds the MST. *2-SafeLinks Algorithm* functions slightly better than the *3-SafeLinks* as it has less limitations.

- The *Prim-Based Pairing Algorithms* generates worse results than the limited *N-SafeLinks* algorithms. A possible reason is that *Pairing Algorithm* is a *greedy* algorithm, which attaches nodes in parallel to the current tree. Hence there is the possibility that the nodes make links, which are not optimal, because they do not have other options. With the introduced limitations of *N-SafeLinks*, the nodes must have at least *N* possible links to be added, hence the probability of having a better link is higher. We see that for *N*=2 this limitation improves the MST metric, yet increasing *N* further to 3 does not bring further improvement. It even deteriorates the performance slightly, because the limitation factor dominates. The nodes having high quality links cannot make connections because they do not have enough possible links.

- The *Prim-Based Local Sinks Algorithm* has the worst performance. The reason is that the limitations for creating new links are stronger than in the other considered algorithms. A node can only make connections when it has at least 2 possible links to the certain subtree. Apart from that, when a node becomes a local sink it has to create its initial tree, where two other nodes are connected to the local sink. These limitations prevent the algorithm to select the links with the highest link quality.

Figure 21 illustrates the dependence of the SPT metric on the node degree and the Table 11 summarizes these results.

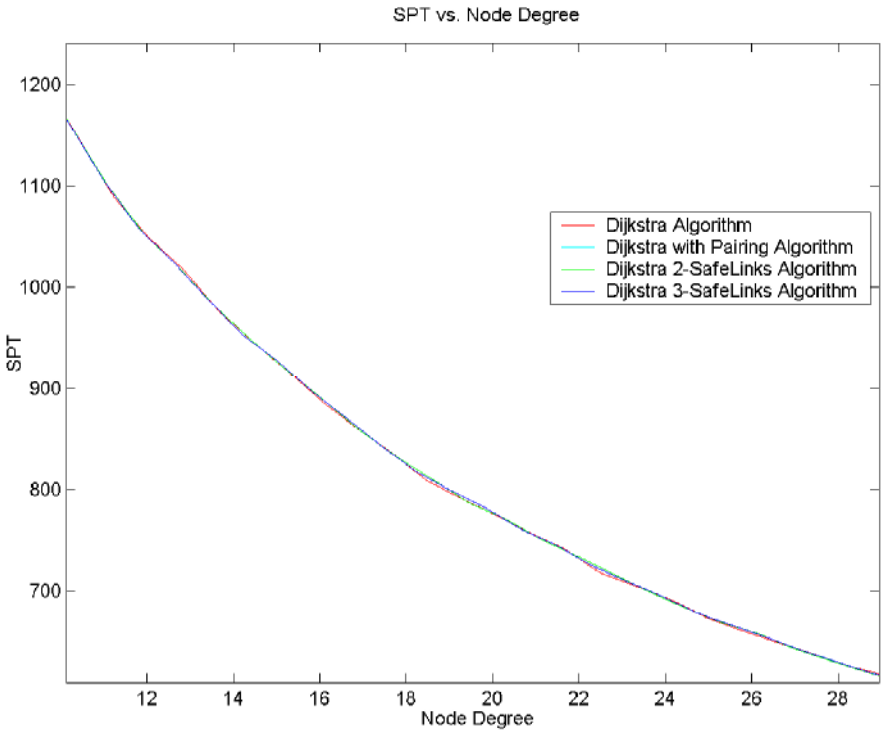


Figure 21: Resulting SPT metric for the *Dijkstra's*, *Dijkstra-Based 2-SafeLinks* and *Dijkstra-Based Pairing* algorithms on the graph with 64 nodes.

Node Degree	Nominal SPT metric [%]			Nominal SPT metric [%]		
	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks
10	100	101.8	101.8	1175.7	1197	1197.4
15	100	100.2	100.2	934	935.6	935.7
20	100	100.8	100.8	775.3	781.2	781.3

Table 11: The comparison of the absolute and nominal SPT metric for the developed algorithms. On the right, the absolute values of the SPT metric are given. On the left, the nominal values of the SPT metrics are given. *Dijkstra's Algorithm* is an optimum algorithm for the SPT, therefore its nominal value is assigned as 100% .

The observations we obtained from Figure 22 and Table 11 are as follows:

- There is a negative correlation between node degree and SPT metric. The reason is that with increasing node degree, more links with higher link qualities become available.

- In opposition to the MST metric, the *Dijkstra-Based Pairing* and *N-SafeLinks Algorithms* behave very similar to the optimum *Dijkstra's Algorithm* in terms of the SPT metric. It is an interesting result as irrespective of the introduced limitation of *N-SafeLinks Algorithm* or *greedy Pairing Algorithm*, the algorithms generate very similar (with a maximum of 2% difference in SPT metric) graphs. An important factor is the selection of weights for the SPT metric (see section 5.2.3). As the effect of the hops are amplified with that selection, for each *Dijkstra-Based* algorithm there is the tendency to construct links that are not having minimum costs, i.e., best qualities in order to keep the number of hops low. In the case of the *Prim-Based* algorithms, a node with an optimum link can be neglected because it does not have enough possible links to the current tree. On the other hand, for *Dijkstra-Based* algorithms, there is a higher possibility that the preferred link does not have a minimum cost. As evident from the results, these tendencies dominates the introduced limitation for *Dijkstra-Based Algorithms* and the generated trees are very close to the optimal.

5.3.1.2 *Effect of Network Size*

To see the effect of the network size, we considered 3 networks with 36, 64 and 100 nodes respectively. The Figure 22 illustrates a network with 36 nodes.

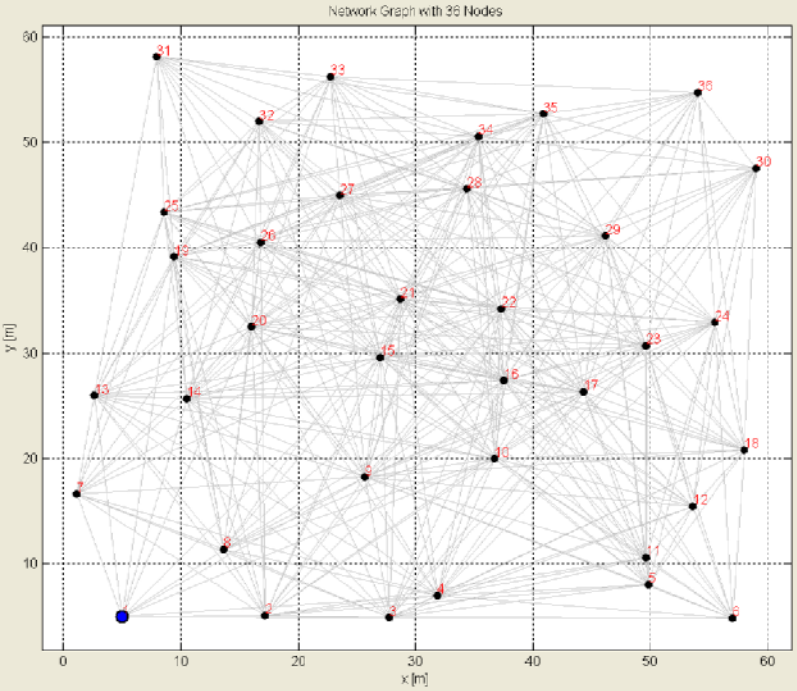


Figure 22: A network with 36 nodes. The sink is located in the lower left corner, whereas all other nodes are distributed in the ramaining square regions. Every region has exactly one node. The grey lines show the physical links between the nodes.

For each node number, we created numerous random networks of different sizes, on which we applied different routing algorithms, e.g., *Prim's Algorithm*, *Prim-Based Pairing Algorithm*, *Prim-Based 2-SafeLinks Algorithm* etc. In order to examine the difference in the MST and SPT metrics, we generated for each node number, networks of different node degree. It was realized by changing the edge lengths of the terrains, where the square geometry of the terrain is preserved. We considered the MST and SPT metrics that are defined in section 5.2.3 The results of the tests with different algorithms are summarized in Table 12 and Table 13.

Number of Nodes in the Network	Nominal MST metric [%]					Absolute MST metric				
	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks
36	100	105.8	107.5	108.8	115.0	45.98	48.66	49.45	50.01	52.88
64	100	105.9	107.3	110.5	116.1	75.26	79.7	80.74	83.13	87.36
100	100	107.1	106.9	115.3	117.1	126.65	135.68	135.33	145.98	148.34

Table 12: The nominal and absolute MST metric for networks with 36, 64 and 100 nodes.

Number of Nodes in the Network	Nominal SPT metric [%]			Nominal SPT metric [%]		
	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks
36	100	100.0	100.0	363.87	363.87	363.87
64	100	100.1	100.2	881.07	882.37	882.41
100	100	100.0	100.0	2203.1	2203.2	2203.3

Table 13: The nominal and absolute SPT metric for networks with 36, 64 and 100 nodes.

We observe from these tables that although there is a slight increase in the nominal MST metrics for the proposed algorithms, the network size does not have a significant effect on the MST and SPT metrics.

5.3.1.3 *Effect of Terrain Geometry*

To analyze the effect of the terrain geometry on the MST and SPT metrics for different algorithms, we considered networks with 64 nodes. For these networks, two types of terrain were studied. The first one was a square terrain, which is divided to 8x8 squares. In every square one node was located randomly. The sinks is placed in the lower left square (see Figure 23).

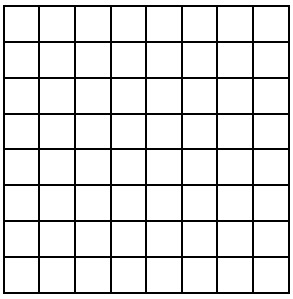


Figure 23: The square terrain that is comprised of 64 equal square regions.

The second terrain was a node chain with a width of 2 and length of 32 nodes. Again the nodes are located randomly on each of the 64 equally large squares (see Figure 24).

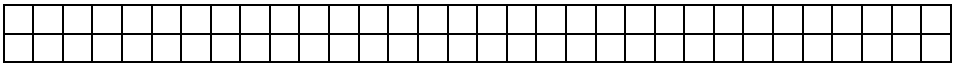


Figure 24: The terrain in the chain form that is comprised of 64 (2 x 32) equal square sub-terrains.

The simulation results are summarized in Table 14 and Table 15

Node Degree	Nominal MST metric [%]					Absolute MST metric				
	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks
10	100	109.9	106.8	108.3	114.3	93.3	102.5	99.6	101	106.6
15	100	111.4	106.0	107.4	116.3	71.3	79.4	75.6	76.6	82.9
20	100	106.6	102.8	103.9	111.9	64	68.2	65.8	66.5	71.6

Table 14: The nominal and absolute MST metrics for the networks with a square terrain.

Node Degree	Nominal MST metric [%]				Absolute MST metric			
	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with Local Sinks	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with Local Sinks
9	100	106.8	102.6	109.8	76.6	81.8	78.6	84.1
12	100	103.1	101.6	106.1	64.3	66.3	65.3	68.2
15	100	100.3	100.2	102.4	63	63.2	63.1	64.5

Table 15: The nominal and absolute MST metric for the networks with a 2x32 chain terrain.

From the comparison of the tables we can draw the following observations:

- The network with 2x32 chain-terrain has better MST metric values than the square terrain for every considered algorithm. The reason is that there are less possible paths to the sink in a network with a chain geometry than the network with square geometry. If the node degrees are the same, links on the possible paths must have better qualities for the chain geometry than the square geometry. This is also the reason, why the absolute minimum spanning tree, i.e. the tree with all links having the minimum cost of 1 is achieved with a node degree of 15 for the chain geometry whereas a node degree over 20 is needed for the square geometry.

- The absolute and nominal differences between the MST metrics of different algorithms are smaller for the chain geometry than the metrics for the square geometry. They are also closer to the optimal *Prim's Algorithm*. The reason is again that the selection of the possible paths are much more limited for the chain geometry than the square geometry and the possibility that all algorithms select similar paths is larger.

The differences between SPT metrics for the chain and square geometry are summarized in Table 16 and Table 17.

Node Degree	Nominal SPT metric [%]			Nominal SPT metric [%]		
	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks
10	100	101.8	101.8	1175.7	1197	1197.4
15	100	100.2	100.2	934	935.6	935.7
20	100	100.8	100.8	775.3	781.2	781.3

Table 16: The nominal and absolute SPT metrics for the networks with a square terrain.

Node Degree	Nominal SPT metric [%]			Absolute SPT metric		
	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks
9	100	100.0	100.0	2496	2496.1	2496.1
12	100	100.0	100.0	1866.2	1866.2	1866.2
15	100	100.0	100.0	1479.8	1479.8	1479.8

Table 17: The nominal and absolute SPT metric for the networks with a 2x32 chain terrain.

We can make the following observations from these tables:

- The SPT metric values are much higher for the chain geometry than the values for the square geometry. The reason is that a spanning tree needs much more hops for the chain geometry rather than the square geometry for a specified node degree. The hops are very expensive for the SPT metric (see section 5.2.3) and hence the results are very high for the chain geometry.

- There is almost no difference between different algorithms for the chain geometry, whereas the developed algorithms are slightly worse (within 2% difference) for the square geometry. The reason is that there is a limited number of paths in the chain geometry.

5.3.1.4 *Tests with Link Failures*

The last factor that we analyzed was the link failures. To observe the effect of them, we broke 20 percent of the all possible links in the graphs, that we considered for the analysis with the node degree. The effect of the link failures on MST metric is illustrated in Figure 25 and Figure 26.

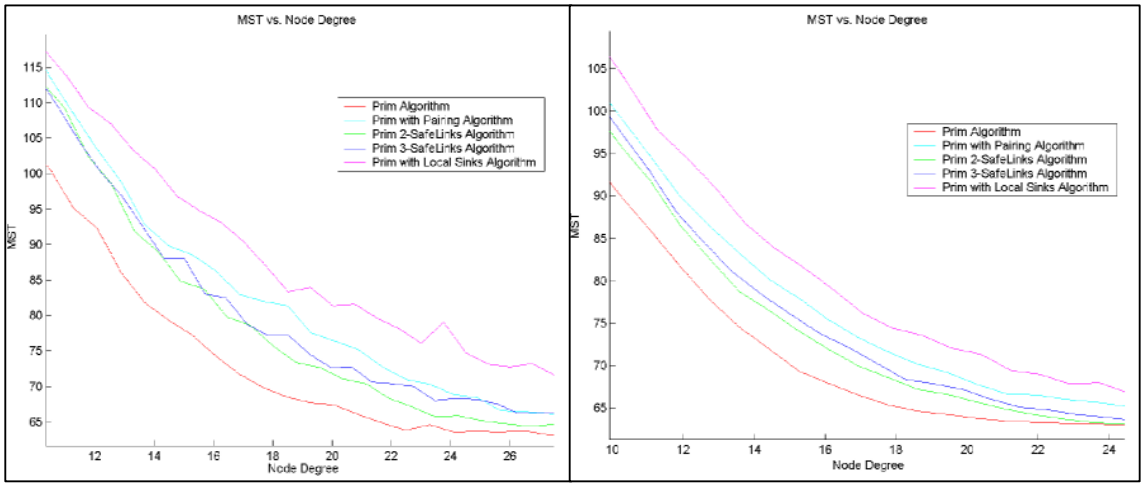


Figure 25: The MST metric versus node degree curves for different algorithms. The plot on the right belongs to the graphs without link breaks whereas the plot on the left belongs to the graphs with 20 % link break. The considered networks have 64 nodes and a square terrain geometry.

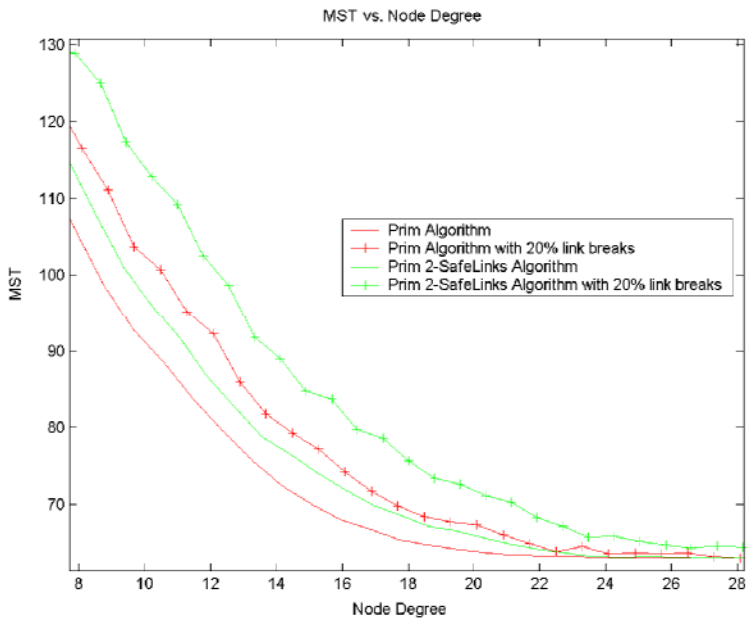


Figure 26: The effect of 20% link breaks on *Prim's Algorithm* and *Prim's 2-SafeLinks Algorithm*, on the networks with 64 nodes and a square terrain geometry.

From the previous 2 figures and the summary given in Table 18 and Table 19, we can conclude the following:

- Breaking 20% of the links in the graph deteriorates the performance of the algorithms about 10%.

- From the tables and the Figure 25 the nominal and absolute difference to the optimal MST metric is higher in case of link breaks. This effect is similar for every developed algorithm, however the impact is slightly larger for the *Prim-Based Local Sinks Algorithm*.

Node Degree	Nominal MST metric [%]					Absolute MST metric				
	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks
10	100	109.9	106.8	108.3	114.3	93.3	102.5	99.6	101	106.6
15	100	111.4	106.0	107.4	116.3	71.3	79.4	75.6	76.6	82.9
20	100	106.6	102.8	103.9	111.9	64	68.2	65.8	66.5	71.6

Table 18: The comparison of the absolute and nominal MST metric for the developed algorithms in case there is no link break.

Node Degree	Nominal MST metric [%]					Absolute MST metric				
	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks	Prim Algorithm	Prim with Pairing	Prim with 2-SafeLinks	Prim with 3-SafeLinks	Prim with Local Sinks
10	100	113.7	109.9	-	115.1	103.3	117.5	113.5	-	118.9
15	100	114.7	107.9	111.1	123.1	78.4	89.9	84.6	87.1	96.5
20	100	114.7	106.6	110.2	122.9	66.8	76.6	71.2	73.6	82.1

Table 19: The comparison of the absolute and nominal MST metric for the developed algorithms in case there is 20% link break. As node degree, the value without link break is used to illustrate the deterioration in the value of the metric.

The effect of the link failures on SPT metric is illustrated in Figure 27 and Figure 28.

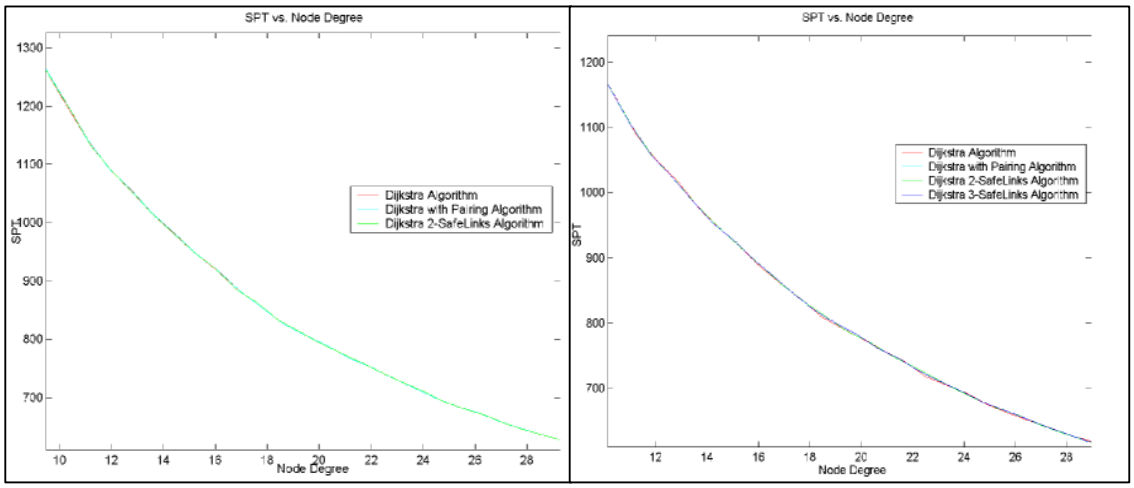


Figure 27: The MST metric versus node degree curves for different algorithms. The plot on the right belongs to the graphs without link breaks whereas the plot on the left belongs to the graphs with 20 % link break. The considered networks have 64 nodes and a square terrain geometry.

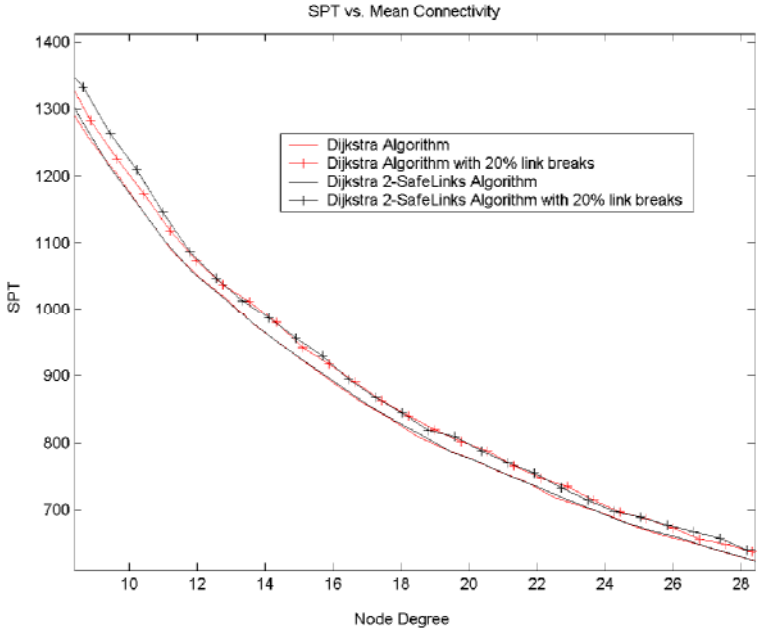


Figure 28: The effect of 20% link break on *Dijkstra's Algorithm* and *Dijkstra's 2-SafeLinks Algorithm* on the networks with 64 nodes and a square terrain geometry.

We observe from the previous 2 figures and the following tables, that the SPT metric deteriorates less than 2% and the difference between developed *Dijkstra-Based* algorithms and the optimum *Dijkstra's Algorithm* remains almost the same.

Node Degree	Nominal SPT metric [%]			Nominal SPT metric [%]		
	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks
10	100	101.8	101.8	1175.7	1197	1197.4

15	100	100.2	100.2	934	935.6	935.7
20	100	100.8	100.8	775.3	781.2	781.3

Table 20: The comparison of the absolute and nominal MST metric for the developed algorithms in case there is no link break. The analyzed networks has 64 nodes.

Node Degree	Nominal SPT metric [%]			Nominal SPT metric [%]		
	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks	Dijkstra Algorithm	Dijkstra with Pairing	Dijkstra with 2-SafeLinks
10	100	100.3	101.4	1204.9	1208.9	1222.1
15	100	100.8	101.1	949.1	956.5	959.2
20	100	100.5	100.1	793.4	797.4	794.2

Table 21: The comparison of the absolute and nominal MST metric for the developed algorithms in case there is 20% link break. As node degree, the value without link break is used to illustrate the deterioration in metric. The analyzed networks has 64 nodes.

5.3.2 Convergence

The second measure that we examined to compare the developed algorithms with the optimum algorithms was *convergence*. A routing algorithm converges, if for a given network the algorithm terminates and finds a routing graph. We analyzed the convergence behaviors of different algorithms for given node degree and specific geometry. Doing this we showed how strong are the effects of the limitations with developed algorithms on the convergence.

5.3.2.1 *Effect of Node Degree*

Node degree, i.e., the average link per node is a crucial factor for the convergence behavior as it is a very good indicator of the connectivity of the nodes in a graph. The higher the node degree of a graph is, the more options the routing algorithms have and the higher the probability of the convergence will be.

To analyze the effect of the node degree on the convergence behaviors, different algorithms are repeatedly run on random graphs with varying node degrees. For each network, we considered a square terrain. The terrains are divided into equal square regions. Within every region, one node is located randomly. The considered network had 64 nodes and by varying the size of the terrain dimensions (preserving the square geometry) we obtained graphs with different node degrees. We discretized the node degree in levels (with 0.25 steps) and measured the probability of convergence for different algorithms based on the simulation results, e.g., if for node degree = 7.5 the algorithm A finds a routing tree in 930 of 1000 total cases we say the probability of the convergence for the algorithm A at node degree 7.5 is 930/1000 = 93%.

To compare the standard *Dijkstra's* and *Prim's Algorithms* with the developed *2-SafeLinks* and *Local Sinks Algorithms*, the standard algorithms were run twice for the same network. If the algorithm converges in the first trial, then one link per node is broken and the algorithm re-routes the network again. If it terminates again, then the test is considered successful otherwise the algorithm fails the test. The motive for this test is fairness with other algorithms as the *2-SafeLinks* and *Local Sinks* algorithm handle one link break per node.

The results of the simulations are illustrated below.

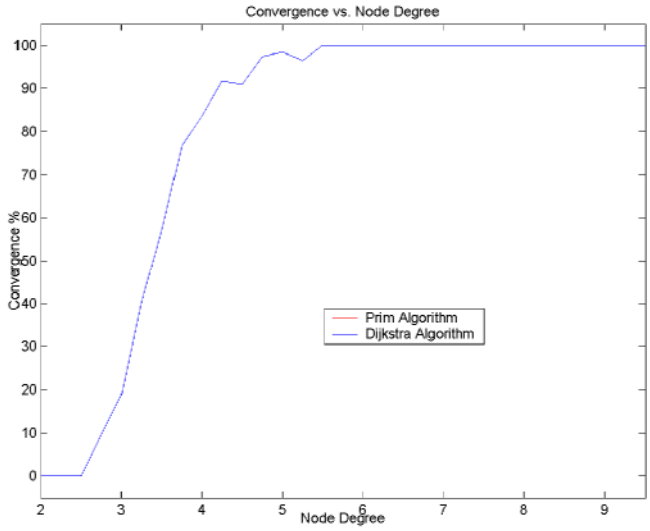


Figure 29: Convergence behavior of the *Prim's* and *Dijkstra's Algorithms* at different node degree levels. The considered networks have 64 nodes and a square terrain geometry. The *Dijkstra's Algorithm* overlaps with the *Prim's Algorithm* because both algorithms need the same condition for convergence: the given graph must be connected.

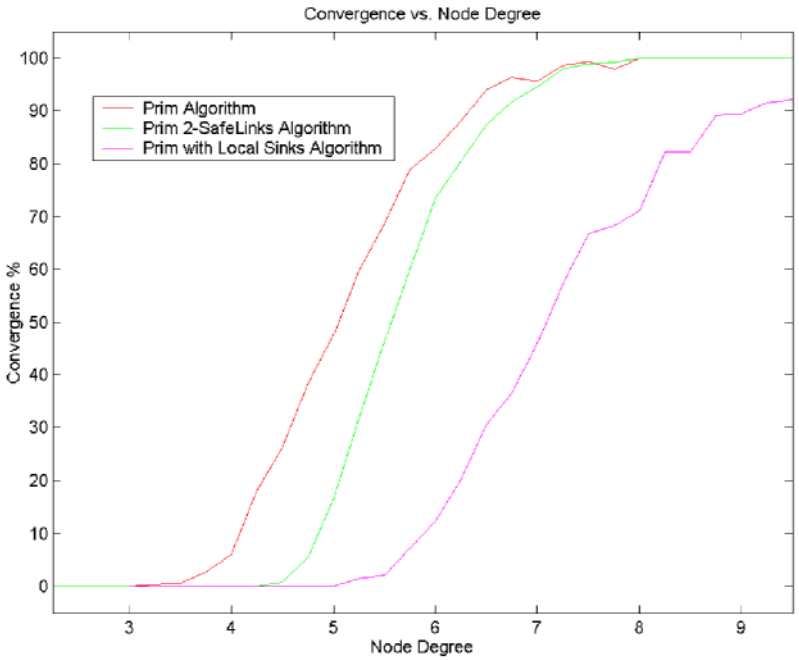


Figure 30: Convergence behavior of the *Prim's Algorithm*, *Prim-Based 2-SafeLinks Algorithms* and *Prim-Based Local Sinks Algorithm* at different node degree levels. The considered networks have 64 nodes and a square terrain geometry. *Prim-Based Local Sinks* and *2-SafeLinks Algorithms* have one alternative parent in case of a link break. To make the comparison between these algorithms and *Prim's Algorithm* fair, one link per node is broken for *Prim's Algorithm* and if the algorithm can converge again in spite of link breaks, we regarded the test as successful otherwise the test is regarded as unsuccessful.

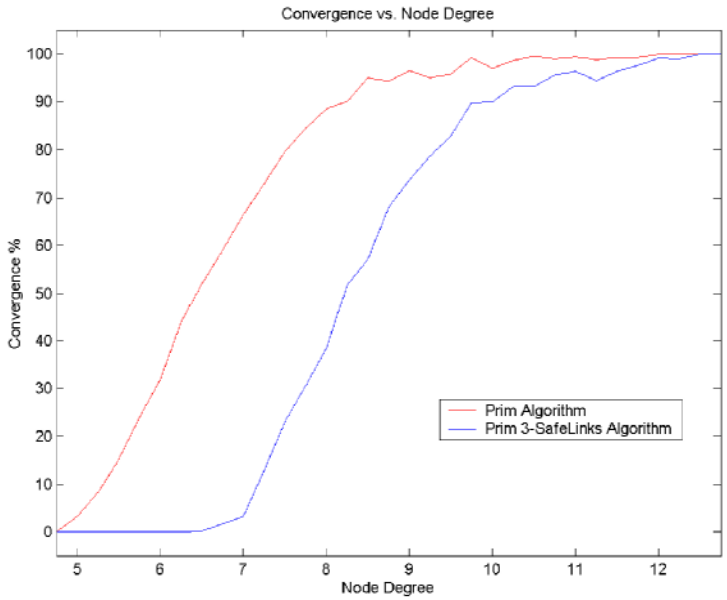


Figure 31: Convergence behavior of the *Prim's Algorithm*, *Prim-Based 3-SafeLinks Algorithms* at different node degree levels. The considered networks have 64 nodes and a square terrain geometry. *Prim-Based 3-SafeLinks Algorithms* have 2 alternative parents in case of link breaks. To make the comparison between the algorithms fair, two links per node are broken for *Prim's Algorithm* and if the algorithm can converge again after re-routing, we regarded the test as successful otherwise the test is regarded as unsuccessful.

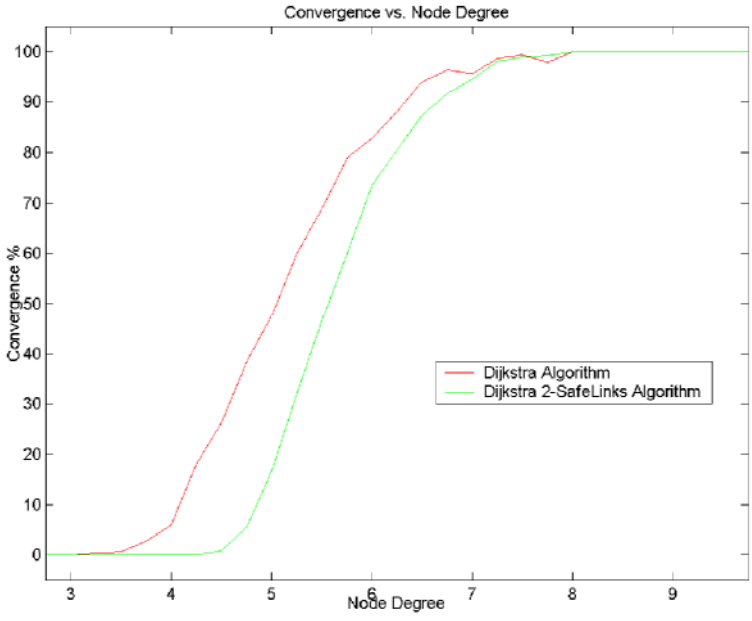


Figure 32: Convergence behavior of the *Dijkstra's Algorithm*, *Dijkstra-Based 2-SafeLinks Algorithms* at different node degree levels. The considered networks have 64 nodes and a square terrain geometry. *Prim-Based 2-SafeLinks Algorithms* have one alternative parent in case of link break. To make the comparison between the algorithms fair, one link per node is broken for *Dijkstra's Algorithm*

and if the algorithm can converge again after re-routing, we regarded the test as successful otherwise the test is regarded as unsuccessful.

Convergence Probability [%]	Prim Algorithm	Prim with 2-SafeLinks Algorithm			Prim with Local Sinks Algorithm		
	Necessary Node Degree	Necessary Node Degree	Absolute Node Degree Difference to Prim	Nominal Node Degree Difference to Prim [%]	Necessary Node Degree	Absolute Node Degree Difference to Prim	Nominal Node Degree Difference to Prim [%]
10	4.1	4.9	0.8	19.5	5.9	1.8	43.9
30	4.6	5.2	0.6	13.0	6.5	1.9	41.3
50	5.1	5.6	0.5	9.8	7.1	2	39.2
70	5.5	6	0.5	9.1	7.9	2.4	43.6
90	6.3	6.7	0.4	6.3	9.1	2.8	44.4
100	7.9	8	0.1	1.3	12	4.1	51.9

Table 22: Convergence probability versus the necessary node degree for different algorithms. The absolute and nominal node degree differences of the developed algorithms to the *Prim's Algorithm* are illustrated. To make the comparison between the algorithms fair, one link per node is broken for *Prim's Algorithm*.

Convvergence Probability [%]	Prim Algorithm	Prim with 3-SafeLinks Algorithm		
	Necessary Node Degree	Necessary Node Degree	Absolute Node Degree Difference to Prim	Nominal Node Degree Difference to Prim [%]
10	5.4	7.2	1.8	33.3
30	5.9	7.7	1.8	30.5
50	6.4	8.2	1.8	28.1
70	7.1	8.8	1.7	23.9
90	8.2	9.9	1.7	20.7
100	11	12.5	1.5	13.6

Table 23: Convergence probability versus the necessary node degree. The absolute and nominal node degree differences of the developed algorithms to the *Prim's Algorithm* are illustrated. To make the comparison between the algorithms fair, two links per node is broken for *Prim's Algorithm* as *3-SafeLinks* has 2 other alternative links per node.

We can make the following observations from the simulation results given in figures 29-32 and summarized in tables 22-23:

- The optimal *Prim's* and *Dijkstra's Algorithms* show the same convergence behavior and they need a node degree higher than 8 for guaranteed (100%) convergence.

- The *Prim's* and *Dijkstra's Algorithms* need the least node degree compared to the developed algorithms for a certain convergence probability. This is an expected result as *Prim's* and *Dijkstra's Algorithms* need a re-routing while *N-SafeLinks* and *Local Sinks Algorithms* simply change to their alternative parents in case of link breaks.

- The absolute and nominal difference between the node degrees for *Prim's and Prim-Based 2-SafeLinks Algorithms* decreases continuously for increasing desired convergence probability, e.g., the nominal node degree difference is approximately 20% for a desired convergence probability $P_{Convergence}$=10% where it decreases to less than 2% for guaranteed convergence ($P_{Convergence}$=100%).

- The absolute and nominal difference between the node degrees for *Prim's and Prim-Based 3-SafeLinks Algorithms* are larger than the *Prim's* and *Prim-Based 2-Safe Links* algorithms. Here we see that as *N* gets larger in the *N-SafeLinks Algorithm*, the convergence behavior gets worse because of the extra limitations because a node has to have *N* possible links to setup a connection.

- Although there are differences in the convergence behavior between the developed algorithms and the standard algorithms (up to 0.5-1 node degree for *2-SafeLinks* and 2-2.5 node degree for the *Local Sinks Algorithm*), these differences are in a certain node degree region. From the previous figures, this region is between node degree=5 and node degree=9. However, for networks that have a higher node degree, all algorithms performs equally well. The Table 24 shows that for the practical examples, the node degree is beyond the critical region and for practical examples, the developed algorithms do not bring any limitations

Scenerio	Max Connectivity	Min Connectivity	Node Degree
Bundeshaus	13	3	9.17
Bundeshaus2Floor	17	4	11.83
GartenStadt	23	7	16.72
GartenStadt2Floor	34	8	20.89
10x10 Walls	12	5	10.06

Table 24: The connectivity statistics of the scenarios provided by SBT. The threshold for the receive power is set to -113 dBm.

5.3.2.2 *Effect of Terrain Geometry*

To analyze the effect of the terrain geometry on the convergence behavior of different algorithms, we considered networks with 64 nodes. For these networks, two types of terrains were studied. The first one was a square terrain, which is divided 8x8 squares. In each of these squares, one node was located randomly. The sinks is in the lower left region (see Figure 23). The other terrain was a node chain with a width of 2 and length of 32 nodes. Again the nodes are located randomly on each of the 64 equally large squares (see Figure 24). The following figures compares the convergence results of the networks with square terrain geometry and chain terrain geometry.

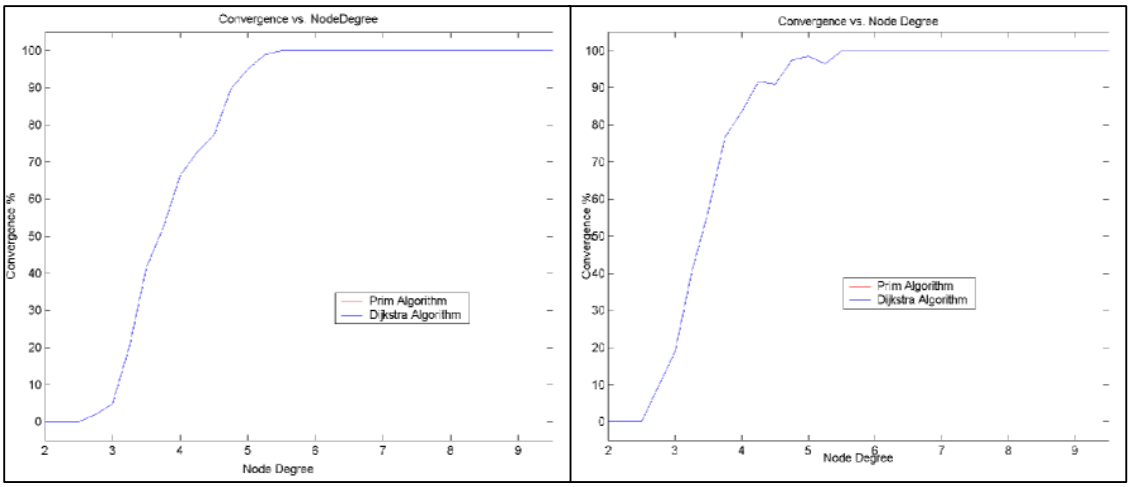


Figure 33: Convergence behavior of the *Dijkstra's* and *Prim's Algorithms* with different node degree levels. On the left, the simulation results for the terrain with chain geometry and on the right there are the results for the square geometry. The tests are made with randomly generated graph with 64 nodes.

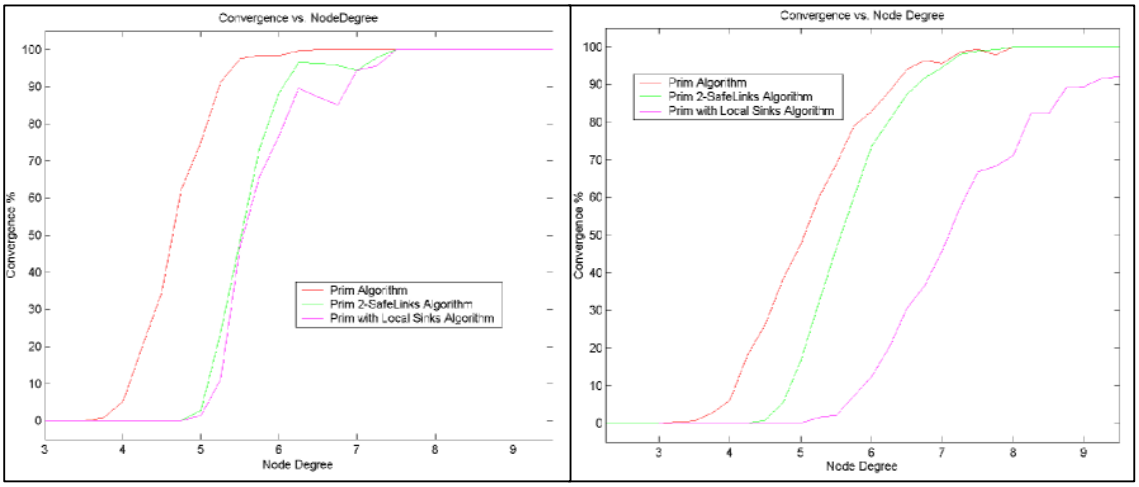


Figure 34: Convergence behavior of the *Prim's Algorithm*, *Prim-Based 2-SafeLinks Algorithm* and *Prim-Based Local Sinks Algorithm* with different node degree levels. On the left, there are the simulation results for the terrain with chain geometry and on the right there are the results for the square terrain geometry. The tests are made with randomly generated graphs with 64 nodes.

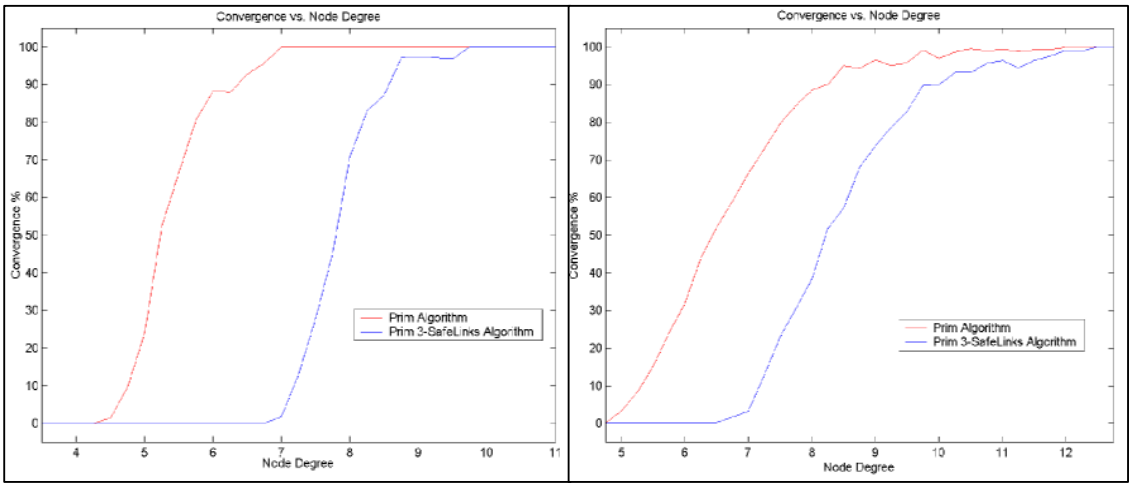


Figure 35: Convergence behavior of the *Prim's Algorithms* and *Prim-Based 3-SafeLinks Algorithm* with different node degree levels. On the left, there are the simulation results for the terrain with the 2x32 chain geometry and on the right there are the results for the square terrain geometry. The tests are made with randomly generated graphs with 64 nodes.

From the figure 32-35 and following tables, we can draw the following observations:

- The most significant difference between two results is following: The *Prim-Based Local Sinks Algorithm* that needed 20-25% higher node degree than *2-SafeLinks* for a given convergence probability in the square terrain, performs almost the same as *2-SafeLinks Algorithm* for the chain geometry. The main reason is that the *Local Sinks Algorithm* a node must have at least 2 links to the subtree it is supposed to be added. Other possible links are not counted. In a chain-geometry, subtrees are added successively on a line and hence *Local Sinks Algorithm* does not introduce extra limitations compared to *2-SafeLinks Algorithm*. However for the square geometry, the subtrees grow in a parallel fashion and because the possible links of a node are shared by different subtrees, the convergence behavior deteriorates.

- For the chain geometry, the graphs for convergence behavior have a steeper transition from zero probability to 100% convergence probability. This result can be expected as in a chain geometry, there are less possibility for the node distribution and hence there is a more precise convergence threshold in terms of node degree (this threshold is around 4.5 for *Prim's Algorithm*, 5.5 for *Prim Based 2-SafeLinks and Local Sinks*, 7.5 for *Prim-Based 3-SafeLinks Algorithm*).

- The last observation is that the difference between *Prim's Algorithm* and *Prim-Based 2-Safe Links* and *3-SafeLinks Algorithms* increases with chain geometry. A reason for this situation is that in the intermediate steps of the *N-SafeLinks Algorithms* there are not many choices for possible links to the current partial spanning tree for the chain geometry. On the other hand *Prim's Algorithm* can find other possible ways with re-routing.

Convergence Probability [%]	Prim Algorithm	Prim with 2-SafeLinks Algorithm			Prim with Local Sinks Algorithm		
	Necessary Node Degree	Necessary Node Degree	Absolute Node Degree Difference to Prim	Percentage Node Degree Difference to Prim [%]	Necessary Node Degree	Absolute Node Degree Difference to Prim	Percentage Node Degree Difference to Prim
10	4.1	5.1	1	24.4	5.2	1.1	26.8
30	4.4	5.3	0.9	20.5	5.4	1	22.7
50	4.6	5.5	0.9	19.6	5.5	0.9	19.6
70	4.9	5.7	0.8	16.3	5.8	0.9	18.4
90	5.2	6	0.8	15.4	6.5	1.3	25.0
100	6.3	7.2	0.9	14.3	7.5	1.2	19.0

Table 25: Convergence probability versus the necessary node degree for *Prim's* and *Prim-Based 2-SafeLinks Algorithm* and *Prim-Based Local Sinks Algorithm*. For the *Prim's Algorithm* , one link per node is broken. For the simulations the network with 64 nodes and a terrain with chain geometry is used

Conververgence Probability [%]	Prim Algorithm	Prim with 3-SafeLinks Algorithm		
	Necessary Node Degree	Necessary Node Degree	Absolute Node Degree Difference to Prim	Percentage Node Degree Difference to Prim [%]
10	4.8	7.2	2.4	50.0
30	5.1	7.5	2.4	47.1
50	5.2	7.8	2.6	50.0
70	5.6	8	2.4	42.9
90	6.3	8.5	2.2	34.9
100	7	9.4	2.4	34.3

Table 26: Convergence probability versus the necessary node degree for *Prim's* and *Prim-Based 3-SafeLinks Algorithm*. For the *Prim's Algorithm*, two links per node is broken as the *Prim-Based 3-Safe Links Algorithm* has 2 alternative links. For the simulations the network with 64 nodes and a terrain with chain geometry is use

5.3.2.3 *Maximum Subtree-Hopcount*

Another factor that we analyzed is the maximum subtree hopcount $h_{\max}^{subtree}$, i.e., the maximum allowed hops of a subtree for the *Prim-Based Local Sinks Algorithm*. The simulation results are illustrated in Figure 36.

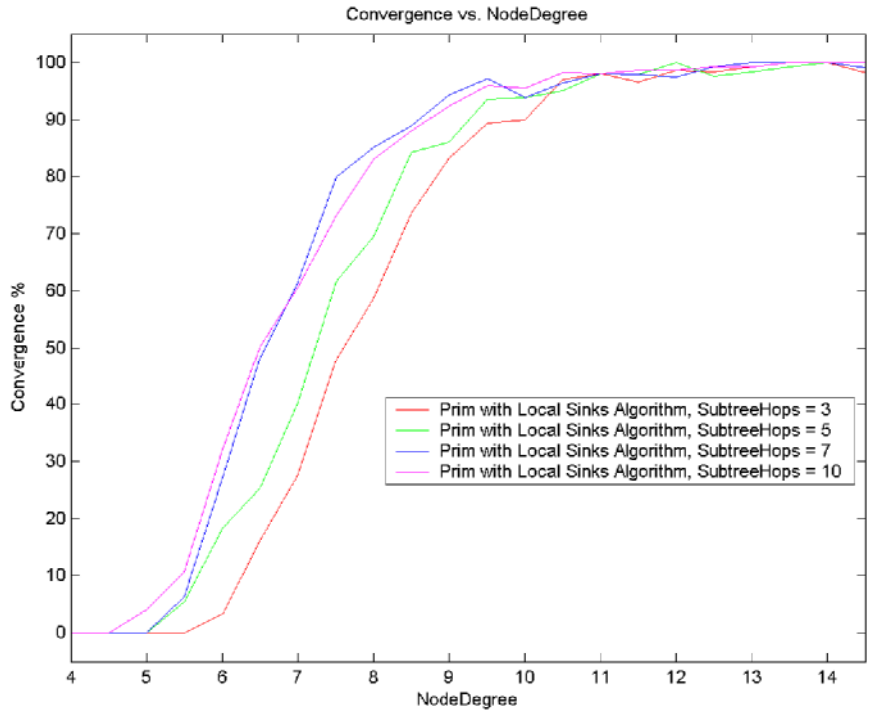


Figure 36: Convergence behaviors of the *Prim-Based Local Sinks Algorithm* with different $h_{\max}^{subtree}$. The tests are made with randomly generated graph with 64 nodes in a square terrain.

Convergence Probability [%]	Prim Algorithm	$h_{\max}^{subtree}$ = 3		$h_{\max}^{subtree}$ = 5		$h_{\max}^{subtree}$ = 7		$h_{\max}^{subtree}$ = 10	
	Necessary Node Degree	Absolute Node Degree Difference to Prim	Percentage Node Degree Difference to Prim [%]	Absolute Node Degree Difference to Prim	Percentage Node Degree Difference to Prim [%]	Absolute Node Degree Difference to Prim	Percentage Node Degree Difference to Prim [%]	Absolute Node Degree Difference to Prim	Percentage Node Degree Difference to Prim [%]
10	4.1	2.1	51.2	1.6	39.0	1.5	36.6	1.4	34.1
30	4.6	2.5	54.3	2.1	45.7	1.5	32.6	1.4	30.4
50	5.1	2.5	49.0	2.1	41.2	1.5	29.4	1.4	27.5
70	5.5	2.9	52.7	2.6	47.3	1.7	30.9	1.9	34.5
90	6.3	3.7	58.7	3	47.6	2.3	36.5	2.4	38.1

Table 27: Convergence probability versus the necessary node degree, for *Prim-Based Local Sinks Algorithm* with various maximum subtree hopcounts. The results are expressed based on the necessary node degree for the *Prim's Algorithm*, i.e., the nominal and absolute additional node degree over the needed node degree for *Prim's Algorithm*.

From Figure 36 and Table 27, we can draw following conclusions:

- With decreasing number of the maximum allowed subtree hops, the convergence behavior of the *Local Sinks Algorithm* gets worse, e.g., the needed node degree for

90% convergence is 10 for $h_{\max}^{subtree}$ =3 while it is 8.7 for $h_{\max}^{subtree}$ = 10. The reason is that for lower maximum subtree hopcounts, more local sinks and subtrees must be generated. Hence the possible links of a node can be shared by different subtrees and this can prevent the nodes to make links because a node must have at least 2 possible connections to the subtree it will be attached (see section 4.3 for the details of the *Local Sinks Algorithm*)

- The difference between convergence behaviors diminishes for maximum allowed subtree hopcount over 7, whereas the convergence performance obviously deteriorates if $h_{\max}^{subtree}$ is selected less than 5.

5.3.3 Power Consumption

In this section, we analyze the power consumption of the CWSN application for different algorithms. We assume that we utilize the double tree concept, i.e., one tree that is generated by *Prim's Algorithm* or a *Prim-Based Algorithm* will be used for monitoring and another tree that is generated by *Dijkstra's Algorithm* or a *Dijkstra-Based Algorithm* will be used for the alarm messages. The parameter dependencies and constraints are given in section 4.4 . However in that section, all the calculations are based on the assumption that local supervising (*Local Sinks*) concept is utilized for monitoring tree. For the algorithms that do not use the *Local Sinks Concept*, the equation (5) becomes

$$\Delta_M = \tau_{Fail} - (h_{\max}^{mon} + N_{retries}^{mon})\Delta_h \quad , \tag{16}$$

because a node crash can only be detected by the sink node. Due to this reason, we simulated the power consumptions for sampling and monitoring events with different wake-up intervals T_w and determined the value T_w that minimize the total power. (see Figure 37).

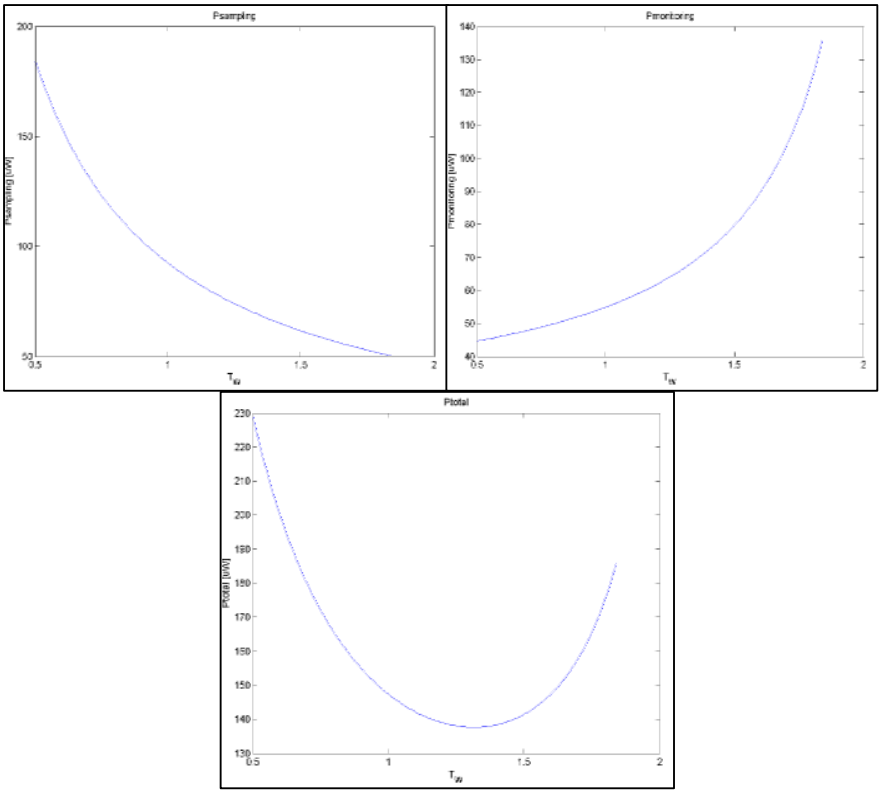


Figure 37: The consumed sampling power $P_{Sampling}$, monitoring power P_{Mon} and total power P_{tot} for different wake-up intervals T_w. The power for sampling is inversely proportional with the T_w (on upper left), whereas the power needed for monitoring increases with larger T_w. The total power, which is a sum of these two parts, have a minima in terms of T_w. With the simulations we found the total power minimizing wake-up interval numerically.

The calculations are made for the normal operation where there is the periodic traffic of the monitoring messages. We assume that there are no fire events and no link failures. The considered networks are the following:

1. Randomly generated network with 64 nodes in a square terrain with edge length = 140m. The node degree of the graph is 9.5.

2. Randomly generated network with 100 nodes in a square terrain with edge length = 180m. The node degree of the graph is 9.4.

3. "Bundeshaus Scenario" provided by SBT. The network has 41 nodes and a node degree of 9.2.

4. "Bundeshaus with 2 Floors Scenario" provided by SBT. The network has 81 nodes and a node degree of 11.7.

For each network we consider the following algorithm pairs for monitoring and alarm trees:

1. *Prim's Algorithm* for the monitoring tree and *Dijkstra's Algorithm* for the alarm tree. The local supervising concept cannot be used for this pair.

2. *Prim-Based 2-SafeLinks Algorithm* is used for the monitoring tree and *Dijkstra-Based 2-SafeLinks Algorithm* is used for the alarm tree. The local supervising concept cannot be used for this pair.

3. *Prim-Based Local Sinks Algorithm* is used for the monitoring tree and *Dijkstra-Based 2-SafeLinks Algorithm* is used for the alarm tree. The maximum allowed subtree hopcount for the *Local Sinks Algorithm* is 10.

4. *Prim-Based Local Sinks Algorithm* is used for the monitoring tree and *Dijkstra-Based 2-SafeLinks Algorithm* is used for the alarm tree. The maximum allowed subtree hopcount for the *Local Sinks Algorithm* is 5.

For the power calculations, we assume that the clock drift is θ=30ppm and messages are sent only with OK-BIT, i.e., the data packet only contains one bit for the present messages and its size is 13 bytes. The total power is the sum of the power for sampling $P_{Sampling}$, power for monitoring P_{Mon} and link testing P_{Link}. As the link testing part is very sensitive to clock drift and it depends strongly on the frequency scheme as well as the MAC layer parameters, we analyze only the total power that is the sum of $P_{Sampling}$ and P_{Mon}, i.e.,

$$P_{tot} = P_{Sampling} + P_{Mon} \quad .$$

Next, we illustrate the simulation results for the power consumption on the four considered networks. For every network, the four algorithm pairs for monitoring and alarm tree are simulated.

5.3.3.1 *Random Graph with 64 Nodes*

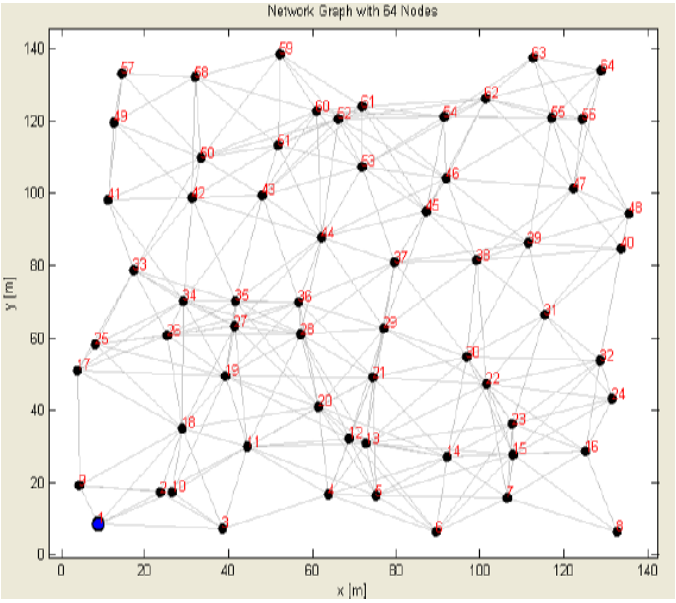


Figure 38: The randomly generated network with 64 nodes and 140m x140m square terrain. The grey lines are the possible links. The minimum connectivity for the network is 3 while the maximum connectivity is 14 and the node degree is 9.5

	K_{Mon}	K_{Alarm}	$h_{\max}^{mon}$	$h_{\max}^{alarm}$	$h_{\max}^{subtree}$	Δ_h [s]	Δ_M [s]	T_w [s]	$P_{Sampling}$ [μW]	P_{Mon} [μW]			P_{tot} [μW]		
										max	min	mean	max	min	mean
1	3	5	19	6	-	9.2	116.0	1.15	80.4	69.5	21.6	37.2	149.9	101.9	117.6
2	3	5	17	6	-	10.1	118.6	1.26	73.3	68.4	21.3	37.0	141.8	94.6	110.3
3	2	5	18	6	10	11.4	164.3	1.43	64.7	42.6	17.9	30.2	107.2	82.6	94.9
4	2	5	17	6	5	11.4	221.4	1.43	64.7	35.8	15.6	25.7	100.4	80.3	90.4

Table 28: The CWSN parameters (Δ_M, Δ_h and T_w), parameters of the graphs (K_{Mon}, K_{Alarm}, $h_{\max}^{mon}$, $h_{\max}^{alarm}$ and $h_{\max}^{subtree}$) and the resulting monitoring, sampling as well as the total power consumption per node for 4 different algorithm pairs. The first row refers to a monitoring tree with *Prim's Algorithm* and an alarm tree with *Dijkstra's Algorithm*. The second row refers to *Prim-Based 2-SafeLinks Algorithm* and *Dijkstra-Based 2-SafeLinks Algorithm* pair. The third and fourth rows refer to *Prim-Based Local Sinks Algorithm* and *Dijkstra-Based 2-SafeLinks Algorithm* pair with different $h_{\max}^{subtree}$.

5.3.3.2 *Random Graph with 100 Nodes*

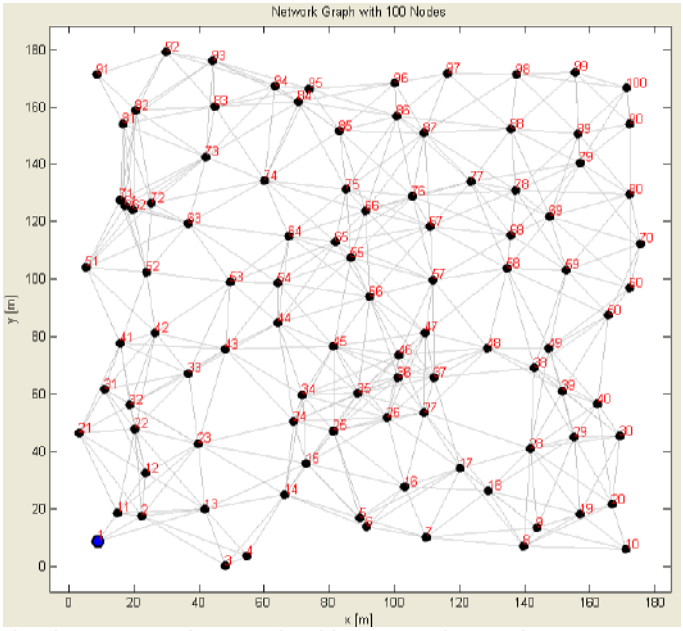


Figure 39: The randomly generated network with 100 nodes and 180m x180m square terrain. The grey lines are the possible links. The minimum connectivity for the network is 3 while maximum connectivity is 14 and the node degree is 9.4

	K_{Mon}	K_{Alarm}	$h_{\max}^{mon}$	$h_{\max}^{alarm}$	$h_{\max}^{subtree}$	Δ_h [s]	Δ_M [s]	T_w [s]	$P_{Sampling}$ [μW]	P_{Mon} [μW]			P_{tot} [μW]		
										max	min	mean	max	min	mean
1	4	5	33	8	-	5.0	128.6	0.63	146.7	79.4	20.3	35.1	226.1	167.0	181.8
2	3	6	25	8	-	7.4	106.6	0.93	99.4	73.9	22.7	39.7	173.3	122.0	139.1
3	2	6	22	8	10	8.9	192.2	1.11	83.2	38.7	16.6	27.7	121.9	99.8	110.8
4	2	6	22	8	5	8.9	236.7	1.11	83.2	34.5	15.2	24.8	117.7	98.3	108.0

Table 29: The CWSN parameters (Δ_M, Δ_h and T_w), parameters of the graphs (K_{Mon}, K_{Alarm}, $h_{\max}^{mon}$, $h_{\max}^{alarm}$ and $h_{\max}^{subtree}$) and the resulting monitoring, sampling as well as the total power consumption per node for 4 different algorithm pairs (see section 5.3.3 for the description of these algorithm pairs). The simulations are carried out for the network with 100 nodes (see Figure 39).

5.3.3.3 *Bundeshaus Scenario from SBT*

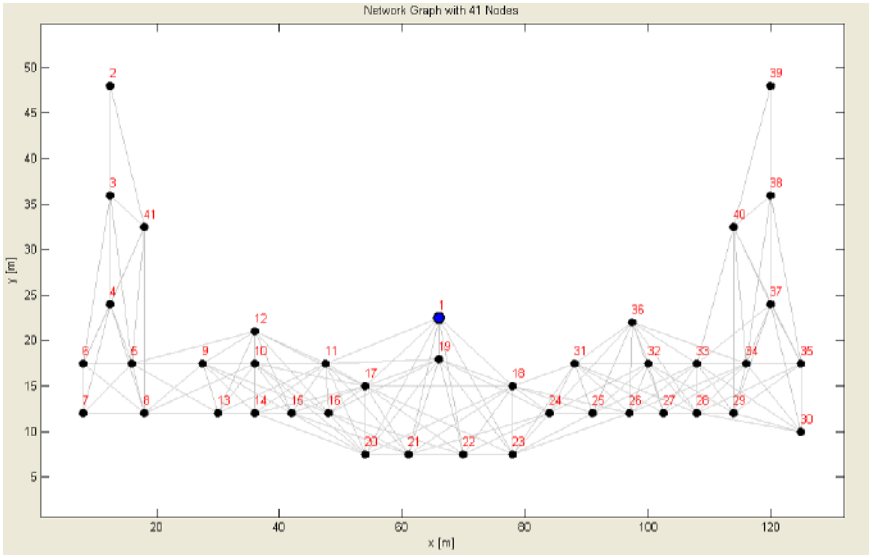


Figure 40: The network of Bundeshaus scenario provided from SBT. The network has 41 nodes including one sink node. The grey lines are the possible links. The minimum connectivity for the network is 3 while maximum connectivity is 14 and the node degree is 9.4.

	K_{Mon}	K_{Alarm}	$h_{\max}^{mon}$	$h_{\max}^{alarm}$	$h_{\max}^{subtree}$	Δ_h [s]	Δ_M [s]	T_w [s]	$P_{Sampling}$ [μW]	P_{Mon} [μW]			P_{tot} [μW]		
										max	min	mean	max	min	mean
1	3	5	15	5	-	11.0	123.4	1.38	67.0	66.5	20.8	36.0	133.5	87.8	103.0
2	2	5	16	5	-	11.1	111.0	1.39	66.5	55.2	22.1	38.7	121.7	88.6	105.1
3	2	5	16	5	10	13.3	143.3	1.67	55.5	46.4	19.2	32.8	101.9	74.6	88.2
4	2	5	15	5	5	13.3	210.0	1.67	55.5	36.8	16.0	26.4	92.3	71.4	81.8

Table 30: The CWSN parameters (Δ_M, Δ_h and T_w), parameters of the graphs (K_{Mon}, K_{Alarm}, $h_{\max}^{mon}$, $h_{\max}^{alarm}$ and $h_{\max}^{subtree}$) and the resulting monitoring, sampling as well as the total power consumption per node for 4 different algorithm pairs (see section 5.3.3 for the description of these algorithm pairs). The simulations are carried out for the Bundeshaus scenario that has 41 nodes.

5.3.3.4 *Bundeshaus with 2 Floors Scenario from SBT*

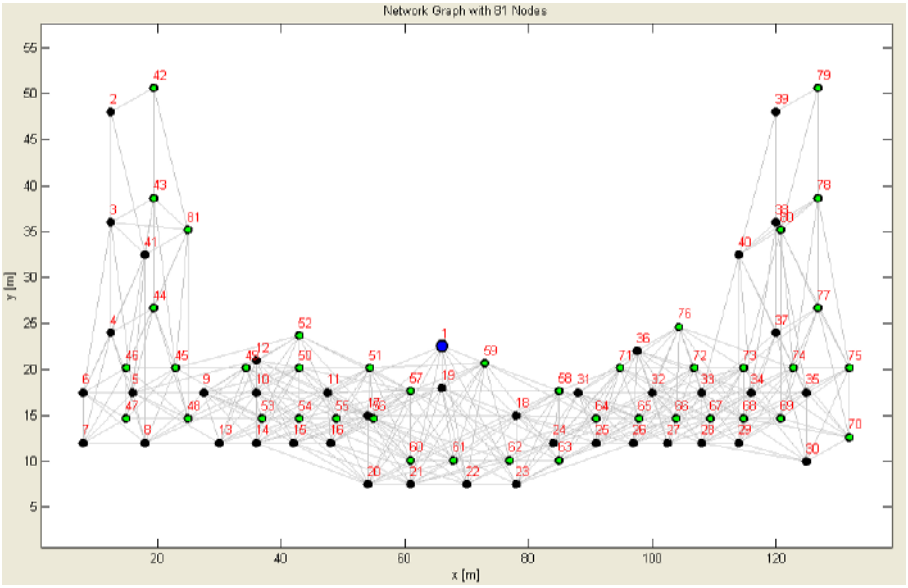


Figure 41: The network of Bundeshaus with 2 floors scenario provided from SBT. The network has 81 nodes including one sink node. The nodes in the second floor are in green. The minimum connectivity for the network is 3 while maximum connectivity is 14 and the node degree is 9.4

	K_{Mon}	K_{Alarm}	$h_{\max}^{mon}$	$h_{\max}^{alarm}$	$h_{\max}^{subtree}$	Δ_h [s]	Δ_M [s]	T_w [s]	$P_{Sampling}$ [μW]	P_{Mon} [μW]			P_{tot} [μW]		
										max	min	mean	max	min	mean
1	4	7	29	6	-	6.3	110.4	0.79	117.0	88.6	22.2	38.8	205.6	139.2	155.8
2	2	6	16	6	-	11.1	111.0	1.39	66.5	55.2	22.1	38.7	121.7	88.6	105.1
3	3	6	18	6	10	11.4	164.3	1.43	64.7	54.9	17.9	30.2	119.6	82.6	94.9
4	2	6	17	6	5	11.4	221.4	1.43	64.7	35.8	15.6	25.7	100.4	80.3	90.4

Table 31: The CWSN parameters (Δ_M, Δ_h and T_w), parameters of the graphs (K_{Mon}, K_{Alarm}, $h_{\max}^{mon}$, $h_{\max}^{alarm}$ and $h_{\max}^{subtree}$) and the resulting monitoring, sampling as well as the total power consumption per node for 4 different algorithm pairs (see section 5.3.3 for the description of these algorithm pairs). The simulations are carried out for the Bundeshaus with 2 floors scenario that has 81 nodes.

5.3.3.5 *Discussion*

In the sections 5.3.3.1 -5.3.3.4 , we considered four different networks and applied four different algorithm pairs to generate monitoring and alarm trees. We presented the total power optimizing CWSN parameters, graph parameters and consumed power results for the analyzed algorithm pairs. For a fixed T_w the sampling power $P_{Sampling}$ is constant for every node. However, the monitoring power P_{Mon} depends on the number of the branches as for a node, the number of acknowledges per monitoring interval is equal to the number of its children. Hence the nodes having the most number children have the maximum P_{Mon} whereas the leaf nodes consume the least monitoring power. This is why we indicated a minimum, maximum and average P_{Mon} and hence P_{tot} in the tables 27-30 where we summarized our findings. From these results , we can draw the following conclusions:

- The first algorithm pair, i.e., *Prim's Algorithm* for the monitoring tree and *Dijkstra's Algorithm* for the alarm tree has the least power efficiency. The most important reason for this weak power efficiency is that this algorithm pair does not utilize the local supervising concept. Therefore, the number of hops in the monitoring tree becomes the limiting factor for the monitoring interval to fulfill the maximum allowed latency for node crashes τ_{Fail}. The decrease in the monitoring interval increases the monitoring power immensely and hence T_w must be selected less than the maximum limit of $\tau_{Fire}/(h_{\max}^{alarm} + N_{retries}^{mon})$ as it was demonstrated in Figure 37. As the original *Prim's Algorithm* has the tendency to make more hops than any other considered algorithm, the explained limitation on the monitoring and wake-up interval are highest and the power efficiency is the worst for the first algorithm pair.

- The second algorithm pair, i.e., *Prim-Based 2-SafeLink Algorithm* for the monitoring tree and *Dijkstra-Based 2-SafeLink Algorithm* for the alarm tree has a slightly better performance than the first algorithm pair but it is clearly inferior to the last algorithm pair, namely *Prim-Based Local Sinks Algorithm* for the monitoring tree and *Dijkstra-Based 2-SafeLinks Algorithm* for the alarm tree. The reason that the second algorithm pair works better than the first one is that *Dijkstra-Based 2-SafeLinks Algorithm* does not make more hops than the original *Dijkstra's Algorithm* and the *Prim-Based 2-SafeLinks Algorithm* has the tendency to make less hops and introduce less limitations on Δ_M and T_w compared to the standard *Prim's Algorithm*. However, the fact that also this algorithm pair does not use the *Local Sinks Concept*, the explained limitations on Δ_M and T_w still exist and the power efficiency is significantly lower than the one of the last algorithm pair.

- The last algorithm pair, i.e., the *Prim-Based Local Sinks Algorithm* for the monitoring tree and the *Dijkstra-Based 2-SafeLinks Algorithm* for the alarm tree has the best power performance compared to all the other algorithm pairs. The reason for this high power efficiency is that the hops of the monitoring tree do not limit the application parameters. The node crashes can be detected by the local sinks and their respective technical alarms can be sent to the global sink over the alarm tree within τ_{Fire}. Therefore the power calculations in section 4.4 applies here. The limiting factor for this last algorithm pair is $h_{\max}^{subtree}$ and its effect can be noticed by comparing the last 2 rows in the tables. The algorithm using $h_{\max}^{subtree}$=5 provides up to 10-15% better power efficiency than the algorithm with $h_{\max}^{subtree}$=10 as with less $h_{\max}^{subtree}$, Δ_M can be selected higher for a lower monitoring power.

- Using the *Prim-Based Local Sinks Algorithm* for the monitoring tree and the *Dijkstra-Based 2-SafeLinks Algorithm* for the alarm tree, the sum of the sampling and maximum monitoring power can be reduced to 100 μW or even further (see Table 30 and Table 31).

- The limiting factor is the number of maximum branches K_{Mon} and the maximum number of hops in the alarm tree $h_{\max}^{alarm}$. While an increment in the number of branches raises P_{Mon} linearly, every additional hop in the alarm tree lowers the sampling

interval T_w further and increases $P_{Sampling}$. The effect of the maximum branches can be seen from the last two rows of the Table 31 whereas the effect of $h_{\max}^{alarm}$ is exemplified by Table 28 and Table 29. Assuming that $N_{retries}^{alarm}$=1, for a worst-case total power of approximately P_{tot}=100 μW, $h_{\max}^{alarm}$ must not exceed 6 and K_{Mon} cannot be larger than 3. If the topology does not allow the maximum alarm hops $h_{\max}^{alarm}$ to be less than 9 or 10, the sampling power is alone above 100 μW and the aimed limit is already exceeded. Hence a topology cannot have too many nodes and it should provide the connectivity to keep the $h_{\max}^{alarm}$ low.

- For the given battery models in section 4.4.8 , a network lifetime over 5 years can be estimated if the fourth algorithm pair is used as the power consumption did not exceed 120 μW in any test. For networks with small number of nodes, the first and second algorithm pairs can be expected to have a lifetime over 3 years. However for the scenarios having more nodes (network with 100 nodes, Bundeshaus with 2 floors), these algorithm can have pick power consumption values over 200 μW, which introduce the risk that a network lifetime over 3 years cannot be reached, especially if the environment is exposed to temperature fluctuations (see Table 7).

- The number of nodes and the topology set the values $h_{\max}^{alarm}$ and K_{Mon}. From the tests with random nodes, we conclude that with one sink that is located in the corner of the network, 80 or up to 100 nodes can be covered where a total power consumption near 100 μW can be reached. Placing the nodes intelligently and locating the sink in the middle of the terrain, more nodes can be covered without exceeding the power limits.

5.4 Summary

5.4.1 Overview on the Simulation Results

From the simulation results and their respective observations, we can reach the following important points:

- With the link weights, calculated from the RSSI-mappings (13) and (15) , the proposed algorithm performs very close to the optimal MST and SPT algorithms in terms of the metric value. (For MST the proposed algorithms do not exceed 20%, where for SPT the deviation from the optimum is limited to 2%). The link breaks, network size and terrain geometry do not significantly change or deteriorate the relative performance of these algorithms.

- The proposed algorithms do not bring significant limitations on the topology for the convergence. As we have seen in section 5.3.2 , in the critical region the *Local Sinks Algorithm* needs up to 2.5 and the *2-SafeLinks Algorithm* needs up to 1 possible link more than the standard algorithms. However, these differences shrink for linear geometries (which are more common geometries inside the buildings, e.g., corridors). Moreover, the node degree of the practical examples are beyond the critical region, i.e., node degrees between 5 and 9 (see Table 24).

- At last, using the new developed algorithms, power consumption can be reduced greatly. The *Prim-Based 2-SafeLinks Algorithm* has the tendency to make less brunches K_{Mon} and less hops $h_{\max}^{mon}$ than the standard *Prim's Algorithm*. Hence it is more power efficient. The tendency is inherited also by the *Local Sinks Algorithm* as it is derived from the *Prim-Based 2-SafeLinks Algorithm*. The *Local Sinks Algorithm* reduces the power consumption further due to local supervising concept.

5.4.2 Suitability of Algorithms for CWSN Application

From the previous sections we have seen that the developed algorithms do not introduce significant limitations, e.g., on convergence or MST and SPT metrics. On the other hand, for the CWSN application they are proved to be more power-efficient than the standard algorithms. The power analysis in section 4.4 and the simulation results from section 5.3.3 showed that using the *Prim-Based Local Sinks Algorithm* for the monitoring tree and the *Dijkstra-Based 2-SafeLinks Algorithm* for the alarm tree provides significant reduction (up to 40%) in the power consumption. At last, they can handle up to one link break per node without the need of any re-routing. Hence we have verified that this combination is the most suitable selection in the CWSN implementation.

6. GloMoSim Simulation of the CWSN Application

6.1 The Network Simulator: GloMoSim

GloMoSim (Global Mobile System Simulator) was used as the simulation platform for the CWSN application because of the structured protocol layers and easy developing facilities [17] . The data flow between the layers are illustrated in Figure 42.

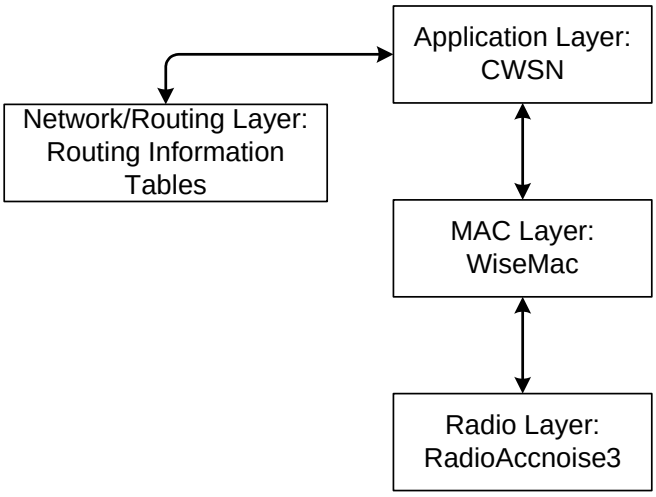


Figure 42: The block diagram of the layers in the implementation. The routing tables are initially read by the CWSN application from Network/Routing Layer and during the normal operation, CWSN application exchanges information with MAC Layer, which provides the connection from the application layer to the radio layer.

6.2 CWSN Application

The CWSN application is a fire detection wireless network (FD-WSN) application whose requirements and functions are explained in section 1.2 and [1]. We have implemented the CWSN application in GloMoSim on two different levels, network and application layers. The MAC and Radio layers were provided by CSEM[19] [4], [5]. In the next two section, we will explain these developed layers in short. For the details of the implementation, the reader can refer to [18].

6.3 Network Layer

The routing information is provided by the network layer to the CWSN application. The routing information of a node contains the main and alternative parents (A_1, A_2) as well as the main and alternative monitoring parents (M_1, M_2). It also has additional information about the number of hops and the quality of links of a node to these parents. The routing information is generated off-line by the MATLAB Graph Analyzer Tool (see Section 5.1) and written to the file called, *Routing Information Table.*

[19] CSEM: Centre Suisse d'Electronique et de Microtechnique.

6.3.1 Reading the Routing Information Tables

In order to facilitate our static routing protocol in GloMoSim, the ROUTING-PROTOCOL in the configuration file should be selected as STATIC. The name of the file keeping the *Routing Information Table*, e.g. RoutingInfo.in, should be given in STATIC-ROUTING-INFORMATION-FILE. When a simulation is started, the *Routing Information Table* is read by the respective functions and the necessary routing information is saved into the data structure of every node.

6.3.2 Routing-Information Structure in the Network Layer

The routing information of each node is saved in its *NetworkRoutingInformationTable* structure. Every node has a *NetworkRoutingInformationTable* which is comprised of four rows for each parent (main alarm parent A_1, alternative alarm parent A_2, main monitoring parent M_1 and alternative monitoring parent M_2). The rows are saved in the data structure called *NetworkRoutingInformationTableRow* that contains the node address, the parent address, hop count to the sink, hop count to the local sink, the node type and the RSSI level. Definitions of these two structures are illustrated in Figure 43.

```
typedef struct {
  NODE_ADDR nodeAddress;          /* destination address */
  NODE_ADDR parentAddress;        /* parentaddress */
  NODE_ADDR localSinkAddress;     /* address of the node's local sink*/
  int nodeType;                   /* type of the node */
  int hopCount;                   /* hopCount */
  int relHopCount;                /* hopCount to local sink */
  int rssi;                       /* RSSI value of its link */
} NetworkRoutingInformationTableRow;

typedef struct {
    int size;  /* number of entries */
    int allocatedSize;
    NetworkRoutingInformationTableRow *row;
} NetworkRoutingInformationTable;
```

Figure 43: Definitions of the *NetworkRoutingInformationTable* and *NetworkRoutingInformationTableRow* data structures

6.3.3 Interface with the Application Layer

The information in these structures are read by the application layer in the initialization phase. During the normal operation of CWSN application, these structures can be changed, updated or emptied by the application layer, in case the routing tables need to be altered[20].

6.4 Application Layer

The application layer is responsible for the following functions:

- **Periodic monitoring messaging:** All nodes except the sink send periodic monitoring messages. For each subtree the messages are collected in the local sink of that subtree. The local sinks and the global sink check for the presence of the nodes in their subtrees.

[20] This feature is not used in the current version of the CWSN implementation and the routing tables are fixed during the entire simulation.

- **Technical alarms:** In case of node crashes or communication failures, the nodes that cannot inform the local sink about their presence are reported by the local sink. The local sink initiates in this case a *technical alarm* that is reported over the fast alarm tree to the global sink within the required time τ_{Fire}.
- **Fire alarms:** The nodes detecting a fire event send a fire alarm message over the fast alarm tree to the global sink within the required time τ_{Fire}.
- **Multi-frequency operation**: The application can assign a set of frequencies to each node. Currently this is done once during initialization.
- **Links testing:** If multi-frequency option is selected, the possible links are tested for the synchronization that is needed for the given frequency scheme [19].

The application promises also the following features other than its main functions

- **Robustness**: the described functionality is also provided in case of technical failures such as link and channel failures specified in [1]. The protocol handles one link break per node (can be generalized to more than one failing link).
- **False alarm rate:** For the normal operation, i.e., only monitoring operation without fire events, no false alarms occurred during a simulation time of 10 days.
- **Power optimization:** The developed double-tree topology with local sinks and monitoring message aggregation minimizes the power consumption (see section 5.3.3 for details).
- **Power consumption:** The total power consumption (including all layers, link testing and clock drift estimation) amounts to as low as $P_{tot} \approx 100\mu W$ (for example in the Bundhaus scenario of SBT).

The application uses routing information and a double-tree topology with local sinks, where the following items apply.

- Every node has two monitoring and two alarm parents (M_1, M_2 , A_1 and A_2)
- The two alarm trees (the first tree holds the main links whereas the second tree contains the alternative links) are generated with a *Dijkstra-Based 2-SafeLinks Algorithm.* This is a tree with the minimal maximal hop count $h_{\max}^{alarm}$ (low latency). This is important to maximize the wake up interval T_w and thus to minimize the power needed for sampling.
- The two monitoring trees are generated with a *Prim-Based 2-SafeLinks Algorithm* to obtain a monitoring tree with optimal link qualities and a small number of children per parent K_{mon}. Nodes are monitored in subtrees of depth $h_{\max}^{subtree}$ by a local sink. Missing nodes are reported to the global sink sending a technical alarm on the fast alarm tree.
- All required information is read from the Routing Information Tables during initialization. During operation routing is static, which means no re-routing takes place. The alternative parents and the *2-SafeLinks Algorithm* guarantee connectivity to the sink in case of link breaks.

The application layer exchanges information with the network layer to access the routing tables and with MAC layer to receive and send messages.

6.5 Tests and Simulation Results

Tests are carried out for the scenarios provided by SBT. For the tests and their evaluations, the given steps were followed:

1. Generation of the Routing Information Tables using MATLAB Graph Analyzer Tool
2. Simulations were run with these tables and other inputs, e.g., pathloss matrix, node positions etc.
3. The output files were processed with scripts to determine the power consumption, timeliness and correctness of the alarms.

In the following section, we consider Bundeshaus scenario as a test example.

6.5.1 Bundeshaus Scenario

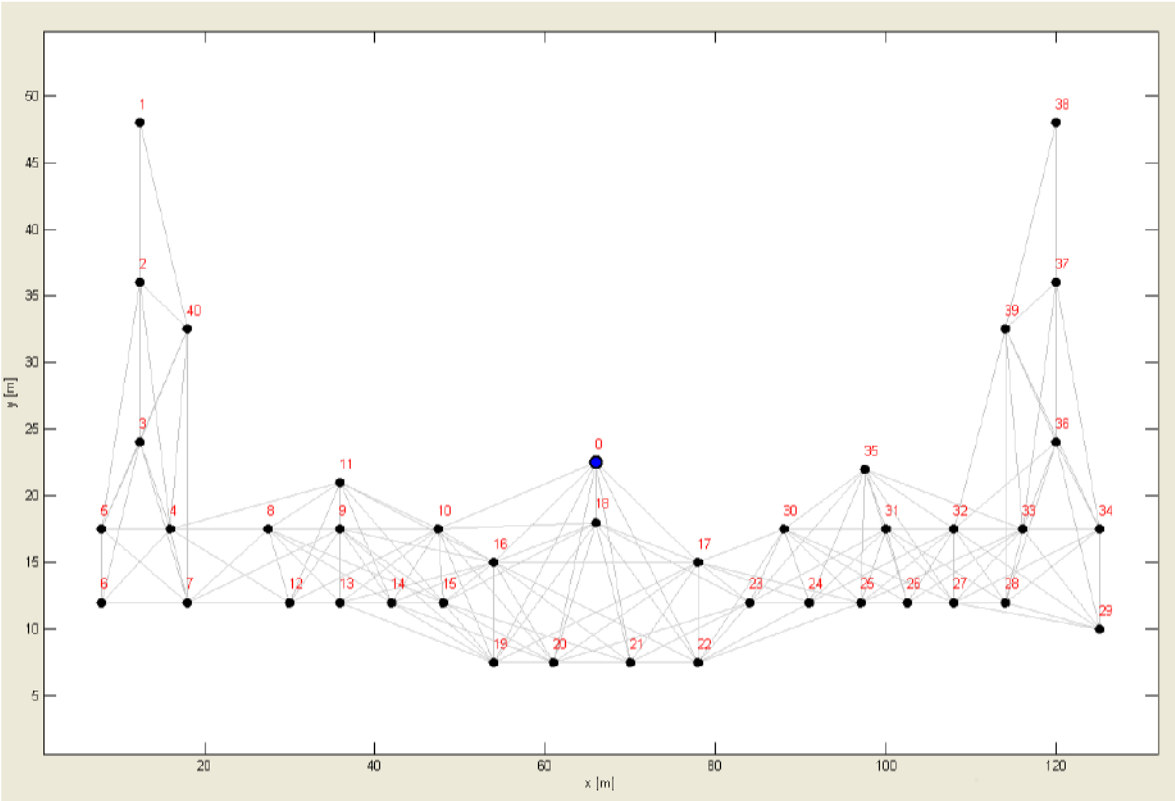


Figure 44: *Bundeshaus* scenario with 40 fire detection nodes (black) and a sink node (blue). The physical links, i.e. with minimum -113dBm signal strength, are also displayed.

6.5.1.1 *Generated Monitoring and Alarm Trees*

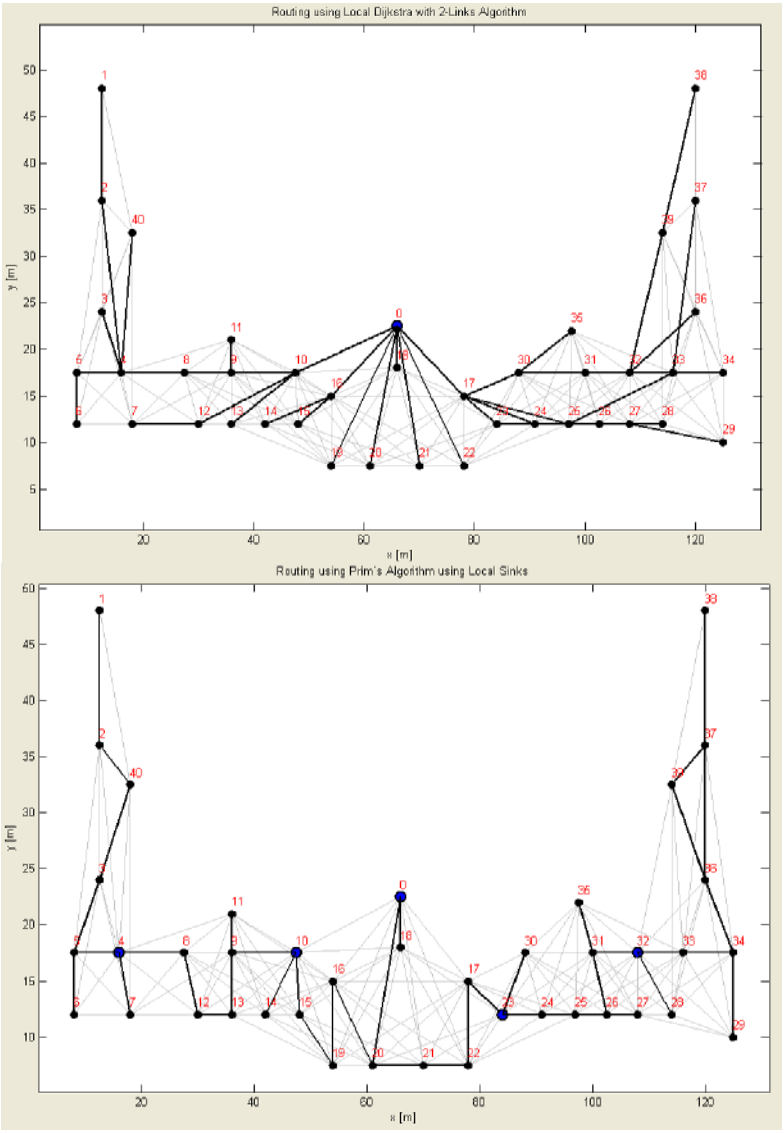


Figure 45: Generated Alarm-Tree for the *Bundeshaus* scenario (top). It has a maximum number of branches of $K_{Alarm} = 4$ whereas the maximum number of hops is $h_{\max}^{alarm} = 5$. The generated Monitoring-Tree (buttom) has 4 local sinks and 1 global sink (all in blue). The routing tree has a maximum number of branches of $K_{Mon} = 2$ whereas the maximum number of hops is $h_{\max}^{mon} = 15$.

6.5.1.2 *Animation of the Simulations using NAM*

The reception and transmission of the messages as well as the state of the nodes could be displayed by the network animator NAM. A NAM-snapshot of the Bundeshaus scenario is illustrated in Figure 46.

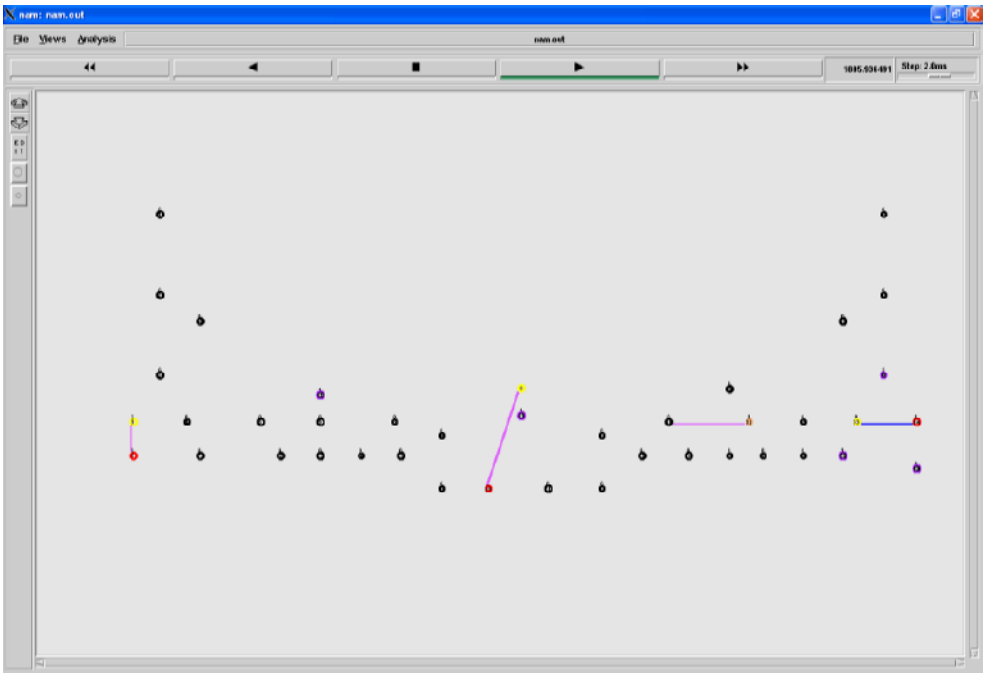


Figure 46: The NAM-snapshot of the simulation for the Bundeshaus scenario. The lines denote the transmission of messages. Different colors of the lines differentiate the frequency with which the messages are sent. The sleeping nodes are displayed in black, whereas the nodes on transmission are colored in red, the reception in reception are in yellow. The violet nodes are listening the medium if it is idle or not.

6.5.1.3 *Power Consumption Results*

The results we obtained for 10 days normal operation are summarized in Table 32. All tests are carried out with typical CWSN parameters, $\Delta_M = 220s$, $\Delta_h = 12s$, $\Delta_L = 1h$, $T_w = 1.5s$.

$h_{\max}^{subtree}$	K_{Mon}	K_{Alarm}	$h_{\max}^{Mon}$	$h_{\max}^{Alarm}$	P_{mean} [uW]	$P_{\min}$ [uW]	$P_{\max}$ [uW]	θ [ppm]	Ok-Bit
5	2	4	15	5	23	23	134.33	5	No
5	2	4	15	5	93.44	79.8	107.07	5	Yes
5	2	4	15	5	111.9	90.27	187.76	30	No
5	2	4	15	5	129.62	95.42	154.55	30	Yes

Table 32: Normal operation, without link breaks and fire alarm. The simulation time was 10 days. The results show that total power consumption of 100μW can be achieved utilizing *Ok-Bit messages* instead of full status arrays and providing small *clock drifts* (green number).

7. Conclusion

Fire detection – wireless sensor network (FD-WSN) applications are safety critical applications, where the sensor nodes monitor the environment and notify the sink the with alarms, when fire events are detected. The latencies in a FD-WSN cannot exceed some limits due to the safety considerations and the power consumption for the communication must be kept very low because the wireless nodes are battery-driven, usually simple AA-Alkaline batteries for low cost.

The goal of this thesis was to devise and develop routing algorithms and concepts to be used for the specific CWSN application, which is a FD-WSN application with strict timeliness and power requirements (see section 1.2.3). The developed algorithms had to be locally implementable and they had to guarantee the connectivity of the routing tree in case of communication failures, specified in [1]. Therefore, we first developed a routing algorithm called *Pairing Algorithm* that enables the local implementation of the standard MST and SPT algorithms without large deviations from the optimal algorithms. As the second step, a *N-SafeLinks Algorithm* was introduced and merged with the *Pairing Algorithm.* The *N-SafeLinks Algorithm* provides static re-routing, where we proved, that the connectivity to the sink (loop-freeness) is guaranteed even in the case of link breaks[21].

To minimize the power consumption by keeping the timing requirements, we developed a *Double-Tree Concept* for the CWSN application. In this concept, the alarms were sent over a SPT based fast alarm tree and the monitoring operation uses a MST based monitoring tree. A further concept that we introduced for power optimization was the *Local Sinks Concept*, where special nodes called *Local Sinks* monitors their own subtrees. We merged this concept with the *N-SafeLinks Algorithm* and obtained *Local Sinks Algorithm* that is specifically adapted for the CWSN Application.

For the simulations we developed a MATLAB Graph Analyzer Tool. The simulations that we carried out to compare our algorithms with the standard algorithms showed that the newly developed algorithms perform very close to optimum MST and SPT algorithms in terms of these metrics. With the tests on convergence, we showed that the new routing algorithms do not introduce significant limitations on the topology and at last we have seen that the new routing algorithms are much more energy efficient than the standard MST and SPT algorithms.

Finally, we implemented the CWSN application (network and application layers) in GloMoSim, where the routing information is generated off-line by the MATLAB Tool and saved in the network layer. The simulations results indicated that a minimum total power of about $100\mu W$ can be reached. The routing algorithm with the CWSN application is also proved to be robust against link breaks and suitable to provide the required timeliness.

Although the main goals and requirements are achieved, there are the following open issues and future work:

- Implementation of the routing algorithm in the GloMoSim. A preliminary protocol concept is given in section 3.3 but the protocol must be refined for the implementation.

- In case of permanent environment changes the static tables may not be enough. A protocol must be devised first to detect the permanent topology changes. The protocol

[21] The *N-SafeLinks Algorithm* can handles up to *N*-1 link breaks per node.

should realize re-routing where it updates the routing tables without violating the connectivity.

- More GloMoSim tests must be carried out with several scenarios to find the optimal cost functions to use in the present routing algorithms (until now only *Prim's* and *Dijkstra's* metrics have been used). Tests are also needed to find the optimal RSSI mapping for the link weights.

Appendix A: Symbols and CWSN Parameters

Symbol	Value (typical)	Description
Δ_M	220 s	interval of monitoring messages
Δ_h	12 s	hop interval during monitoring wave
Δ_L	1h	link testing interval
τ_{wave}	60 s	propagation time of monitoring wave within subtree to local sink $\tau_{wave} = h_{\max}^{subtree}\Delta_h$
$h_{\max}$	20	maximal number of allowed hops
$h_{\max}^{subtree}$	5	maximal number of allowed hops in monitoring subtree
$h_{\max}^{alarm}$	6	maximal number of hops in alarm tree
$h_{\max}^{mon}$		maximal number of hops in monitoring tree
$N_{retries}^{mon}$	1	number of late wave considered by the local sink
$N_{retries}^{alarm}$	1	number of considered retries of alarm messages
N_S		number of slots per hop interval
$K_{\max}$	3	maximal number of children per parent in monitoring tree
K_{Mon}	3	maximal number of children that selected same parent as alternative monitoring route
K_{Alarm}	6	maximal number of children that selected same parent as alternative alarm route
θ	1ppm – 100ppm	quartz drift tolerance
L_{DATA}	45/13 bytes	size in bits of a data packet without/with OK-BIT
L_{ACK}	13 bytes	size in bits of an ack packet
L_{Link}	13 bytes	size in bits of an empty link testing packet
T_w	1.5 s	WiseMac wakeup interval
T_P		WiseMac preamble duration
T_{DA}		duration of data and ack package transmission
P_{tot}		total power consumed
$P_{Sampling}$		power consumed by sampling
P_{Mon}		power consumed by alarm
P_{Link}		power consumed by link testing

Table 33: Parameters and symbols used in the CWSN application.

Appendix B: Radio Parameters

Symbol	Description	Value
P_Z	Power consumption in RADIO_SLEEP state	0.0006 mW
$\hat{P}_R$	Power consumption increment in RADIO_READY_RX, RADIO_RECEIVING, RADIO_SWITCH_RX and RADIO_SWITCH_TX states	59.7 mW
$\hat{P}_T$	Power consumption increment in RADIO_TRANSMITTING state	75.3 (+5 dBm)
$\hat{P}_S$	Average power consumption increment in RADIO_SETUP_RX and RADIO_SETUP_TX states.	14.7 mW
T_S	Setup time	3.85 ms
T_T	Turn-around time and frequency change time	0.64 ms
T_I	RSSI integration time (5 symbols at 10 kbps)	0.6 ms

Table 34: Parameters of the given radio model (table taken from [3]).

List of Abbreviations

CSEM	Centre Suisse d'Electronique et de Microtechnique
CWSN	Critical Wireless Sensor Network
MST	Minimum Spanning Tree
SBT	Siemens Building Technologies, Siemens Schweiz
SPT	Shortest Path Tree
WSN	Wireless Sensor Network

References

[1] P. Blum, Syrah WL – *SyrahNet, Problem Specification,* January 2006.

[2] P. Blum, CTI Status Meeting – *Status Presentation,* September 2006.

[3] A. El-Hoiydi, *Medium Access Control Protocol for Wireless Fire Detection - Analysis and Specification*, CSEM Technical Report, June 2006

[4] A. El-Hoiydi, *WiseMAC Glomosim Model*, CSEM Technical Report, Juli 2006

[5] A. El-Hoiydi, *Radio Models*, CSEM Technical Report, March 2006

[6] R. C. Prim, *Shortest connection networks and some generalizations*, Bell System Technical Journal, Vol. 36, pp. 1389-1401, 1957.

[7] E. V. Dijkstra, *A note on two problems in connection with graphs*. Numerische Mathematik 1, pp. 269-271, 1959.

[8] B. Y. Wu, K.-M. Chao, *Spanning Trees and Optimization Problems*, Chapmann & Hall/CRC, 2004.

[9] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in the improved network optimization algorithms,* J. ACM, volume 34, pp. 596-615, 1987.

[10] D. Bertsekas and R. Gallager, *Data Networks*, Prentice Hall, Inc., 1987.

[11] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves, *A loop-free extended Bellman-Ford routing protocol without bouncing effect*, in Symposium proceedings on Communications architectures & protocols. ACM Press, pp. 224-236, 1989.

[12] W. D. Tajibnapis, *A correctness proof of a topology information maintenance protocol for a distributed computer network,* Communications of the ACM, no. 20, pp. 477-485, 1977.

[13] R. G. Gallager, P. A. Humblet and P. M. Spira, *A distributed algorithm for minimum weight spanning trees,* ACM Transactions on Programming Languages and Systems, volume 5-1, pp. 66-77, 1983.

[14] B. Awerbuch. *Optimal Distributed Algorithms for Minimum Weight Spanning Tree,* Counting, Leader Election, and Related Problems. In Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC), New York City, New York, May 1987.

[15] M. Faloutsos and M. Molle, *Optimal distributed algorithm for minimum spanning trees revisited*, Symposium on Principles of Distributed Computing, pp. 231-237, 1995.

[16] R. Perlman, *An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN*. In Proc. of the Ninth Symposium on Data Communications, pp. 44-53, 1985.

[17] X. Zeng, R. Bagrodia and M. Gerla, *GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks*, In Proceedings of the 12th Workshop on Parallel and Distributed Simulations (PADS '98), Banff, Alberta, Canada, May 1998.

[18] T. Ikikardes and M. Hofbauer, *Protocol Development, Analysis, Concepts and Glomosim Implementation*, Siemens Schweiz AG / BIC-ET, Technical Report, July 2006.

[19] M. Hofbauer, *Time-synchronization of hops (global) and children (local) and parent-frequency distribution scheme*, Siemens Schweiz AG / BIC-ET Protocol Concept, February 2006.