UNIVERSITÉ CATHOLIQUE DE LOUVAIN
ÉCOLE POLYTECHNIQUE DE LOUVAIN
DÉPARTEMENT D'INGÉNIERIE INFORMATIQUE

On monitoring large-scale Wireless Mesh Networks

Promoteur:
Prof. Olivier Bonaventure
Lecteurs:
Prof. Yves Deville
Dr. Benoit Donnet

Mémoire présenté en vue de l'obtention du grade de **master ingénieur civil en informatique** option **networks and telecommunications**
par Gregory Detal

Louvain-la-Neuve
Année académique 2008-2009

Acknowledgments

I wish to especially thank Prof. Olivier Bonaventure for giving me the great opportunity to work on this project. I would like also thank him for providing the necessary assistance during all those months.

I would like to thank Benoit Donnet for all his readings and useful corrections, without that I could not have completed this project.

I am also pleased to thank Delphine Vandresse, for all her support and encouragements all along this work.

I finally want to thank my parent for their help and support throughout my studies.

Abstract

THESE last years have seen the emergence of *Wireless Mesh Networks* (WMNs), a technology for providing cost effective broadband Internet access. Typically, a WMN follows two objectives: offer connectivity to end-users (i.e., wireless clients) and form a self-organized wireless backbone. However, due to the highly dynamic characteristic of wireless devices, it is difficult to ensure if those two objectives are met by the WMN. Monitoring a WMN is thus a key issue for network deployers.

In this work, we define three major objectives that a monitoring tool has to respect in order to facilitate the network deployers' life. First, to provide a real-time monitoring, the basis of all monitoring. Second, to avoid monitoring overheads, as the wireless capacity is limited, we prefer to limit the interference with the client traffic. Third, being able to validate a network status, as due to the dynamic characteristic, the routing protocols may not accurately represent the topology. In particular, based on the latter objective, we provide an efficient algorithm for testing paths in a WMN. We introduce the *topology coverage*, which is a way to cover the topology graph with path, i.e., all links of the network is covered with at least one path, in order to test them and accurately evaluate the network status.

In this thesis, we evaluate our solution on two aspects. First, we demonstrate on generated topologies the efficiency of our topology coverage. In particular, we show that our algorithm behaves well with all possible simulated topologies and has take a reasonable amount of time to compute a solution, less than 100 seconds for a 250 nodes network. Second, we construct a small testbed and deploy a prototype of our solution. We perform a set of experiments and show the usefulness of our solution when the routing protocols does not detect failures.

Résumé

Ces dernières années ont vu l'émergence des *Réseaux Sans-fils Maillés* (RSM), une technologie fournissant un accès rentable à l'Internet haut-débit. Habituellement, un RSM suit deux objectifs: offrir la connectivité aux utilisateurs finaux (c'est-à-dire, des clients sans-fils) et forme un 'backbone' auto-organisée. Cependant, en raison de la caractéristique très dynamique des RSM, il est difficile de s'assurer si ces deux objectifs sont atteints par le RSM. Monitorer un RSM est donc un enjeu essentiel pour les exploitants de réseau.

Dans ce travail, nous définissons trois objectifs principaux que se doit d'implémenter un outil de monitoring, afin de faciliter la vie des responsables des réseaux. Premièrement, de permettre un monitoring en temps-réel, la base de tout monitoring. Deuxièmement, d'éviter les 'overheads', comme la capacité du sans-fils est limitée, nous préférons limiter les interférences avec le trafic client. Troisièmement, être capable de valider un état de réseau, en raison de la caractéristique dynamique, les protocoles de routages peuvent ne pas représenter correctement la topologie. En particulier, basé sur ce dernier objectif, nous fournissons un algorithme efficace pour tester les chemins d'un RSM. Nous introduisons la *couverture de réseau*, qui est un moyen de couvrir un graphe de topologie avec des chemins, c'est-à-dire, que chaque lien soit couvert par au moins un chemin, dans le but de les tester et évaluer avec précision l'état du réseau.

Dans ce travail de fin d'étude, nous évaluons notre solution de deux manières différentes. Premièrement, nous démontrons, sur des topologies générées, l'efficacité de notre couverture de réseau. En particulier, nous montrons que notre algorithme se comporte bien dans avec toutes les différentes topologies simulées et qu'il prend un temps raisonnable pour calculer une solution, moins de 100 secondes pour un réseau de 250 noeuds. Deuxièmement, nous construisons un petit banc d'essai et déployons un prototype de notre solution. Nous réalisons un ensemble d'expériences

et montrons l'intérêt de notre solution quand les protocoles de routages ne détectent pas les erreurs.

Contents

1 Introduction **1**

1.1 Objectives 1

1.2 Chapter overview 3

2 Wireless Mesh Network **4**

2.1 General overview 4

2.2 Routing protocol families 6

2.2.1 Proactive 6

2.2.2 Reactive 6

2.2.3 Hybrid 7

2.3 Routing protocols 7

2.3.1 BATMAN 7

2.3.2 AODV 9

2.3.3 OLSR 11

2.3.4 MeshDV 13

2.3.4.1 Routing protocol inside the backbone 14

2.3.4.2 Client management protocol 15

2.4 Comparison 16

2.5 Conclusion 18

3 Monitoring **20**

3.1 Related works and existing solutions 21

3.2 Objectives 22

3.2.1 Real-time topology monitoring 22

3.2.1.1 The Management plane approach 23

3.2.1.2 The control plane approach 24

3.2.1.3 Evaluation 24

3.2.1.4 In the OLSR case 26
3.2.2 Limit the amount of monitoring messages 26
3.2.3 Validate network state viewed by Routing protocols 27
3.3 Characteristics to monitor 28
3.4 Conclusion 29

4 Topology coverage 30
4.1 Graph theory 30
4.2 The topology coverage problem 32
4.2.1 A relaxed problem 33
4.3 A typical test 35
4.4 Centralized Algorithm 36
4.4.1 Algorithm behavior 37
4.4.2 Algorithm complexity 38
4.4.3 Possible heuristics 39
4.4.4 Evaluation 39
4.4.4.1 Topology models 40
4.4.4.2 Results 40
4.5 Decentralized Algorithm 44
4.5.1 Graph partitioning 44
4.5.2 Parallel computation & Leader election 44
4.5.3 Algorithm complexity 46
4.5.4 Possible heuristics 47
4.5.5 Evaluation 47
4.5.5.1 Topology models 47
4.5.5.2 Results 50
4.6 Conclusion 57

5 A practical application 59
5.1 Testbed 59
5.1.1 Material 59
5.1.2 Environment 60
5.2 Architecture & roles 60
5.2.1 The sniffer 61
5.2.2 The topology server 62

5.2.3 The leader 62
5.2.4 The pinger 63
5.3 Configuration 63
5.3.1 The routers 63
5.3.1.1 Base configuration 63
5.3.1.2 Hardware & software Limitations 64
5.3.1.3 Interfaces configuration 64
5.3.1.4 Behavior configuration 65
5.3.1.5 OLSR configuration 65
5.3.2 The sniffers 66
5.4 Implementation 66
5.4.1 Application Programming Interface (API) 67
5.4.2 Details & functionalities 68
5.4.2.1 The sniffd 68
5.4.2.2 The monitor 70
5.4.2.3 The leaderd 70
5.4.2.4 The pingd 71
5.5 Review & tests 72
5.5.1 Evaluation 72
5.5.2 Node down 73
5.5.3 Packet loss 74
5.5.4 Delay 75
5.6 Conclusion 76

6 Conclusion and future work **77**
6.1 Further work 79

Bibliography **80**

Appendices

A Decentralized Algorithm: Evaluation **86**
A.1 Heuristics evaluation 86
A.1.1 Regular networks 86
A.1.2 Structural generators 88
A.1.3 Degree-based generators 89

A.2 Coverage quality 90
A.2.1 Regular networks 90
A.2.2 Structural generators 91
A.2.3 Degree-based generators 92

B A practical application & simulation: Source code 93

List of Figures

2.1 Infrastructured/backbone WMN example. 5
2.2 BATMAN – example of flooding OGM when a new node (A) joins, each number representing a step 8
2.3 AODV – Route request example. 10
2.4 OLSR – Broadcasting packet in a wireless mesh network originated from the center node. 12
2.5 OLSR – Broadcasting packet in a wireless mesh network from the center using a selection of MPRs (represented in black). 12
2.6 MeshDV – CREQ example. 16

3.1 Network Architecture with active monitoring. 24
3.2 Network Architecture with passive monitoring. 25

4.1 The P_4-decomposition for an exact cover of the graph. 31
4.2 The $(2, 4)$-decomposition for an exact cover of the graph. 31
4.3 Shortest path forwarding – example topology and corresponding forwarding tables. Each row of the table give the next-hop to reach a node. . . . 33
4.4 Example topology graph with four possible route. 38
4.5 Algorithm search example for Fig. 4.4. 38
4.6 Waxman – Number of paths used to cover a graph when the number of edges varies and the number of nodes is fixed. 41
4.7 Waxman – Number of edges covered by at least 2 paths, i.e., over-covered, when the number of nodes is fixed. 41
4.8 Waxman – Average number of paths per edges with the h_{basic} heuristic. . 42
4.9 Waxman – Maximum number of different measurement paths starting at each node. 42
4.10 Waxman – Average computation time w.r.t. the number of nodes. 43

4.11 Waxman – Average computation time w.r.t. the number of edges ($n = 120$). 43
4.12 Waxman – Average *All-Pairs Shortest Paths* computation time w.r.t. the number of nodes. 43
4.13 Partitionning of a 25 nodes manhattan grid into 3 parts. 45
4.14 Partitionning of a 24 nodes random topology into 5 parts. 45
4.15 Manhattan topology. 48
4.16 Fully connected topology. 48
4.17 Hypercube topology ($n = 3$). 48
4.18 Coverage time box-plot for each topology models ($h_{overcover}$). 51
4.19 Computation time cumulative distribution (all models together). 51
4.20 Waxman – Number of paths used to cover ($h_{overcover}$), comparison of the two algorithms. 53
4.21 Waxman – Maximum number of path covering each edges ($h_{overcover}$), comparison of the two algorithms. 53
4.22 Waxman – Average number of different paths starting at each node. 53
4.23 Waxman – Maximum number of different paths starting at each node, comparison of the two algorithms. 53
4.24 Waxman – Average number of times each edges is over-covered. 54
4.25 Waxman – Number of paths used to cover the topology graph. 54
4.26 Waxman – Average computation time w.r.t. the number of nodes for cluster size $\in [10, 20, 40]$ nodes with $h_{overcover}$. 55
4.27 Waxman – Path length distribution for cluster sizes $\in [10, 20, 40]$ nodes with $h_{overcover}$. 55
4.28 Waxman – Comparison of the number of paths when the three heuristics are used. 56
4.29 Waxman – Comparison of the Maximum number of paths starting at each node when the three heuristics are used. 56
4.30 Waxman – Comparison of the Maximum number of paths covering each edge when the three heuristics are used. 56
4.31 Waxman – Length distribution of path used to cover the graph. 56

5.1 The Accton MR3201A router. 60
5.2 Testbed – Routers and Sniffer positioning in the Reaumur building. 61
5.3 State of the testbed view by the application. 73
5.4 State after the reboot of one node. 73

5.5 State after the simulation of 70% packet loss. 75
5.6 State after the simulation of an increasing of 1000ms in the delay on one node. 75
5.7 Probing delay evolution when an increase of 100ms delay is applied (at 400sec). 76

A.1 Manhattan – Comparison of the Maximum number of paths covering each edge when the three heuristics are used. 87
A.2 Hypercube – Comparison of the Maximum number of paths starting at each node when the three heuristics are used. 87
A.3 Full mesh – Comparison of the number of paths when the three heuristics are used. 87
A.4 Full mesh – Comparison of the Maximum number of paths starting at each node when the three heuristics are used. 87
A.5 Hierarchical Top-Down – Comparison of the number of paths when the three heuristics are used. 88
A.6 Hierarchical Top-Down – Comparison of the Maximum number of paths starting at each node when the three heuristics are used. 88
A.7 Hierarchical Top-Down – Comparison of the Maximum number of paths covering each edge when the three heuristics are used. 88
A.8 Hierarchical Top-Down – Length distribution of path used to cover the graph. 88
A.9 BA – Comparison of the number of paths when the three heuristics are used. 89
A.10 GLP – Comparison of the Maximum number of paths starting at each node when the three heuristics are used. 89
A.11 GLP – Comparison of the Maximum number of paths covering each edge when the three heuristics are used. 89
A.12 GLP – length distribution of path used to cover the graph. 89
A.13 Manhattan – Maximum number of different paths starting at each node. 90
A.14 Hypercube – Average number of times each edges is over-covered. 90
A.15 Full mesh – Average number of different paths starting at each node. . . . 90
A.16 Full mesh – Average number of times each edges is over-covered. 90
A.17 Hierarchical Top-Down – Average number of different paths starting at each node. 91

A.18 Hierarchical Top-Down – Maximum number of different paths starting at each node. 91
A.19 Hierarchical Top-Down – Average number of times each edges is over-covered. 91
A.20 Hierarchical Top-Down – Number of paths used to cover the topology graph. 91
A.21 BA – Average number of different paths starting at each node. 92
A.22 GLP – Maximum number of different paths starting at each node. 92
A.23 GLP – Average number of times each edges is over-covered. 92
A.24 BA – Number of paths used to cover the topology graph. 92

Chapter 1

Introduction

Wireless Mesh Networks (WMNs) [AWW05] have witnessed a tremendous growth over the last years. A WMN is a communication network involving radio nodes organized in a mesh topology. The main objective of WMNs is to offer connectivity to end-users, which can be laptops, cell phones, etc. To offer this connectivity, a WMN deploys a self-organized backbone composed of two kind of nodes: routers and gateways.

The emergence of WMN in the last years can be explained by its ease and simplicity as it provides a self-configured network preserving connectivity in case of network failure. The connectivity is maintained through the mesh character of the topology that supplies great stability in case of changing conditions.

Likewise, integration is another reason of the popularity of WMNs. Mesh hardware is typically small, noiseless, and easily encapsulated in weatherproof boxes. This means it also integrates nicely outdoors as well as in human housing. It allows one to cover a difficult area, e.g., offering Internet in an old building, a third world city, building an enterprise networking, etc. All those case plead for an extension and a development of WMNs.

1.1 Objectives

Handling and controlling a large-scale WMN is a difficult task. As everything is changing very fast, understanding the source of the problem becomes very difficult.

E.g., it is not obvious to understand that the source of a problem (e.g., loss of most of the packets) is the distance between two nodes. It is even difficult if the routing protocol does not detect the failure and still consider the link as usable.

The subject of our work is to provide a monitoring tool that will efficiently help WMNs maintainers in their daily task and avoid them painful hours.

WMN monitoring and analyze has been a research topic these last years [SFQ+07, GMCN08, NBB+07, NK08]. However some of them do not define clearly their monitoring objectives. Therefore, we chose to define three objectives that are essential to an efficient WMN monitoring tool. First, to provide a real-time monitoring, the basis of all monitoring. Second, to avoid monitoring overheads, as the wireless capacity is limited, we prefer to limit the interference with the client traffic. And finally, being able to validate a network status.

This latter objective is the main contribution of this work. The other objectives have already been discussed previously [SGG+02, SG04, GMCN08]. As WMNs have a highly dynamic characteristic, we need to validate the current network status, in order to, e.g., validate the routing protocol configuration, the nodes positions, etc. To fulfill this objective, we provide a contribution: the *topology coverage*. The topology coverage is a way to cover a network topology graph with paths, i.e., all edges of the network is covered with at least one path, in order to test them and validate the network status. To find a topology coverage, we develop a centralized algorithm. However, it comes with a major flaw: the computation time. It takes too much time to find a solution. We, therefore, develop a decentralized solution based on graph partition [KL70] in clusters. This latter algorithm presents a time saver compared to the first one, while still keeping on average the same behavior.

To evaluate our solution, we construct a small testbed and deploy a prototype of our solution. The prototype is a very simple graphic user interface, allowing to visualize the topology and different problems. We perform a set of experiments and show that, even if this implementation is not one of our goals, it provides an efficient way to detect issues and help the maintainer solve them, especially when the routing protocols does not detect those failures.

1.2 Chapter overview

This report is divided in four chapters. The first one is an overview of WMNs. It introduces in detail the WMN behavior and functionalities and explain four routing protocols belonging to different families.

The second chapter fully describes three objectives for an efficient monitoring, in order to facilitate the life of network maintainers by helping them understanding the network behavior. These objectives are linked to the limited capacity of wireless devices. The first objective is to provide an efficient real-time monitoring. The second to limit the monitoring overhead. And the last, to validate the network status.

The third chapter develops the theoretical part of the report. As there are no research about the latter monitoring objective, we decide to present a theoretical contribution: the *topology coverage*. Deep analysis of the different algorithms is performed in order to validate their efficiency.

The last chapter contains the description of a practical implementation of all the objectives described in the second chapter. The application is tested, on a small testbed, in order to visualize its correct behavior in face of changes.

We finally conclude this thesis by reminding its mean contributions and discerning potential future directions.

Chapter 2

Wireless Mesh Network

2.1 General overview

Wireless mesh networks (WMNs) [AWW05] are a special type of communication network involving two kind of radio nodes: *mesh routers* and *mesh clients*. Mesh routers are those with a limited mobility and form the *mesh backbone*, while mesh clients can be either stationary or mobile and can form a mesh network among themselves and with mesh routers. Theses two types of nodes are connected wirelessly and together compose the *mesh topology*. The coverage area depends on nodes transmission range and is sometimes called the *mesh cloud*. A WMN can be viewed as a subset of *mobile ad-hoc networking* [LH98] (MANet) where the nodes are not free to move arbitrarily. Thus, the infrastructure is more or less static and has a low dynamic aspect.

Recently, there has been a real interest in WMNs technology, mostly because it provides an infrastructure where nodes auto-establish and maintain connectivity between themselves as in MANet. A WMN goes further than MANet as it is automatically self-organized and self-configured, which is an interesting feature for a low-cost, reliable, and easy manageable solution. Conventional clients (e.g., laptop, desktop, phone, etc.) can either connect the mesh network via a wired network (e.g., ethernet), directly connected to a mesh router, or either connect the mesh cloud with their wireless network interface.

The most common architecture is the infrastructured/backbone WMN, as illustrated in Fig. 2.1. Respectively, wired and radio links are displayed as solid and dashed lines. It includes two types of routers, the so-called backbone mesh

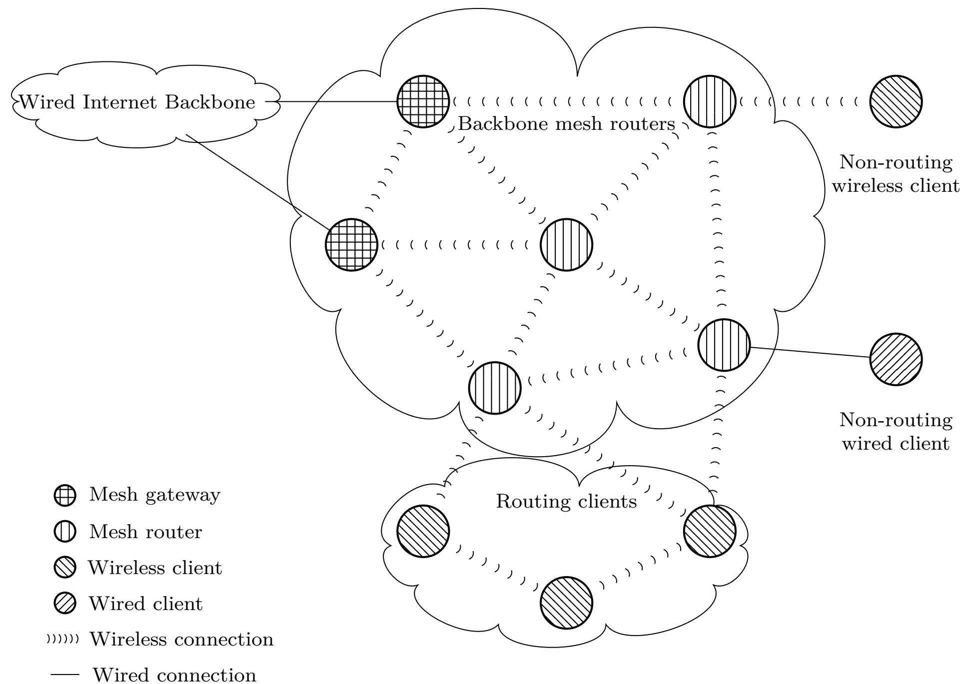


Figure 2.1: Infrastructured/backbone WMN example.

routers and also router acting as gateway providing an Internet access to the network. These wireless routers form the core by building the meshed, self-configured, WMN backbone. Conventional clients can directly communicate with mesh routers. If the wireless standard (e.g., IEEE 802.11 [wif07], WiMAX [wim04], etc.) used by the backbone differs from the one used for the client connectivity, clients must communicate with a base station connected to the mesh network with an Ethernet connection.

Mesh routers and gateways are installed at fixed positions (e.g., roof of houses for instance). The backbone is a permanent infrastructure. However new routers can be readily added without modifying those already installed, since they use radio communication.

Clients, unlike mesh routers, have a high mobility. WMNs clients can leave and join at any time. In Fig. 2.1, two groups of clients are introduced. The *non-routing mesh clients* do not participate in the routing process of the backbone WMN. They are limited to establish a direct communication (wired or wirelessly) to a single mesh router only, as in conventional communication between a wireless client and its access point. The *routing mesh clients* are able to construct their own subnetwork,

by connecting not only mesh routers but also other routing mesh clients. This provides peer-to-peer networks among the end-users. Theses clients are not only passive nodes but need to to perform routing and self-configuration functionalities as well as providing end-user applications to customers.

2.2 Routing protocol families

Routing protocols are essential in a MANet, as it allows communication between two non-directly connected devices. The routing problem consists of finding an optimal path through the network according to some performance criterion. Nodes have initially no knowledge about the network topology. The main idea is that each node broadcasts its presence and forwards other's presence. After a certain time, a node knows enough information on the whole topology to reach any node in the network.

Routing protocols can be classified according the way they discover the network and how they construct their routing table. Below, we give a brief overview of the three main routing protocol classes.

2.2.1 Proactive

Proactive routing protocols [AWD04] construct the routing table before even needing it by distributing periodically the topology throughout the network. At any time, it can identify the current status of each node of the network. This has the advantage of minimizing the delay when a route is needed. The main drawback is that if the network is not stable (e.g., high mobility) the amount of data exchanged requires a large network bandwidth.

2.2.2 Reactive

Reactive routing protocols [AWD04] are developed to overcome the limitation of pro-active routing protocols. It constructs the routing table when node asks for a route. A node does not know the current network topology. Rather, it finds a route on demand by flooding route-request (control) packets. Reactive protocols have smaller control traffic overhead than pro-active protocols. However, these protocols can have larger delays due to the fact that a route needs to be discovered before data can be actually sent.

2.2.3 Hybrid

Hybrid routing protocols [AWD04] mix both of above mentioned protocols. It acts as a proactive protocol in a limited zone around the node and becomes reactive beyond that zone.

2.3 Routing protocols

Below, we give an overview of some popular routing protocols that can be used with WMN.

2.3.1 BATMAN

Better Approach To Mobile Ad-hoc Networking [NALW08] (BATMAN) relies on simple message exchanging, *originator message* (OGM), to measure a route quality. The information about the entire topology is not stored in each single node but spreads across the whole network. Each node stores and perceives information about the best next-hop towards all other nodes. The algorithm does not try to find a path to the destination, but rather forwards the packets to the next-hop which is on the best way to the destination.

BATMAN detects neighbors and distant nodes by sending and forwarding broadcast packets (OGM) using UDP. An originator message contains few information, such as an *originator* (synonym to a BATMAN node) address, a sequence number, and a *Time-To-Live* (TTL) to avoid loops. An OGM advertises an originator existence and is used for link quality and path detection.

Fig. 2.2 represents the OGMs flooding strategy by showing each message exchanged when a new originator joins (node A) the network. The flooding strategy can be decomposed in three steps:

1. Every time interval, an OGM is broadcasted to all the node's neighbors.

   In Fig. 2.2, Node A broadcasts its presence (arrival in the network) by sending an OGM to all its neighbors (nodes B and D), i.e., sending it to the broadcast address.

2. Received originator messages are forwarded to all nodes of the mesh cloud, until there is no need to re-broadcast them. In Fig. 2.2, Node B forwards the OGM received by A to node C. Each node is aware of the node A's presence.

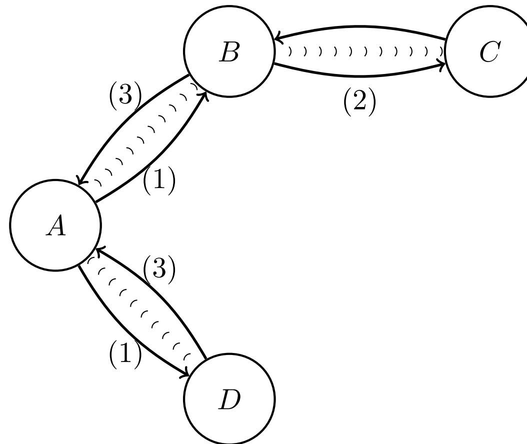


Figure 2.2: BATMAN – example of flooding OGM when a new node (A) joins, each number representing a step

3. A node considers a specific link to be bidirectional for a specific point in time if the reply (re-broadcast) of the self initiated OGM has been received from the corresponding link neighbor. In Fig. 2.2, when node A receives the self originated OGM from nodes B and D, it knows that the links to these neighbors are bidirectional.

A bidirectional link allows symmetric communication between nodes. BATMAN protocol ensures that a route consists of bidirectional links only.

BATMAN is based on sequence numbers. Each originator numbers its OGMs to avoid repeated messages. Those sequence numbers are recorded in a dedicated Sliding Window. The amount of sequence number recorded in the window, which contains always the set of recently received originator messages, is used as a metric to choose a path over another.

Once a node has received an originator message from another node, some information must be updated: the counter of recently received sequence number, and the sliding window must be purged to represent the upper and lower bound of the value's range. Upon these updates, each node must perform the *best neighbor ranking* algorithm, to find a route for a destination, as follows: The link from which the most in-window OGM sequence number has been received is said to be the new best link to the originator of the OGM. The chosen neighbor is then stored in the routing table as the next-hop for this originator. This means that the next-hop for a destination, is the neighbor from which the node has received the most information, i.e., the most in-window sequence numbers received for a destination. Thus, the

path chosen is not always the shortest but the most “reliable”, where there is less packet loss.

In addition to broadcasting originator’s presence, OGM can also carries information about a gateway to a network or a host connected to this originator. This is performed by attaching one or several *Host Network Announcement* (HNA) header extension to the originator message. If a node provides an Internet access, it must provide some information about its available bandwidth. So that individual node on the network can decide which Internet gateway to use.

When a node receives an OGM with an HNA extension, the route is added, similarly as for originator, to the routing table and the next-hop is the best ranking neighbor for the originator. In case of not receiving an OGM from a known originator for a given of time, the route is considered invalid and removed from the routing table.

2.3.2 AODV

Ad-hoc On Demand Distance Vector [PBRD03, LH98](AODV) is a distance vector algorithm. However, unlike classic Distance Vector routing, AODV is a reactive protocol. It requests a route when needed and does not require to maintain routes that are not used in active communications.

AODV is a loop-free algorithm: by avoiding the *count-to-infinity* problem, it offers a quick convergence when changes occur in the network. The count-to-infinity problem is the situation where nodes update each other in a loop. The use of destination sequence numbers guarantees the avoidance of this count-to-infinity problem. When a link failure occurs, only the affected nodes are immediately warned so that they are able to invalidate all routes using the lost link.

AODV defines three types of control messages (over UDP) to maintain routes:

***Route Request* (RREQ)** message is emitted by a node when a route to a destination is needed (e.g., if the route is expired, unknown or invalid).

***Route Reply* (RREP)** message is unicasted in response to a RREQ.

***Route Error* (RRER)** message is broadcasted, when a link failure occurs, to notify other nodes of the lost link.

When a route to a new destination is needed, a node broadcasts a RREQ to all its neighbors. The RREQ propagates through the network, until it reaches the

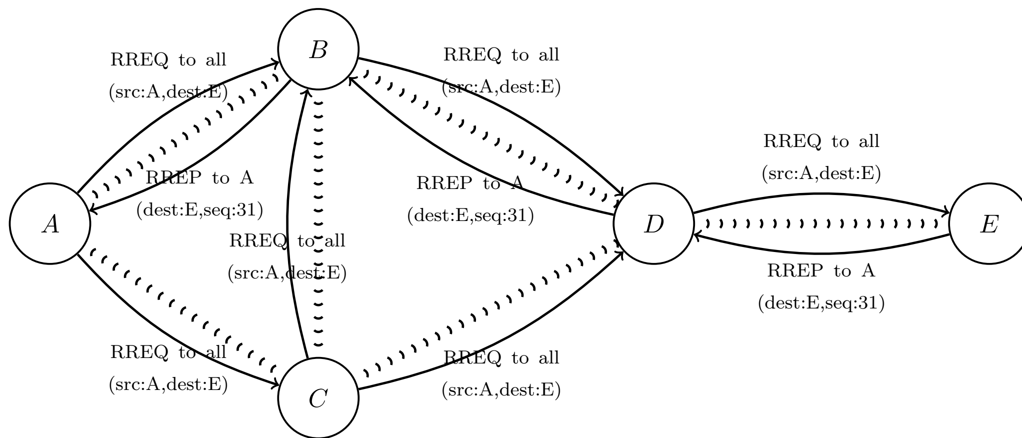


Figure 2.3: AODV – Route request example.

destination itself or a node with a "fresh enough" route to the destination. A "fresh enough" route is a valid route entry with more recent information than the one contained in the RREQ. The route is then made available to the originator of the RREQ by sending back a RREP.

Each node monitors the link status to its neighbors by sending periodically *Hello* messages (over UDP). A link is considered as broken if no Hello message was received within a given time. When a link is lost, a RERR is used to indicate to the affected nodes the destinations that are no longer reachable. To be able to perform this reporting mechanism, each node must maintain a precursor list of nodes that are likely to use this link as a next hop towards each destination.

AODV, in addition to enable unicast routing, is able to perform multicast routing, but it will not be covered in this document [PBRD03].

Fig. 2.3 presents a scenario in which node A wants to connect node E. It forwards a RREQ packet to its connected mesh routers (nodes B and C). When a node receives it, it first checks its own routing table and if it has a "fresh enough" route to the destination, it replies with a RREP message. In this example, we assume that no node has a "fresh enough" route to the destination. Therefore, the node A broadcasts a RREQ to all its neighbors, and waits for a RREP. If the reply is not received within a certain amount of time, the node must rebroadcast the RREQ or assume that the destination is not reachable after a certain number of retransmission. When an intermediate node receives a RREQ and does not have a route to the destination (nodes B, C and D). It rebroadcasts the RREQ. The intermediate node also creates a reverse route to the originator that will be used if

the node receives a RREP back to the node. This route has a lifetime much lower than an actual route entry. When the RREQ reaches a router having a "fresh" route to the destination (node E), a RREP is generated by this router and unicasted to the requesting node (node A). While forwarding the reply, a route is created and when the RREP reaches the originator node, there exists a route to the destination.

When a node detects a link failure, it first invalidates all routing entries which next hop is the incriminated neighbor. It then generates an RERR to all the nodes that are actively using each of these routes, informing them that these routes are no longer valid. These route errors may either be broadcasted or unicasted, depending on the number of nodes concerned by this link failure.

2.3.3 OLSR

Optimized Link State Routing [CJ03, Tø04, A. 09] (OLSR) is a table-driven proactive routing protocol. It was originally developed for MANet but can also be used in wireless mesh network. OLSR uses a link state scheme in an optimized manner to diffuse topology information. Unlike classic link state algorithm, it provides an efficient flooding mechanism for controlling traffic by reducing the number of required transmission and, consequently, preserve bandwidth. This mechanism is called *MultiPoint Relaying*.

OLSR uses flooding of *Hello* and *Topology Control* (TC) messages, respectively, to discover and diffuse topology information throughout the network. Each node computes the next hop destination for all nodes according to this information, providing that the shortest path is chosen. To avoid infinite loops, each packet is identified by a sequence number stored by each forwarding node. If a node receives a packet with a lower or equal sequence number than the last retransmitted packet, the packet is considered as a duplicate and is silently discarded.

The goal of MultiPoint Relaying is to restrict the set of nodes retransmitting a packet from all neighbor nodes to a subset of those. The size of this subset depends on the size of the network. This concept is achieved by selecting *Multipoint Relays* (MPR) from all possible directly connected neighbors. A node selects its MPRs such that there exists a path to all its 2-hop neighbors via a selected MPR.

Fig. 2.4 and Fig. 2.5 show a flooding scenario, with and without a MPR set, initiated by the central node. Each arrow shows a single transmission of the same message. In Fig. 2.4, there are 24 retransmissions, while in Fig. 2.5 there are only

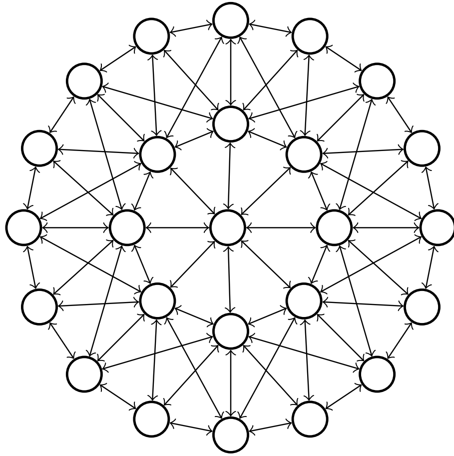


Figure 2.4: OLSR – Broadcasting packet in a wireless mesh network originated from the center node.

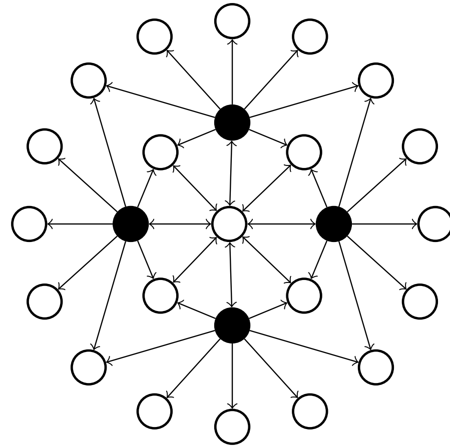


Figure 2.5: OLSR – Broadcasting packet in a wireless mesh network from the center using a selection of MPRs (represented in black).

four retransmissions. The classical way of broadcasting messages leads to a much higher number of retransmissions than with a MPR selection. Nevertheless the same result is achieved in the two possible ways, i.e., all nodes have received at least one version of the transmitted message.

OLSR nodes can be *multi-homed*, i.e., they can run OLSR on multiple communication interfaces using multiple identifiers. Each node with multiple interfaces must announce periodically information describing its interfaces configuration to other nodes in the network. This is accomplished through flooding a *Multiple Interface Declaration* (MID) message using the MPR flooding mechanism. A MID message essentially consists of a list of interfaces addresses on which a node runs OLSR. When adding a route to a node in the routing table, OLSR will add routes to all addresses contained in the MID message, using the same next hop.

Hello messages have other advantages than only detecting neighbors. They are also used to achieve link-sensing, two-hop detection and MPR selection signaling. A Hello message is composed of all known links and neighbors. A node must perform link-sensing on each interface, in order to detect links between the node interface and the neighbor interface. The two-hop neighbor detection is achieved by listing and storing all neighbors contained in a received Hello message from a symmetric neighbor. This database containing all nodes reachable via a symmetric neighbor is used for MPR calculation. After performing selection of MPR, nodes mark their

selected MPR in Hello message.

Based on the constructed symmetric neighbor list, topology information can be disseminated through the network using the multipoint relaying algorithm. This is done using TC messages. TC messages are flooded on a regular interval, but are also generated when changes are detected in the network. A TC message consists of a list of neighbor addresses and an advertised neighbor sequence number (ANSN). The ANSN is a sequence number associated with a node advertised neighbor set. It is not increased every TC generation, but represents the “freshness” of the information, i.e., a high ANSN. This means that ANSN is increased when a change in a node advertised neighbor set is detected.

Upon the reception of a TC message, information about the known topology must be updated. If this information has changed, then the route calculation algorithm must be performed and the routing table is recalculated. Therefore the routing table is only recalculated when a neighbor appears or disappears, when there is a two-hop neighbor change or when there is a MID change. To construct this table, OLSR performs a standard shortest path algorithm. If a node contains interfaces which do not participate in the OLSR MANet, it must inject external route information in the OLSR network. This is achieved by periodically issuing to all nodes a *Host Network Association* (HNA) message, containing sufficient information for any node to update the routing table information.

2.3.4 MeshDV

MeshDV [IF05, Rou08] is a *cross-layer* IPv6 hybrid routing protocol. Cross-layer means that information is shared between different communication layers from the OSI model. Routes between WMRs are established via a proactive approach, while a reactive routing protocol is used to manage client mobility and reduce routing table. The protocol is separated in two different sections, one to construct routes between each WMRs and another to obtain on-demand routes between clients and to manage clients mobility.

Packets sent between two clients associated on two different WMRs are encapsulated in an IPv6 tunnel between these two WMRs. This approach avoids keeping state in the WMRs along the path. Furthermore, when a client moves, only the WMRs at the edge must update information. WMRs along the path do not need to make update, while continuing to relay packets.

2.3.4.1 Routing protocol inside the backbone

The routing protocol must maintain a routing table containing an entry for each WMRs identified by its IPv6 address. To achieve the construction of this routing table, each WMR sends periodically its routing table to all is neighbors via a *Route Update* (RU) message. This proactive route computation is performed using a Distance Vector approach. The metric used within MeshDV is a cross-layer one. This metric is based on the transmission rate information obtained from the Data Link layer (MAC layer).

RU messages may be sent via two different ways. The first one consists of sending the whole routing table, while the second one consists of sending only recently changed routing entries. These two ways are called respectively *Full Dump* and *Partial Dump*. In order to maintain consistency in the network each WMR must send periodically a Full Dump of its routing table. This is known as *Periodic Route Update*. Between two Periodic Route Updates, some changes might occur in the network. When these changes are consequent, it is not possible to wait for the next Periodic Route Update. So, when a node detects an important change, it must send a Partial Dump to announce the event. However, if the Partial Dump announces more than 60% of the total number of routing entries in the routing table, a Full Dump must be sent instead.

MeshDV tries to aggregate updates in the network by setting a minimal time between two RU. This is interesting when there exists a flapping (up and down frequently) link in the network because it is not efficient to send an RU message each time a single change occurs.

MeshDV allows distinction between changes, i.e., when important changes (e.g., broken link, new router, etc.) occurs a Partial Dump is sent, while non important changes are not announced until next Periodic Route Update.

Upon reception of a Route Update message, each WMR must update its local routing table if necessary. If an entry is not refreshed for a certain amount of time, it is considered as not participating in the WMN any more. The node detecting this event sets the corresponding metric to infinite and schedules an RU message, Partial or Full dump, depending on the amount of concerned routing entries. If a node detects a broken link with one of its one-hop neighbor, each route entries which used this node as next-hop is removed from the routing table. A WMR can

act as a gateway to other networks. To announce a gateway capability, a flag is set in the RU message header. The default gateway corresponds to the one which has the best route.

Sequence numbers are used inside Route Update message and are used to determine if the received route is interesting. Sequence number value also corresponds to a route status, e.g. a valid route has an even sequence number while a broken route has an odd sequence number.

2.3.4.2 Client management protocol

The client management protocol uses two different tables, the first one is the *Local Client Table*, which lists the directly associated clients, the second one is the *Foreign Client Table*. It lists the foreign clients that are requested by a local client. Several message types are used to manage clients:

***Client Request* (CREQ)** message is used to locate foreign client on the WMN. If a node receives a CREQ and does not know the requested client, it forwards the request to its neighbors. Otherwise, it acknowledges the message with an ACK message.

***Acknowledgement* (ACK)** message is used to either acknowledge a CREQ or a CWIT message.

***Client Withdraw* (CWIT)** message is used to inform a WMR, that a requested client as moved. This message must also be acknowledged.

***Client Error* (CERR)** message is used to inform a WMR, that it has a wrong information about a client location.

Once a WMR has discovered a new client's IPv6 address, it must check whether the client was previously associated to another WMR. If the client was not formerly associated with another WMR, the WMR adds the corresponding entry in the Local Client Table. Otherwise, it must first ensure that the former WMR has detected the client's departure by sending a CWIT message and, after receiving the acknowledgement, adds the client entry in the table. The WMR receiving a CWIT message or detecting a leaving client must remove the corresponding client from its Local Client Table.

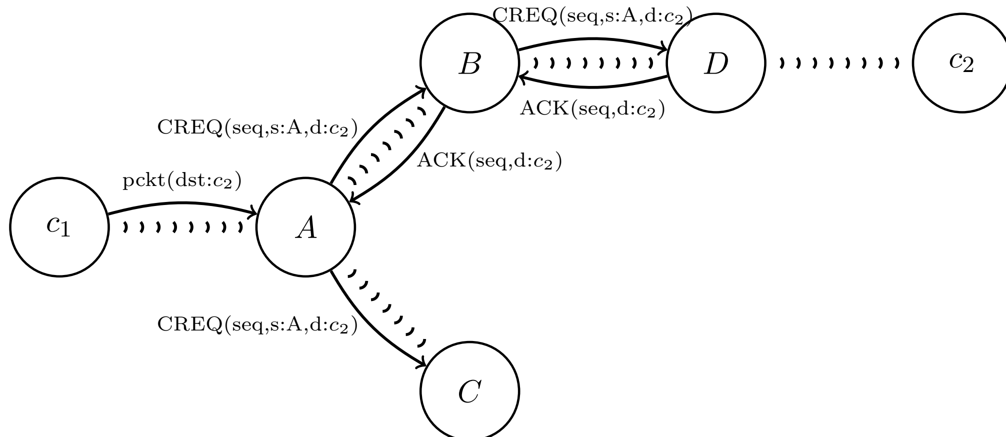


Figure 2.6: MeshDV – CREQ example.

If a WMR needs to locate a foreign client, it sends a CREQ message to all other WMRs. Fig. 2.6 illustrates a client request, where c_1 and c_2 are client nodes and others are WMR. The CREQ is initiated by node A, when it receives a packet from c_1 addressed to c_2. Node A does not know the foreign client c_2 because it is not present in its Local Client Table and in its Foreign Client Table. A CREQ must then be created and flooded through the WMR network. If a WMR receives a CREQ message it looks for the requested address in its Local Client Table. If the WMR finds it, it acknowledges the sender via a ACK message. Otherwise, it forwards the message to its neighbors (node B). In Fig. 2.6, node D acknowledges the initiator of the CREQ, node A, via an ACK message. If the originator of the CREQ receives an ACK message, it can add the client in its Foreign Client Table. A Foreign Client Table entry is removed when the route has not been used for a certain amount of time or by the reception of a CERR message. A CERR message is emitted when a WMR receives a packet for a client that are not in its Local Client Table.

2.4 Comparison

So far, we have seen the functionalities of various WMN routing protocols. Tab. 2.1 shows the properties that those protocols share or not have by comparing their theoretical results.

As it can be seen from Tab. 2.1, none of theses protocols specify any special security measures, nor *Quality of Service* (QoS) support. However, those protocols

	BATMAN	AODV	OLSR	MeshDV
Family	-	reactive	proactive	hybrid
Distributed	✓	✓	✓	✓
Loop free	-	✓	✓	✓
MANet	✓	✓	✓	-
WMN	✓	✓	✓	✓
Security	-	-	-	-
Multicast	-	✓	-	-
IPv6	-	-	-	✓
QoS	-	-	-	-

Table 2.1: Comparison between WMN routing protocols.

can readily be modified to include preexistent cryptographic coding protocols (e.g., IPSec, etc.). Therefore, allowing one to:

1. Protect the mesh cloud against malicious nodes, by authenticating each of them.
2. Ensure the confidentiality of communications.
3. Ensure the integrity of the network, by avoiding malicious message to be injected (e.g., TC messages in OLSR, RREP in AODV, etc.).

All protocols are distributed, therefore do not depend on a centralized node and thus can quickly react to topology changes.

OLSR is the only proactive routing protocols in this study. It has also the most in common with traditional wired routing protocols, such as, OSPF and IS-IS. OLSR is the most used protocol in WMN. Being a proactive protocol, OLSR uses power and network resources in order to propagate data about possibly unused routes. As said in Sec. 2.2.1, if too many changes occur causing a non-stable network, proactive routing protocols will exchange non negligible amount of data. On the other hand, as each node knows the whole topology, delays are minimized when a route is acquired. Being a link state routing protocol, each OLSR node needs to compute optimal path in the network, requiring a large amount of CPU, depending on the size of the network and on the embedded hardware. By using MPRs to broadcast topology information, OLSR removes some redundancy, which may lead to problems if the network has a high packet loss rate.

The main advantage of AODV is the inherent property of all reactive protocols: routes are established on demand. So the bandwidth used to exchange control data

is lower than OLSR. However, multiple RREP in response to a single RREQ can lead to a heavy control overhead. Delays also depend on the amount of time between a RREQ and the first RREP received, which can be important depending on the size of the network. Another advantage of AODV is that destination sequence number are used to find the latest route to the destination. However, this can lead to inconsistent routes, if some intermediate nodes, with invalid topology information, response to a RREQ.

BATMAN is the most recent developed protocol, and is still a work under progress. BATMAN does not have elements of classical routing protocols and, consequently it cannot be classified in a routing protocol family. However, its main advantage is its simplicity, as it relies on a simple message. As BATMAN is still under development, it has several flaws that needs to be corrected:

1. BATMAN does not contains any loop avoidance mechanism, nor any loop detection.

2. BATMAN does not aggregate messages into packets, this may lead to a significant cost in the link-layer technologies (with an important per-frame overhead).

3. BATMAN convergence is exponential in the diameter of the network in the presence of packet loss.

MeshDV is an hybrid protocol as it is both proactive and reactive. It is designed to take full advantage of WMNs' architecture. It is able to form a mesh backbone integrating clients' mobility management. MeshDV is an IPv6 only network, and takes information inside the data link layer to perform route selection. Nevertheless there are still problems on the mobility management. If a client moves during a communication and if the WMR association changes. The old WMR, where the client was connected, does not know that the client has moved. This may lead to losses of packets. This is due to the fact that MeshDV relies on the data link layer which takes some time to detect changes.

2.5 Conclusion

Routing protocols behaviors are very disparate, depending on the network state some protocols may react in a better way that others. Choosing the right protocols for a

network is not very simple, one may try classic protocols such as OLSR and AODV, that are standardized by the IETF. Other could choose more recent protocols such as BATMAN, that encounter a tremendous growth over the last years and may give better result that others. And finally, one could choose MeshDV which deals with client movement, which is very interesting for every WMNs.

Chapter 3

Monitoring

The main goal of network management is to ensure the quality of services the networks provides. To achieve this, the network maintainer must monitor and control the different connected elements in the network.

We are especially interested in one aspect of the network management: network monitoring. This aspect is concerned by analyzing the behavior and the status of the network nodes that automatically generates the topology. Accuracy and efficiency in the monitoring is therefore essential and critical for various functions of network management.

To implement the network monitoring part, three architecture are possible (a combination may also be envisaged):

The management plane is based on queries pooling (queries on each nodes) to monitor the network.

The control plane implements the routing protocols to listen to the messages exchanged between the routing protocols nodes.

The data plane supports packet-forwarding functionality, such as destination-based forwarding, filtering, and tunneling, in order to analyze the existing state of the network.

The control plane and the management plane are two related notions as they achieve the same goal: an accurate network state monitoring. On the other hand the data plane mechanism is based on probing to obtain an accurate network state. The most common probing technique is to execute ping between two pairs of nodes to

verify the status of the route between them. A well known data plane management software is the *Cisco IOS IP Service Level Agreements* (SLAs) [ips09]. It allows the network maintainer to perform measurement such as delay jitter, congestion, availability of path, etc.

As we have seen in Chap. 2, there exist several completely different routing protocols families. Theses protocols have disparate behaviors and, thus, it is difficult to deploy a monitoring system handling all the possible protocols.

In this work, we choose to develop an OLSR (see Chap. 2, Sec. 2.3.3) monitoring tool. The main reason is that OLSR is currently widely used and well tested. It is highly portable [A. 09], i.e., it exists implementations on the three main OS: Windows, Mac OS X and Linux.

3.1 Related works and existing solutions

Monitoring and analyzing of routing protocols have become areas of research recently. In wired networks, numerous studies appeared on the subject of OSPF monitoring [SGG+02, SG04, SIG+02], in which Shaikh et al. described a couple of techniques to achieve a good tracking of the OSPF topology. This work has been entended to protocols such as BGP and there are more commercial products products used by ISPs to perfom this monitoring [Pac09].

In the wireless mesh network domain, lot of works has been achieved. Sailhan et al. present monitoring system of self-organized sub-systems [SFQ+07]. The key idea is to use a cluster-based hierarchical structure, maintaining monitoring data in a distributive manner. The distributed structure is then used to propagate and aggregate the monitoring data, resulting in a reduced bandwidth usage.

Gupta et al. study the issue of efficient monitoring in WMN, mostly the impact of the monitoring overhead in the user data traffic [GMCN08]. They observe that it is crucial to have the appropriate technique (i.e., the right frequency, the right monitoring manner, etc) for an application in order to maintain the balance between minimizing bandwidth consumption and measurement data accuracy.

Naudts et al. describe a planning tool to help rescue teams by deploying a WMN [NBB+07]. To offer a real-time overview of the network, they implemented, as done by Sailhan et al. [SFQ+07], a distributed way to gather data, based on existing probing techniques.

Nanda and Kotz describe, as Naudts et al. [NBB+07], a solution providing a communication infrastructure in a real environment [NK08]. They describe *Mesh-Mon* a solution, where mesh nodes, embedded in each emergency vehicle, form the wireless mesh backbone. Mesh-Mon is not, as previous existing studies, a centralized solution. Nodes are designed to actively cooperate, predict, diagnose and resolve network problems in a scalable manner.

3.2 Objectives

Wireless mesh network monitoring requires a trade-off between accuracy of real-time information and measurement overhead. Network monitoring may introduce overheads, which can interfere with the end user data flows and thus decrease the network performance.

We want to design a WMN monitoring system that meet the following objectives:

Provide real-time and efficient monitoring of routing protocols behaviors
: This tracking can be used to identify problems and troubleshoot them. It can also be used to validate the configuration of the routing protocol. And it allows the maintainer to visualize the network topology.

Limit the amount of monitoring messages
: The monitoring tool has to provide a maximum amount of information on the network status, while minimizing the network resource usage.

Accurately validate the network state viewed by the routing protocol
: As explained in Chap. 2, Sec. 2.1, WMNs have highly dynamic characteristics. Sometimes, routing protocols do not represent accurately the topology, due to this dynamic problem. We thus need a tool that accurately represents the topology to identify problems that are not covered within routing protocols.

In the following we discuss the recent work that has been done about these three objectives.

3.2.1 Real-time topology monitoring

Having a good real-time topology monitoring is the central part of each possible network monitoring. Such real-time monitoring could be used to:

1. Easily identify problems and their sources, e.g., operators could quickly identify a link that is flapping (i.e., up and down constantly) and then troubleshoot it.

2. Accurate presentation of the network topology viewed by routing protocols.

3. Validation of the routing protocol configuration, for maintenance or traffic engineering.

Such an objective has already been discussed in the case of the OSPF routing protocol [SGG+02, SG04]. Two different approaches to solve this problem: a proactive one (management plane) and a passive one (control plane).

Theses two solutions involve a special device storing information about the current and past network status. We call this device the *topology server*. This device is the central piece of information and thus must be reliable and robust, ensuring that information is always available.

3.2.1.1 The Management plane approach

This solution involves communicating with each routers via the SNMP protocol [CFSD90]. SNMP is a simple way to monitor all types of network. The SNMP MIB defines public variables that can be read remotely. These variables contain information such as the available memory on the node, the default route, etc. SNMP allows one to define TRAPs, that are triggered upon changes. A SNMP TRAP is a message which is initiated by a node and sent to the requested system, so that when a change occurs, e.g., a neighbor is up or down, a TRAP is sent to the management station. A TRAP message is sent as an UDP datagram.

With these two mechanisms (TRAPS and queries), the topology server can keep track of the topology. To achieve this, the topology server must register for TRAPs and query all nodes to construct the entire topology. Fig. 3.1 shows the network architecture of such a solution. Each arrow corresponds to a possible SNMP query/TRAP. The number of queries is thus proportional to the number of nodes in the network. As SNMP runs on top of UDP, the queries are not reliable and we need thus to support acknowledgment messages to avoid this issue.

To be able to query and configure TRAPs on routers (all black dots in Fig. 3.1), the topology server must have information about them, e.g., IP addresses mainly. It must then run a network discovery procedure, which is used to discover all routers in

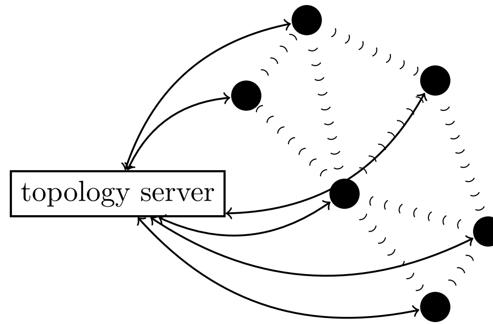


Figure 3.1: Network Architecture with active monitoring.

the network. This process starts with a single router configured as a seed. The process extracts OSPF variables from the router's MIB, e.g., neighbors OSPF routers, interfaces, the OSPF area, etc., and then restarts the process with the newly discovered routers until there are no more routers to discover.

3.2.1.2 The control plane approach

This solution is quite simple: a special node is added to the network and does not interfere with previously existing nodes, i.e., it runs the routing protocol, but does not announce itself as an OSPF router. Its only assignment is to listen the routing protocol packets and forward them (or inform about changes) to the topology server.

Fig. 3.2 shows the network architecture in case of packet sniffing, the white node represents the listening node. Compared to the previous solution (Fig. 3.1), less bandwidth is used to retrieve information. To increase reliability, we can add as many special nodes as we want. However there is an obvious trade-off between reliability and bandwidth usage.

This approach only works if the routing protocol is link state, e.g., that each node knows the whole topology.

3.2.1.3 Evaluation

Shaikh et al. evaluate their two approaches, in terms of two operational issues [SGG+02, SG04]:

The ease of deployment

In the control plane approach, there must be a physical link between OSPF routers and the listening node. So it must be simpler to deploy the management plane approach, as it only requires a simple box: the topology server.

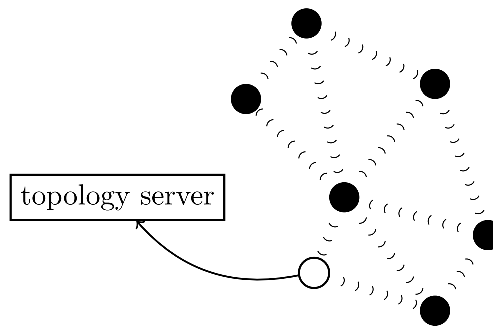


Figure 3.2: Network Architecture with passive monitoring.

The overhead

In the management plane approach, all routers have to respond to SNMP queries. They need also to generate TRAPs when the topology changes, e.g., when a new OSPF node is up each router generates a SNMP TRAP. On the other hand, the control plane approach quietly listens to the network and sends messages only when there is a change. The latter solution is then better in terms of generated overheads.

Finally, Shaikh et al. confront the two approaches in terms of reliability and timeliness. The informing process can be divided in two parts: the generation of the herald message and its propagation. The latter is the major component of the total delay before the change reaches the server. The propagation mainly depends on the path taken by the message from the source of the change to the server. Since OSPF messages are flooded, they take the shortest path from the source to the server, while TRAP messages take the minimum weighted path to reach the server. The latter path, may thus, not be the path with minimum delay or the minimum number of hops.

Simulations compare the two approaches in terms of the four points discussed above. The simulations show that the superiority of the control plane approach, mainly in terms of reliability and robustness even in face of drastic network conditions. The main issue with the management plane approach is the lack of reliability of SNMP TRAPs. On the other hand OSPF messages are flooded from a reliable manner, which ensures that they always reach the server. In addition this flooding provides multiple paths increasing the robustness of the information. This robustness provides that they reach the server even if the network is in a transient state, provided that the network remains connected, while the normal forwarding of mes-

sages could fail.

3.2.1.4 In the OLSR case

Based on this research, we need to decide whether the management plane or the control plane approach is the best to monitor OLSR behavior. We can review the comparisons made above within the case of OLSR:

The ease of deployment

In WMN, we are in a wireless world, so there is no problem to include the listening nodes. We can easily include them anywhere in the network, providing that they can reach the topology server in some way.

The overhead

As this is our second monitoring objective (see Sec. 3.2.2), the overhead needs to be extremely low. Compared to wired networks, wireless networks are highly dynamic and changes can occur very often. The management pane approach generates a lot of TRAPs, for each change, depending on the monitoring configuration, in the worst configuration possible it generates n TRAPs for a n nodes topology. While the control plane approach only sends one message for each change and can combine several changes in a single message, allowing a limited overhead. The latter solution is thus the better one.

The other comparisons (i.e., reliability and timeliness) remain valid for OLSR, as it is a Link-State routing protocol, as OSPF.

We thus choose the control plane approach to carry out the real-time topology monitoring.

3.2.2 Limit the amount of monitoring messages

WMN is known to be capacity limited [AWW05, JS03], as it is affected by many factors, such as: network topology, traffic patterns, node density, etc. In the case of WMN monitoring, we care about the user traffic, and as it is capacity limited, we do not want to interfere with it.

To avoid such inconvenience, we want to limit as much as we can the overhead. In Sec. 3.2.1, we discussed the way to acquire the network topology, we chose a solution by already taking into account the need to limit the overhead. Nevertheless, we can go further and try to limit even more the monitoring overheads.

Gupta et al. discussed the impact of the monitoring overhead in the case of the management plane approach (see Sec. 3.2.1.1), in WMN on the forwarding of user data and a threshold-based monitoring [GMCN08]. They also evaluate the impact the impact of the frequency of reporting the monitoring data on the end-user's performance.

The threshold-based monitoring objective is to report data to the topology server only when a given event occurs. This allows one to reduce the monitoring traffic.

An other way to not decrease the performance of the network is to report data at an appropriate frequency. Frequency is an important parameter in each monitoring framework. Reporting data at a high frequency allows us to maintain an accurate image of the network. However, this approach suffers from high monitoring overheads. On the other hand, if a lower frequency is used, the overheads vanish, but the image of the network is not as accurate as with a higher frequency. The frequency should be selected appropriately to not decrease the desired functionality.

Gupta et al. demonstrate that even a small amount of monitoring overheads can cause a large degradation on the end's user performances [GMCN08]. Even if this result is based on the management plane approach, it increases the need to limit as much a possible the overhead. To evaluate the impact of the monitoring frequency, the authors performed two tests: One with each node reporting data every second and another every ten seconds. The simulation results showed that the accuracy is high, as expected, when the data are sent every second. Nevertheless, the accuracy drops a little when nodes report every ten seconds, but is still reasonable. And as expected, reporting data every ten seconds achieves slightly better performance in terms of end-to-end delay, throughput, and packets loss than the other case.

3.2.3 Validate network state viewed by Routing protocols

In Chap. 2, Sec. 2.3, we detailed several WMN routing protocols. However, nowadays, hundreds of proposed routing protocols, some of them being standardized and used for many years (OLSR, AODV).

As these protocols are running on top of a highly dynamic network, they could not represent the accurate state of the network. This could be caused by:

1. The architecture of the routing protocol, that is not designed to perform well on different situations.

2. An erroneous configuration of the routing protocol, causing its bad performance. E.g., if the time between OLSR Hello messages is too high, then due to the low reliability of the wireless link layer, these messages could be lost and so, the link will be unusable.

A tool is then needed to validate this network state viewed by the routing protocols. We do not discuss this validation as in Chap. 4 a tool to cover a network topology with paths to validate its state is entirely dedicated.

This tool is based on the data plane approach monitoring. It can give more information than just validate the topology. About paths, it can give information about: the current average delay, the congestion level, etc.

3.3 Characteristics to monitor

In addition to the three main objectives listed in Sec. 3.2, an efficient monitoring needs a lot of useful information to be provided. We describe main characteristics of WMNs that can be measured, in addition to regular monitoring characteristics (e.g., as in basic network monitoring).

Theses characteristics can be retrieved from difference ways, the simplest being the use of SNMP, with a monitoring frequency and a threshold-based monitoring as in Sec. 3.2.2. But a more efficient way is to do as in [SFQ+07], where Sailhan et al. auto-create clusters of node that share information about monitoring and then combine these informations to send it to the topology server.

List of connected clients

Routers in a WMNs may be heavily requested by clients (i.e., it has a lot of connected clients). It could be interesting to know the number of clients connected to a WMR. From this information the network maintainer can reorganize the network topology to be more efficient.

Signal-to-Noise Ratio (SNR)

The SNR is an information of the noise in the channel between nodes. It gives an overview of the link quality and capacity. A low SNR means that two WMR are too far from each other, so that the link quality is very low.

Packets loss

In regular wired networks packets loss are due to congestion. In WMNs, a packet can be lost due to other aspects, as connectivity loss (i.e., low SNR), a moving client, etc. It could be interesting to know the loss causes.

Average delay/load on each link

Link capacity can vary in time, it can then be interesting to have a temporal overview of the delay/load compared to the current link capacity.

Monitor much important nodes

Gateways are always crucial, thus we need more informations about them than other nodes.

Traffic proportion by nodes

This is related to the previous item. To be able to balance, in terms of proportion of traffic, more efficiently the network, we could monitor the amount of data transiting in each nodes.

From theses monitor characteristics, the network maintainer can take useful actions to resolve the underlined issues. Some action could also be taken within the monitoring tool, as *Traffic Engineering*. E.g, the metric of a link can be changed, manually or dynamically with *Tomogravity* [Var96].

3.4 Conclusion

Overheads are a key topic in wireless monitoring, we need to deal with it. We chose a simple way to acquire the current topology: sniffing the routing protocol messages. In addition to give an accurate state of the network, it avoid using too much message and thus increase the overhead. We also talked about the monitoring frequency, a well-chosen frequency allows to retrieve an accurate state of the network while not interfering with the client traffic.

We also discussed about the impossibility to routing protocol to sometimes not accurately represent the topology, especially in the case of WMNs. To deal with this issue, we define, in the following chapter, a new theory: the *topology coverage*.

Chapter 4

Topology coverage

In this chapter we present a tool to confirm the current network status. This tool is based on the data plane approach to monitoring (see Chap. 3).

We define a network status as a set of link status. Therefore, to validate a network status we need to confirm, one by one, each link of this set. A link status is valid if it is functioning, i.e., packet can successfully pass through it. Hence, a network status is said to be valid if each of its link is.

4.1 Graph theory

From the graph theory we can extract one notion: the *path decomposition* [Don80]. A path decomposition of a graph $G = (V, E)$, a collection of edges E and vertices V, is a list of paths such that each edge appears in exactly one path of the list. From this notion derives two concepts closely related: the $\{P_l\}$*-decomposition* [Pyb96, MqCh06] and the (a, b)*-decomposition* [TR], that takes into account a path length constraint.

The graphs considered in these two problems are finite, connected and undirected. The length of a path is defined by its number of hops. The set of edges and vertices of a graph G are denoted respectively as $E(G)$ and $V(G)$.

We can define the $\{P_l\}$-decomposition and the (a, b)-decomposition as follows:

Definition 1 ($\{P_l\}$-decomposition)
The $\{P_l\}$-decomposition of a graph $G = (V, E)$ is a partition of $E(G)$ into paths P_l of length $l - 1$.

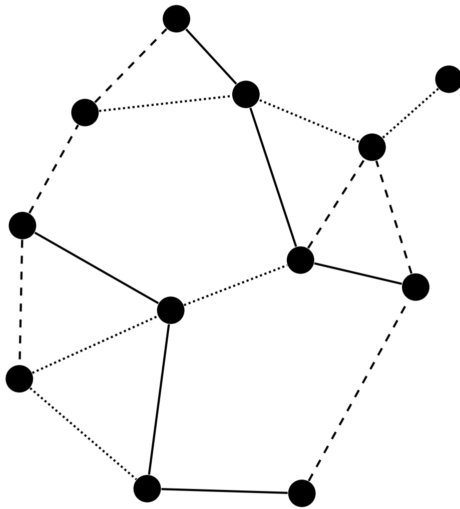


Figure 4.1: The P_4-decomposition for an exact cover of the graph.

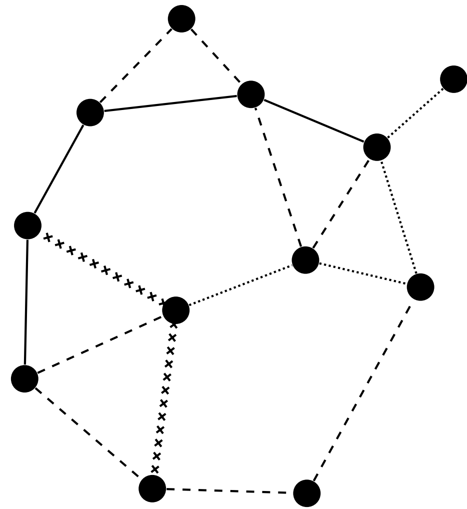


Figure 4.2: The $(2,4)$-decomposition for an exact cover of the graph.

Definition 2 ((a,b)-decomposition)
Given 2 integers $a \leq b$, an (a,b)-decomposition of a graph $G = (V,E)$ is a partition of $E(G)$ into paths of length between a and b.

There are only few differences between these two decompositions. The (a,b)-decomposition allows variable length paths, while the other allows only fixed path length. As a result, the $\{P_l\}$-decomposition can be viewed as a particular case of the (a,b)-decomposition, being in fact a $(l-1, l-1)$-decomposition.

Fig. 4.1 and Fig. 4.2 show examples of a $\{P_4\}$-decomposition and a $(2,4)$-decomposition, respectively, of the same graph.

Differences are significant, between the two figures, mainly because the $\{P_l\}$-decomposition is dependent of the number of edges. We can readily see that the parity of l determines which graph can or can not be decomposed:

Proposition 1
A graph G of size m ($m = |E(G)|$) can be $\{P_l\}$-decomposed if and only if l and m parity is opposite.

In Fig. 4.1, we can see that the graph has 17 edges, if it had a even number of edges then it would have been impossible to find the decomposition. However, the $(2,4)$-decomposition is still possible even if one edge is added or removed.

On the other hand, an (a, b)-decomposition can be trivial. Indeed, if we consider problems with $a = 1$, then the decomposition admits a trivial solution by simply taking one path per edge.

Pyber shows that every connected graph on n vertices can be covered by at most $n/2 + O(n^{2/4})$ paths [Pyb96]. Ming-qing and Chang-hong show that a $\{P_3\}$-decomposition of a graph with order n and size m, if it exists, can be computed with a polynomial time algorithm $O(nm)$ [MqCh06].

Teypaz and Rapine conjecture that the (a, b)-decomposition problem on an unspecified graph is $\mathcal{NP}$-complete if a is greater than three [TR]. They also show that if a graph is not traversable, the minimum (a, b)-decomposition problem is $\mathcal{NP}$-complete. A graph is traversable if all its arcs can be traced in exactly one movement without lifting a pencil.

4.2 The topology coverage problem

From the notions explained in Sec. 4.1, we can derive a new problem: the *topology coverage problem.*

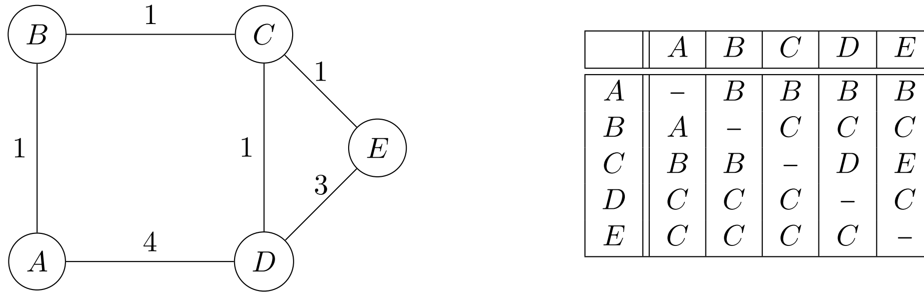
Definition 3 (Topology coverage problem)
Let G be a network topology graph, we define the topology coverage problem as the minimal $(2, b)$-decomposition of G.

The topology coverage problem is an adaptation of the (a, b)-decomposition problem such that:

1. The number of paths is as small as possible so that we avoid to overload nodes,
2. We cover only useful links in the networks, i.e., link that carry data,
3. We avoid trivial solutions, i.e., no simple one-hop paths, because among other things, they are already tested by the routing protocol.

From item 3, we limit the length of paths to be $\in [2, \infty[$.

Recall from Sec. 4.1, that the minimal (a, b)-decomposition cannot be found with a polynomial time algorithm. This issue leads us to consider a relaxed problem.



	A	B	C	D	E
A	–	B	B	B	B
B	A	–	C	C	C
C	B	B	–	D	E
D	C	C	C	–	C
E	C	C	C	C	–

Figure 4.3: Shortest path forwarding – example topology and corresponding forwarding tables. Each row of the table give the next-hop to reach a node.

4.2.1 A relaxed problem

To find a relaxed topology coverage problem, we chose to consider only a limited set of possible paths to cover the graph.

We build this set of possible paths based on the IP behavior: the *shortest path forwarding*. The shortest path forwarding is based on the weight of each edge of the graph. Based on this approach, each router compute the shortest path for each route of the network, i.e., each other nodes.

Once a router has the information about the path trough the network it can build its *routing table*. The routing table contains information about the next-hop, i.e., the neighboring router, where it needs to forward the packet to join a destination. This allows the IP packets to follows fixed paths to join a destination.

This shortest path forwarding induces a limitation of the links (edges of the network graph) usage, as it exists only one path between each node of the network. There is no need for the topology coverage problem to cover non-used links. In fact some of the paths may be used by network maintainer to configure or control one node, etc. However, the major part of the traffic only pass through paths used by the shortest path forwarding.

Based on that information, we define our limited set of paths to be the possible routes in the network. For a n nodes topology, there are $n-1$ paths from one node to the others, so a total of $n(n-1)$ possible routes.

The *All-Pairs Shortest Paths* algorithm (APSP) is used to compute the set of possible routes in the network. There exists several implementations, e.g., the *Floyd-Warshall* algorithm [Flo62] that have a time complexity of $\mathcal{O}(n^3)$.

This problem simplification has two advantages: it simplifies the overall computation complexity and respects item 2 from the problem objectives.

Fig. 4.3 shows one example of the shortest path forwarding and the interest of using a limited set of paths. The routing tables are build upon one of the many possible routing protocols (OSPF, OLSR, etc.). In the figure, links $A \leftrightarrow D$ and $D \leftrightarrow E$ to are not used to forward packets. Let us take the example of the path from A to D. To join D, A has several possibilities, but the least costly path is to join D through B and D, where the total path cost is thee. The simple path, taking the link, between A and D cost more, it has a cost of four. This shows that all links in the network are not useful, some of them might not be used, therefore, there is no interest to test their status.

We could choose a source routing approach instead of the shortest path forwarding. In this case, packets define their path through the network. This allows more flexibility in the coverage, as it is possible to build all possible paths. With this solution, we can compute the coverage of the topology graph and then introduce it to the network environment, without taking care of the network restriction. However, source routing is considered harmful by *Internet Service Providers* (ISPs), as it allows anyone to partially or completely specify the route a packet takes through the network. Therefore, the source routing is deactivated almost everywhere. This is the case for the IPv4 protocol but also for its successor: *IPv6*. The latter was designed to contains a source routing header extension (*RH 0*), however the severity of the source routing threat is considered to be sufficient to warrant the deprecation of the header extension [ASNN07].

An other simplification could be to not limit the problem to disjoint paths, and allow the algorithm to over-cover some edges, i.e., edges belonging to more than one path of the decomposition. Nevertheless, we want to have a minimal over-coverage of edges, i.e., we do not want to congest links inside the network.

From these simplifications we can reformulate the topology coverage problem to be this time a computational problem and not a graph problem anymore.

Definition 4 (Topology coverage problem - computational)
Let $G = (V, E)$ be a network topology graph, where $E(G)$ is defined by useful links, and Φ a set of feasible paths of this graph G such that some edges $e \in E(G)$ belong to at least one path of Φ.

The topology coverage problem is to find the minimum subset Φ' of Φ such that every edge e from $E(G)$, belonging to at least one path of Φ, belongs to at least one path of the Φ' subset.

In this case, we want to find the best possible solution among the set of all possible solutions to a search problem. In the following section we explain the algorithm to solve this optimization problem.

4.3 A typical test

In the previous section we defined the goal of the topology coverage problem but we did not explain how to execute the validation of a network state. As said in Definition 4, the problem is to find a subset of paths. We need to test each path in the subset in order to validate the current network state.

To test a path, we execute an *ICMP Request/Reply* [Pos81], also called ping, from the node starting the path to the ending node. As pings are based on IP packets, they follow the shortest forwarding path. Therefore, they respect the intermediate passage of the paths.

We only consider undirected graphs, pings tests links in both direction. First, in one direction, with the ICMP Request to the destination node and second, back, with the ICMP Reply message to the requesting node. It may happen in some cases that the reply does not follow the same path as the request. E.g, it is the case when the *equal-cost multi-path* (ECMP) routing [Hop00] is used. The ECMP routing is a strategy where the choice of the next-hop to a single destination is based on multiple best paths. Best paths are defined by the same metric cost. To avoid such unwanted issue, we chose to make the hypothesis that we are in a network that does not use such strategy.

To understand how ping test works, let us look at an example based on Fig. 4.3. If we choose to test the path: $A \leftrightarrow B \leftrightarrow C \leftrightarrow D$, a ping is executed between node A and D. Following their routing table, the ping packet passes through B and C before reaching D, and goes back to A following the same path.

In order to validate the state of the network, all paths of the subset must be validated. The device that computes the subset must give to each node that starts a path in the subset one or more destination to be probed (depending on the number of paths starting at each node). Upon the reception of the request, each node starts pinging its destination nodes. Pings results can be the following:

Everything OK

In this case, the reply is received meaning that the path is fully functional in

both direction.

Timeout

This means that there was no reply to the ping (the cause is mainly the loss of the ping) and means that maybe one link is not up or is congested.

Destination Unreachable

This means that the current routing table, on this node or along the path, does not match the topology graph given as parameters to the algorithm.

Destination Down

This means that the route is still valid but the node is down.

The two latter results are extracted from an ICMP message. If a middle router detect that the destination of the ping is unreachable or down, it sends back a Type 3 ICMP message. When a ping fails, the result is sent back to the ping request server to be further analyzed.

4.4 Centralized Algorithm

A trivial algorithm could be to enumerate all possible subsets and select the best of them, i.e., the one with the smallest possible paths. The problem with such a solution is that it is $\mathcal{NP}$-complete and thus not practicable.

A better solution would be to use an heuristic search methods [RN03], such as *A** or *Greedy Best-first search*. Of course these algorithms do not always give the best solution, nevertheless, providing a good heuristic, it will give a solution close to the best one.

We choose to implement a *Greedy Best-first search* like algorithm, as it gives a simple way to prune unwanted paths during the search process.

Algo 4.4.1 defines our implementation of the topology coverage problem. It contains two external procedures:

AllCovered return true if the current set of paths is a terminal state, i.e., a state where each edge is covered.

H is the heuristic function that gives the cost of the current set of paths.

The cost of the current set of path, of course, depends on the heuristic, but must follow some rules. The cost must always be greater or equal to zero and when the

Algorithm 4.4.1 The topology coverage algorithm.

Require: *graph*, a graph representation of a network topology
Require: *routes*, a set of possible routes in the graph, subject to $\forall r \in routes$: $2 \leq \text{Length}(r) \leq \infty$
Ensure: *paths*, a set of paths, subject to $\forall p \in paths : p \in routes$, representing the coverage of the graph

```
 1: paths ← {}
 2: while not AllCovered(graph, paths) do
 3:    min ← ∞
 4:    path ← ∅
 5:    for all p ∈ routes do
 6:       cost ← H(graph, paths ∪ {p})
 7:       if cost < min then
 8:          path ← p
 9:          min ← cost
10:       end if
11:    end for
12:    paths ← paths ∪ {path}
13:    routes ← routes\{path}
14: end while
15: return paths
```

current set of path is terminal (i.e., AllCovered returns `true`) the cost must be exactly zero.

4.4.1 Algorithm behavior

Fig. 4.5 depicts the temporal search evolution of the algorithm, for a very simple case. We can see that the algorithm is a tree search.

The algorithm works as follows: first, it starts with an empty set of paths (root of the tree in Fig. 4.5). At each step it generates the successors of the set, i.e., takes one of the path in *routes* and generates a new set containing one more element. It then evaluates the cost (line 6) of each set and continues the computation with the less costly one. It then checks whether or not it is a solution (line 2). If it is, the algorithm ends returning the current set of paths. Otherwise, the algorithm repeats the computation step until a solution is reached.

As the depth of the tree is not infinite, the depth is lower or equal to the number of paths in *routes*. Consequently, the algorithm will always finish. In the worst case the solution comes when all routes are chosen to cover the graph.

Fig. 4.5 shows exactly the computation steps for a simple graph (see Fig. 4.4) with four possible paths of length greater than 2. The possible paths are: $p_1 := n_1 \leftrightarrow$

$n_0 \leftrightarrow n_3$, $p_2 := n_3 \leftrightarrow n_0 \leftrightarrow n_4$, $p_3 := n_2 \leftrightarrow n_0 \leftrightarrow n_3$ and $p_4 := n_1 \leftrightarrow n_2 \leftrightarrow n_4$. In this case there are four children to the root of the tree. From the evaluation of the cost, it chooses the second child to continue the computation as it has the minimal cost. In Fig. 4.5, the costs are randomly chosen, nevertheless in real cases it depends on the heuristic used. The computation continues until the cost reach zero, the node of the tree at this time is a solution. It is interesting to notice the pruning that is done at each level in the tree. At the first level, the algorithm prunes tree subtrees, two at the second level, etc. The amount of pruning depends on the node level degree, e.g., at level i there are $i - 1$ subtrees pruned. This allows the complexity of the algorithm to be polynomial and not exponential, if we traverse the whole tree to find a solution.

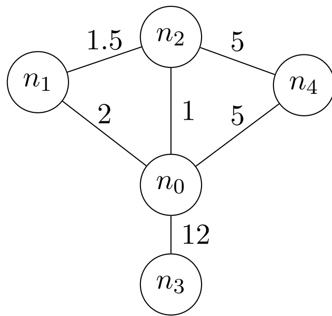


Figure 4.4: Example topology graph with four possible route.

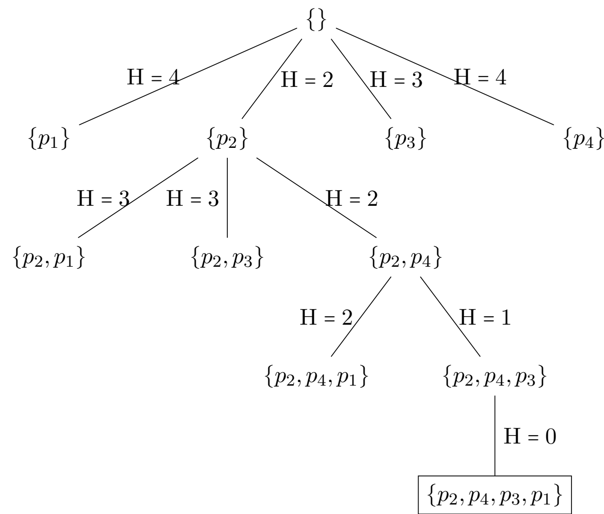


Figure 4.5: Algorithm search example for Fig. 4.4.

4.4.2 Algorithm complexity

The true algorithm complexity is hard to define, as it mainly depends on the heuristic used to compute the solution. However, for the worst case scenario where the solution is all the possible routes, we can specify the complexity.

For a graph G of order n ($= |V(G)|$) and size m ($= |E(G)|$), we define r to be the number of routes used to cover a graph: $r = |routes| = |\Phi'| \leq |\Phi| = n(n-1)$.

The complexity of the algorithm depends of the complexity of the procedures: Terminal and H. Their time complexity is $\mathcal{O}(m)$ as they must travel through all edges to check whether each edge is covered.

In Algo 4.4.1, we see that the while loop, line 2 to 14, is executed at worst $r + (r-1) + (r-2) + \ldots + 2 + 1$ times. It depends on the complexity of the '**for**' loop which is the number of element in *routes*. At each step, The elements in *routes* decrease by one. $r + (r-1) + (r-2) + \ldots + 2 + 1$ can be factored as $\frac{1}{2}(r^2 + r) \leq \mathcal{O}(r^2)$. Thus, in the worst case, the time complexity of the algorithm is $\mathcal{O}(mr^2) \leq \mathcal{O}(mn^4)$.

4.4.3 Possible heuristics

In this section we will look at possible heuristics for the topology coverage problem. A good heuristic is essential to give the best results from search problems. However, finding the best possible solution, i.e., the smallest possible path set, is not our only goal. We need to care about the over-coverage of edges, as we discussed in Sec. 4.2. Two heuristics candidates can be:

h_{basic} tries to limit the number of paths by counting the number of edges not yet covered.

$h_{overcover}$ tries to, in addition to limiting the number of paths, limiting the over-coverage of edges.

The goal of these two heuristics are different, the first one only cares about quickly finding a solution, while the latter, tries to avoid nodes overloading and congestion.

The first heuristic is admissible as it does not over-estimate the cost to reach the goal. On the other hand, the second heuristic over-estimates the cost, as it counts the number of edges over-covered. In this case, the algorithm is not optimal anymore. It is not a real problem, we do not need optimal solutions, we need to find, in an acceptable time, a satisfying solution. A satisfying solution will not over-cover the edges, so we need to control this parameter.

4.4.4 Evaluation

The following section describes the simulations and the results obtained by implementing algorithm 4.4.1.

Sec. 4.4.4.1 discusses the different tools used to generate topologies and the underlining topology models. And Sec. 4.4.4.2 shows the results obtained with the algorithm.

The algorithm was implemented in *Python* [pyt09]. Python is a high level interpreted programming language. The syntax is minimalist while the available libraries are comprehensible. We used the *NetworkX* library [Hag08], which is a package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Especially this package contains a good implementations of a graph structure and algorithms such as the *All-Pairs shortest path* and the *Floyd-warshall* algorithms [Flo62], used to compute routes in the network.

The simulator source code is available in Appendix B.

The setup for the simulations are computers with Intel Core 2 Quad CPU Q6600 @ 2.40GHz with 2GB DDR2. These computers runs the *CentOS* distribution of *GNU/Linux*, with *Python 2.4.3*.

4.4.4.1 Topology models

In this section we explain *Waxman*, the model that is used to simulate algorithm 4.4.1. This model belongs to the *Random graph generators* family.

Random graph generators

The concept of random graph was first introduced by Erdös and Rényi in 1959 [ER59]. The basic idea of random models (sometimes called pure random models) is that a graph can be built completely randomly to model networks. The *Erdös-Rényi* (ER) random graph is very simple: it associates a constant probability to the creation of a link between any pair of nodes.

The Waxman random graph [Wax88] is a small variant of the ER model. This generator assigns coordinates to nodes onto the coordinate plane. In the Waxman model, the probability to create edges is a function of the Euclidian distance between nodes.

Random graph generators do not explicitly attempt to reflect the structure of real networks [Quo09]. Nevertheless, they are attractive for their simplicity and are commonly used to study networking problems.

4.4.4.2 Results

In this section we evaluate the effectiveness of algorithm 4.4.1 among the different topology models, presented in Sec. 4.4.4.1. We study the influence of several param-

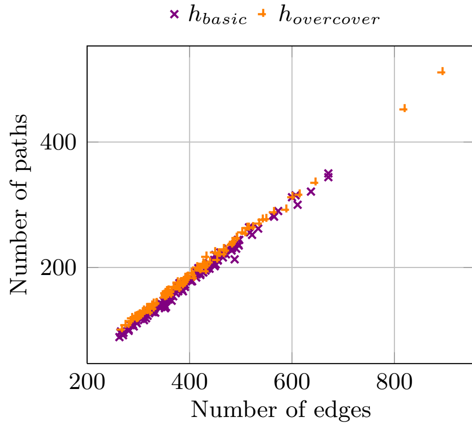


Figure 4.6: Waxman – Number of paths used to cover a graph when the number of edges varies and the number of nodes is fixed.

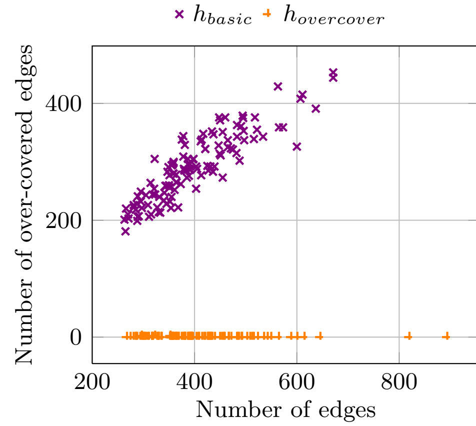


Figure 4.7: Waxman – Number of edges covered by at least 2 paths, i.e., over-covered, when the number of nodes is fixed.

eters of the algorithms: the number of nodes, the number of edges and the cluster size.

The Waxman parameters used to generate topologies are: $\alpha = 0.15$, $\beta = 0.2$ and $m = 2$.

We first evaluate the effectiveness of algorithm 4.4.1 at properly cover the topology.

Coverage ability

The most important result to evaluate in the case of this algorithm is its capacity to cover correctly the topology graph. The different criteria for a good coverage are the following:

1. A minimal number of paths should be used.
2. An acceptable computation time.
3. Edges should not be too much over-covered.
4. The number of probes sent by each node should be limited, i.e., we do not want to overload a node with tests requests.

Fig. 4.6 and Fig. 4.7 show results for the two possible heuristics, $h_{overcover}$ and h_{basic}. Fig. 4.7 shows that, as expected, the number of over-covered edges is 0 or heavily limited for the $h_{overcover}$ heuristic. On the other hand, the h_{basic} presents

poor results, indeed, almost every edge is over-covered. However, the two heuristics show both the same results in Fig. 4.6. They generate, on average, the same number of paths for similar topologies. In addition to that, it seems that the number of paths is almost equal to half the number of edges.

As the two heuristics give the same number of paths it should be interesting to be able to determine which one is the best heuristic to use. As explained before, the h_{basic} heuristic gives worse results than $h_{overcover}$, in terms of over-coverage. Fig. 4.8 shows the mean amount of times edges are over-covered with the h_{basic} heuristic. As it can be seen, on average, the edges are not too over-covered, the maximum is 1.6 paths through an edge, which is acceptable as this does not overload links. In addition, this means that edges will be tested twice, which does not seem to be a bad thing, if one test fails, the other may still succeed. If that case happens, we can hypothesize that the link has a problem but is still functional. However, other results show that the maximum times an edge is over-covered with the h_{basic} heuristic may be 25, which is too much. This may cause congestion if the tests are made simultaneously and may cause false results for the tests if one or more are dropped due to this congestion.

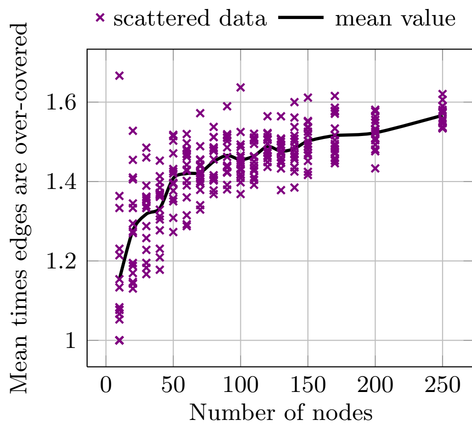


Figure 4.8: Waxman – Average number of paths per edges with the h_{basic} heuristic.

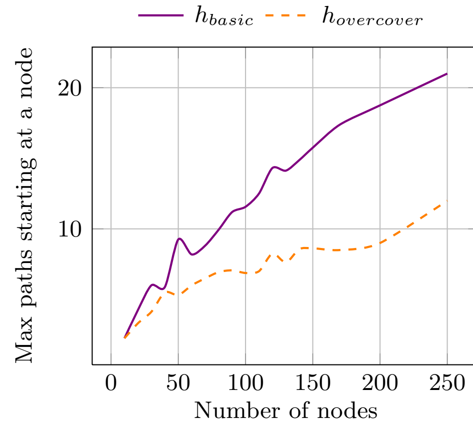


Figure 4.9: Waxman – Maximum number of different measurement paths starting at each node.

Fig. 4.9 shows the differences between the two heuristics in terms of overloading. We can see that $h_{overcover}$ gives better results, it gives fewer tests attributions than the other. This means that routers will less be overloaded by tests request, at most a router will have to do 10 tests at a time, 10 ping tests (see Sec. 4.3) seems acceptable

in a 250 nodes network.

Computation time

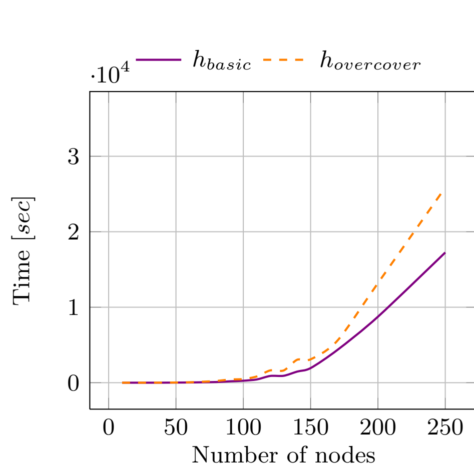


Figure 4.10: Waxman – Average computation time w.r.t. the number of nodes.

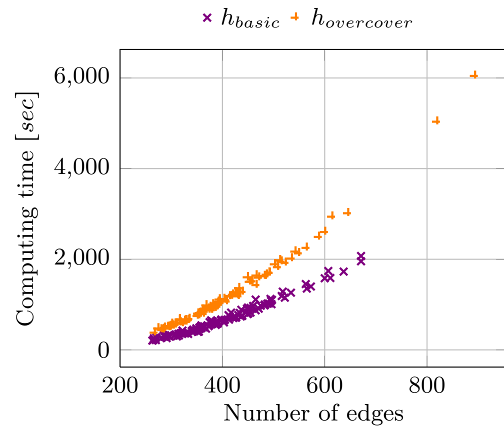


Figure 4.11: Waxman – Average computation time w.r.t. the number of edges (n = 120).

Another point to consider is the computation complexity. Fig. 4.10 and Fig. 4.11 show the evolution of the computation time in function of the number of nodes in the network, and the number of edges. We can see that more time to compute the coverage is required when there are more edges in the network. When there are more nodes there are much more possible routes in the network, for a n nodes topology,

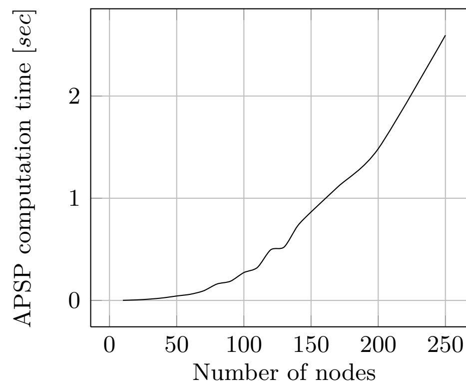


Figure 4.12: Waxman – Average *All-Pairs Shortest Paths* computation time w.r.t. the number of nodes.

there are $\frac{n(n-1)}{2}$ possible routes. So the algorithm needs to browse through all theses possible routes to find the best successor. There are only few differences between the two heuristics. It is interesting to notice that for $n = 250$ the computation time is almost, in the two cases, equals to 2000 seconds. Half an hour is too long to compute a coverage, we do not have this time before testing, as the network may have changed [WJL03].

A last element to confirm is the All-Pairs Shortest Paths algorithm time complexity, see Fig. 4.12, we can see that its computation time is negligible compared to the coverage algorithm.

4.5 Decentralized Algorithm

Early simulation results (see Sec. 4.4.4.2) of algorithm 4.4.1, showed a tendency to be time greedy when the number of nodes is larger than 150. We thus need to modify it, in order to make it usable in real situations.

4.5.1 Graph partitioning

A solution to the time greediness of algorithm 4.4.1 is to partition the graph into clusters. The graph partitioning problem [KL70] consists in dividing the graph into pieces, such that all the pieces have about the same size and there are few connections between them. Graph partitioning is known to be $\mathcal{NP}$-complete, but can be solved in polynomial time with fast heuristics [GJ79], that work well in practice. Fig. 4.13 and Fig. 4.14 show an example of graph partitioning into several clusters in the case of a regular Manhattan topology and a completely random topology. The dashed lines define the separation between clusters. For the Manhattan case, there are 25 nodes, thereby a cluster zone inherits from one more node than the others. On the random graph, the interesting thing to see is that the number of cuts, i.e., the number of edges from one cluster to another, is minimal. The maximal cuts is 5, while still keeping a good repartition of nodes into the clusters.

4.5.2 Parallel computation & Leader election

Upon the creation of partitions, we want to apply a load balanced version of algorithm 4.4.1 to perform a parallel computation on a smaller edge set (the edges of the cluster).

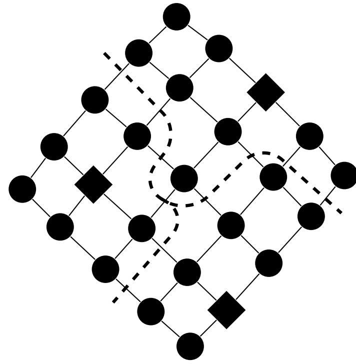


Figure 4.13: Partitionning of a 25 nodes manhattan grid into 3 parts.

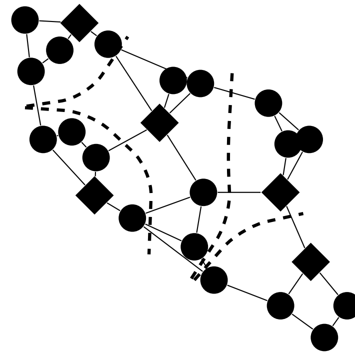


Figure 4.14: Partitionning of a 24 nodes random topology into 5 parts.

We need first to compute this cluster edge set, because graph partitioning does not care about edge repartition into clusters, it only cares about nodes repartition. There are a couple of ways to compute this edge set, based on a graph partitioning:

1. Randomly associate border edges to one of the two connected cluster.

2. By computing the number of edges in one cluster and associate a border edge to the cluster with the smallest number of edges.

We can define a subgraph as a graph whose vertex set is defined by the partition algorithm, i.e., its cluster nodes, and whose edge set is defined as above.

In order to compute a cluster coverage, we need, for each cluster, to elect a *leader*. A leader is the node that will compute the cluster coverage. In Fig. 4.13 and Fig. 4.14, they are respectively 3 and 5 leaders (diamond shape in figures), each running algorithm 4.4.1 on its associated subgraph. The leader can be elected based on several criteria:

1. Randomly.

2. Its current load, e.g., we do not want to overload a gateway or a crucial node.

3. Its computational skills, if the algorithm is time consuming, we prefer to execute it on a node with a fast CPU.

algorithm 4.5.1 represents the decentralized algorithm of the topology coverage problem. The algorithm depends on a single node that controls the computation. The procedures are:

Algorithm 4.5.1 The decentralized topology coverage algorithm.

Require: graph, a graph representation of a network topology
Require: routes, a set of possible routes in the graph, subject to $\forall r \in routes$: $2 \leq \text{Length}(r) \leq \infty$
Ensure: paths, a set of paths, subject to $\forall p \in paths : p \in routes$, representing the coverage of the graph

1: $paths \leftarrow \{\}$
2: $clusters \leftarrow \text{PartitionAlgorithm}(graph)$
3: $leaders \leftarrow \text{LeaderElection}(clusters)$
4: **for all** $leader \in leaders$ **do**
5: $\quad p \leftarrow \text{DistributeWork}(leader, clusters[leader], routes)$
6: $\quad paths \leftarrow paths \cup \{p\}$
7: **end for**
8: **return** $paths$

PartitionAlgorithm executes a partition algorithm on the graph. For simplicity, we want it to return a set of subgraphs.

LeaderElection elects each cluster leader.

DistributeWork parallelizes the algorithm computation. It can be implemented in any way, e.g., the *Parallel Virtual Machine* (PVM). PVM is designed to allow a network of computers/devices to be used as a single distributed parallel processor. It takes three parameters: the leader itself, the possible routes in the network and the subgraph ($clusters[clusters]$) linked to it.

To improve the computation time, a leader, when it receives the information, can try to limit the set of possible routes by only selecting those passing trough its cluster. And finally, one executes algorithm 4.4.1, with input: the restricted set of routes and the cluster graph.

4.5.3 Algorithm complexity

The complexity of this algorithm is difficult to define, as it depends on the parallel computation of algorithm 4.4.1 (seen Sec. 4.4).

As said before, the graph partitioning can be done in polynomial time, Kernighan and Lin showed that it can be done in $\mathcal{O}(n^2)$ [KL70], where n is the number of node in the graph. The leader election, meanwhile, can also be done in a polynomial time, depending on the strategy chosen.

4.5.4 Possible heuristics

The two heuristic, detailed in Sec. 4.4.3, can be used with algorithm 4.4.1, nevertheless, with the decentralized algorithm, we may need to limit the length of the paths. Because, we cannot limit the size of the paths to the diameter of the subgraph, as paths lying inside the cluster may not cover each edges of it. We cannot remove path, because, we do not know, in advance, if the path will be used to cover the graph. Furthermore, each leader does not have information on the other cluster coverage and as we do not want to over-cover edges, we need to limit the length of the paths.

For the decentralized topology coverage algorithm, we can use $h_{basic-limit}$ and $h_{overcover-limit}$ which are the same version as the heuristic depicted before.

The $h_{overcover-limit}$ heuristics can be construct as follows (similarly for the h_{basic} heuristic):

$$h_{overcover-limit}(state) = h_{overcover}(state) + mean_paths_length(state)$$

In this case, we add the mean path length to the cost of a solution. So the algorithm solution will contains small paths.

Again, as said in Sec. 4.4.3, these heuristics are not admissible. However, we do not need optimal solutions, we need to find, in an acceptable amount of time, a satisfying solution. In this case the solution is to find a solution that do not over-cover too much edges.

4.5.5 Evaluation

The following section describes the simulations and the results obtained by implementing algorithm 4.5.1.

Sec. 4.5.5.1 discusses the different tools used to generate topologies and the underlining topology models. And Sec. 4.5.5.2 shows the results obtained with the algorithm.

The same simulation environment was used as in Sec. 4.4.4. To implement the graph partition algorithm we use METIS [Kar08] graph partitioning software.

4.5.5.1 Topology models

In Sec. 4.4.4.1, we simulated on only one topology model: the Waxman model. To obtain accurate results, we chose to simulate over a large number of possible

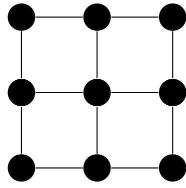


Figure 4.15: Manhattan topology.

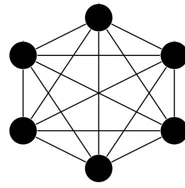


Figure 4.16: Fully connected topology.

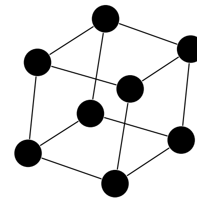


Figure 4.17: Hypercube topology ($n = 3$).

topology models. As if the algorithm is valid for all models, we can assume that the algorithm is valid for a real-world topology.

Topology models can be characterized into several families: regular networks, random, structural and degree-based generators.

Regular networks

A Manhattan topology (Fig. 4.15), is a graph in which the vertices are placed on a grid and the neighboring vertices are connected by an edge.

A fully connected topology (Fig. 4.16), also called a full mesh topology, requires that all nodes are connected to all other nodes. For a n nodes topology, there are $\frac{n(n-1)}{2}$ links.

An hypercube (Fig. 4.17) is n-dimension analogue of a square ($n = 2$) and a cube ($n = 3$). An hypercube is sometimes called a n-cube. For a n-dimension hypercube, there are 2^n vertices and $2^{n-1}n$ edges.

There are other possible structures, such as the star topology, the tree topology, the ring topology, etc. Regular topologies are often used in analytic studies of algorithmic performance because their structure makes them tractable.

Random graph generators

We already talked about random graph generators in Sec. 4.4.4.1. However, other models exists such as the *Doar-Leslie* model [DL93], which is a variation of the Waxman Model. It allows one to specify a desired average node degree.

Structural generators

Structural generators [TGJ+02] are a more interesting class of generators than pure random graph generators. They were designed to model the hierarchical structure of the Internet (as it is a collection of interconnected routing domains). They build the graph by connecting smaller components together.

The top-down hierarchical topology [bri07] is one approach to generate hierarchical topologies. It generates first an AS-level topology, according to different models, such as a random graph generators. Next, it generates a router-level topology inside each AS. And finally, it uses an edge connection mechanism to interconnect router-level topologies, dictated by the connectivity of the AS-level topology.

Degree-based generators

Degree based-generators [TGJ+02] differs from structural generators, as their idea is that network can be characterized by power laws. Degree-based generators are concerned by local properties, i.e., properties of individual nodes within a network. In particular, the local property of most importance is the node degree.

The *Barabási-Albert* (BA) [AB00] topology generator was the first degree-based generator. The idea behind the BA model is that it takes into account of the preferential attachment aspect. It preferentially attaches new nodes to existing well-connected nodes, leading to the incremental growth of nodes and the links between them.

The *Generalized Linear Preference* (GLP) [BT02] model focuses on matching characteristic path lengths and clustering coefficients. A probabilistic method is used to recursively add nodes and links while preserving the selected power law properties.

Generator tools

The are several topology generators available to the research community. Some of them, only using few of the models depicted before. Regular topologies come directly from the NetworkX [Hag08] python package. For generating the other topology models, we need to use these tools:

1. The *Boston University Representative Internet Topology gEnerator* (BRITE) [bri07] for the Waxman model, the BA model, the top-down hierarchical topology model and the GLP model.

2. The *iGEn* generator [Quo09] for the Waxman model.

3. The *Georgia Tech Internetwork Topology Models* (GT-ITM) [gti00] for the Waxman model, the Doar-Leslie model and the Random model.

4.5.5.2 Results

As in Sec. 4.4.4.2, we generate 15 different graphs to execute the algorithm for each different parameter set.

To be consitant with previous evaluation of Algo 4.4.1, we only present in this section results (especially figures) for the random graph generators family. Results for the other families are presented in Appendix A.

Computation time

The first thing to evaluate is the time to compute the coverage. As it was a main drawback of the previous algorithm, we do not that this algorithm suffer from the same issue.

	$h_{overcover}$			h_{basic}			$h_{overcover-limit}$		
	q_1	q_2	q_3	q_1	q_2	q_3	q_1	q_2	q_3
Waxman BRITE	30.1	32.7	37.5	48.2	64.6	135.4	42.3	56.6	74.2
Waxman iGen	54.5	115.3	188.5	52.4	84.2	163.1	38.9	78.3	195.5
Manhattan	48.6	51.4	52.7	38.6	41.6	42.7	41.4	42.1	45.0
Hypercube	53.6	54.9	57.1	72.6	75.2	77.9	73.7	75.1	79.9
Hierarchical	67.1	71.0	74.8	66.0	70.5	80.6	62.0	66.2	71.1
GLP	42.1	56.9	79.9	169.8	227.1	345.7	80.9	139.7	244.9
Full mesh	147.4	155.5	166.0	131.4	137.4	150.7	99.3	102.6	114.4
BA	34.4	40.6	54.9	73.4	134.7	174.3	47.1	91.6	139.6

Table 4.1: Coverage time percentiles – 25th percentile (q_1), median (q_2) and 75th percentile (q_3) in seconds for all possible topologies with all possible heuristics with a cluster size of 20 nodes for a total number of nodes of 250.

Tab. 4.1 gives the average time to compute the coverage of the network. We consider the coverage of network being the time to compute all its clusters in parallel. Therefore, the average coverage computation time is the average cluster coverage computation time. It is easier to visualize the differences in Fig. 4.18 which displays box-plots for each possible topology models, when the $h_{overcover}$ heuristic is used. The lower bound of the box-plot is the 25th percentile, the red line is the median, the upper bound is the 75th percentile, and points out of the box are outlier (an observation that is numerically distant from the rest of the data).

Compared to the result in Fig. 4.10 where it takes on average 2000 seconds to compute, here it takes, on average for all topologies, less for the same number of nodes.

The parameters used for the different topology models (if any) are:

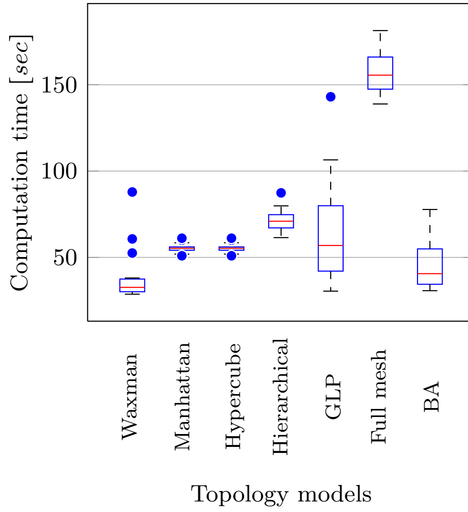


Figure 4.18: Coverage time box-plot for each topology models ($h_{overcover}$).

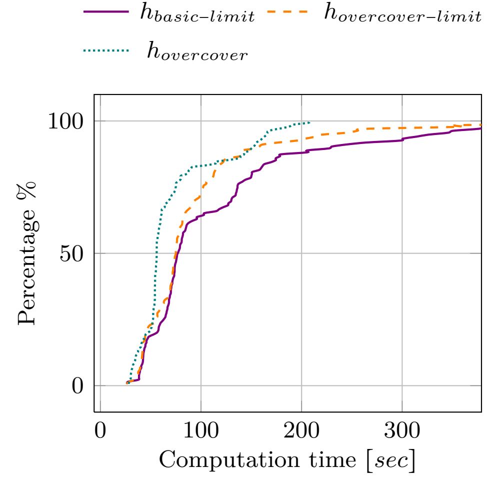


Figure 4.19: Computation time cumulative distribution (all models together).

1. Waxman BRITE and Waxman iGen: $\alpha = 0.15$, $\beta = 0.2$ and $m = 2$.

2. Hierarchical top-down: Waxman with $\alpha = 0.15$, $\beta = 0.2$ and $m = 2$ is used to generate AS-level topologies of size $m = 10$ nodes.

3. GLP: $p = 0.45$, $\beta = 0.64$ and $m = 1$.

There are not much differences between $h_{overcover}$ and h_{basic} heuristics. However, $h_{overcover-limit}$ gives sometimes higher time to compute (in the GLP topology). GLP topologies have much higher number of edges to cover per cluster. As the $h_{overcover-limit}$ tries to find small paths that do not over-cover, it will go deeper in the search tree to find the solution and then take much time to compute.

We can see in Tab. 4.1 that Waxman iGen gives worst results than Waxman BRITE, the reason is that the latter use a fixed number of edges while the first allows more flexibility and generate on average more edges. The result is thus normal: we saw in Sec. 4.4.4.2 that the number of edges influences strongly the computation time.

The computation time seems reasonable. It takes on average less than 100 seconds to cover the graph. Depending on the routing protocol configuration, e.g. the time between two hello messages, there may be no changes within this time frame.

The only issue is for the full mesh topology and the GLP topology where it takes more than 100 seconds on average to compute the whole coverage.

The full mesh case is caused by its number of edges, as we saw in Sec. 4.4.4.2, it mainly cause the algorithm to be time greedy. For a 250 nodes network a full mesh topology contains 31125 edges, while for a Waxman topology there are approximatively 1000 edges. However, the issue cannot be taken into account, as in reality it is nearly impossible to find a 250 nodes full mesh network.

For the GLP topology, the issue is due to the fact that $h_{overcover-limit}$ and $h_{basic-limit}$ are not able to easily find a limited amount of paths to cover the graph. The GLP topology is disparate, it is composed by a small center which is heavily connected, the border nodes are connected to one center node by a single link. As the possible paths are small, the algorithm with the two heuristics takes too much small paths causing it to go deeper in the search tree, an thus taking more time to compute.

Fig. 4.19 gives interesting results, it shows the cumulative distribution of the algorithm computation time for all topology models together. In the case of the algorithm running with the $h_{overcover}$ heuristic, almost 80 percent of the solutions are found within 100 seconds. However, $h_{basic-limit}$ and $h_{overcover-limit}$ gives the same result at different computation time, respectively, 150 and 112 seconds. $h_{overcover}$ show better behavior in terms of computation time, all the solutions are found within 200 seconds.

Coverage quality and algorithms comparison

The main evaluation of this algorithm is the quality of the coverage. It is hoped that this algorithm will have the same behavior as the centralized one.

It is interesting to look at the differences between the centralized and the decentralized algorithms. Fig. 4.20 and Fig. 4.21 show this difference, the same simulation parameter is used in both cases (Waxman iGen and $h_{overcover}$). In fact the differences are minimal. In Fig. 4.20, they both give approximatively the same result. However, the only notable difference, is the over-covering of edges. In the centralized algorithm, $h_{overcover}$ give a very good result, an edge being almost never over-covered. With the decentralized algorithm, it gives less good performance, in the worst case, one edge of a 250 nodes network is covered with 10 paths. This is still acceptable, because: ICMP packets are very small and all tests are not performed simultaneously. This behavior can be explained by the fact that there are no information about the covering shared by the clusters leaders. Therefore, they

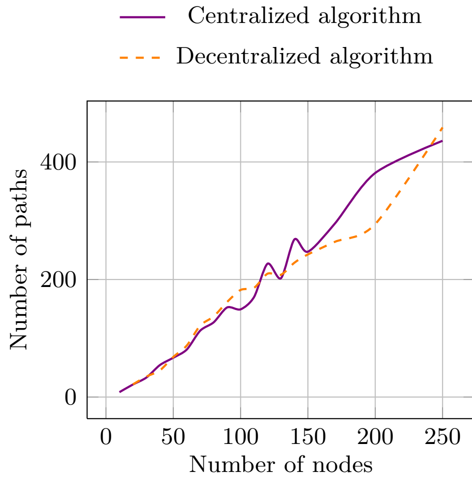


Figure 4.20: Waxman – Number of paths used to cover ($h_{overcover}$), comparison of the two algorithms.

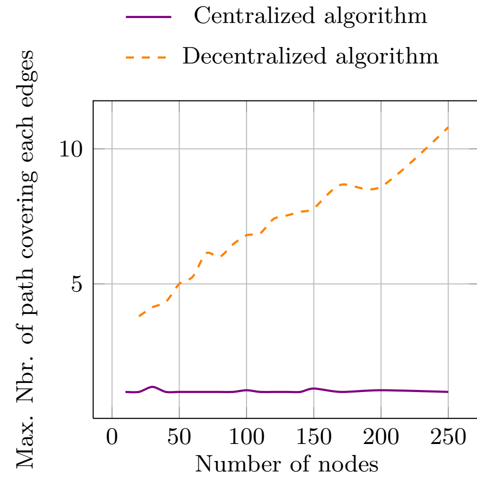


Figure 4.21: Waxman – Maximum number of path covering each edges ($h_{overcover}$), comparison of the two algorithms.

might chose a path where a edge is already covered by an other leader.

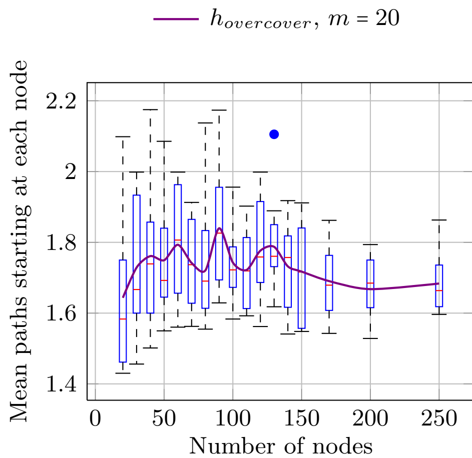


Figure 4.22: Waxman – Average number of different paths starting at each node.

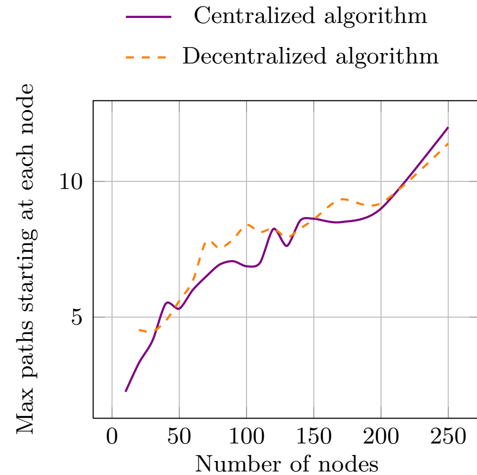


Figure 4.23: Waxman – Maximum number of different paths starting at each node, comparison of the two algorithms.

What are the main differences with the centralized algorithm? First the overloading of request of nodes. In the case of algorithm 4.5.1, Fig. 4.22 and Fig. 4.23 show us the influence of this algorithm in the nodes overloading. We can see that the average number of paths starting at each node is slightly always the same. There are on average 2 paths starting from each node, which is acceptable, as there are not

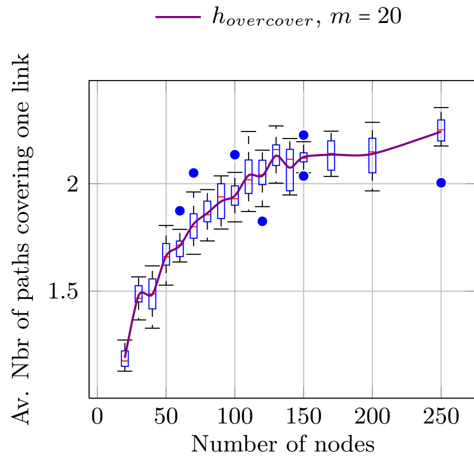


Figure 4.24: Waxman – Average number of times each edges is over-covered.

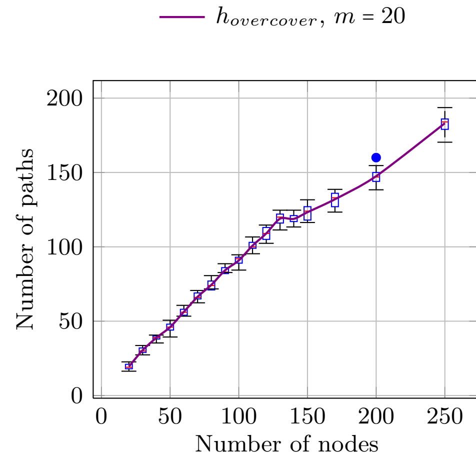


Figure 4.25: Waxman – Number of paths used to cover the topology graph.

much differences between one and two queries. On the other hand the maximum number of paths starting at a node is, for a 250 nodes topology, equal to 12. The result is neither better or worst compared to the centralized algorithm.

Fig. 4.25 gives the amount of paths we need to cover a Waxman topology graph with the $h_{overcover}$ heuristic. We can see that there is a constant coefficient of proportionality between the number of path and the number of nodes.

As said before, an important parameter to observe, is the over-coverage of edges. We do not want to over-cover edges, as it will result in a congestion of this links at the network level. Fig. 4.24 shows the average over-coverage of edges, that we can expect with the decentralized algorithm. We can see that links are slightly covered by only one path at most. This is a good result, it shows that this important parameter is respected.

The results are similar for the other topology families, see Appendix A, Sec. A.2.

Cluster size

Another parameter to evaluate is the influence of the cluster size. We only evaluate the cluster size on the Waxman model.

A priori, when the cluster size is small, the computation time should be very small, as the edges set is small. However, the coverage should be bad, as clusters do not have information about others, e.g., they can take paths that are already over-covered. The opposite should happen for large clusters, in this case it should behave as algorithm 4.4.1.

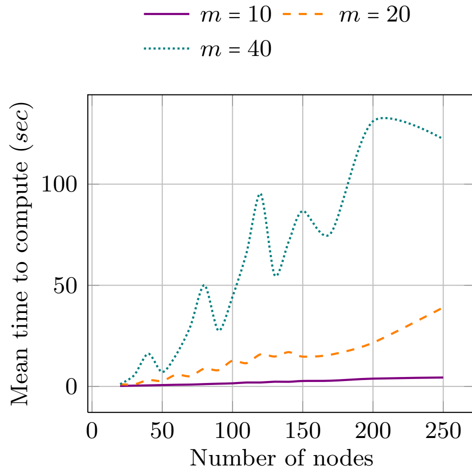


Figure 4.26: Waxman – Average computation time w.r.t. the number of nodes for cluster size $\in [10, 20, 40]$ nodes with $h_{overcover}$.

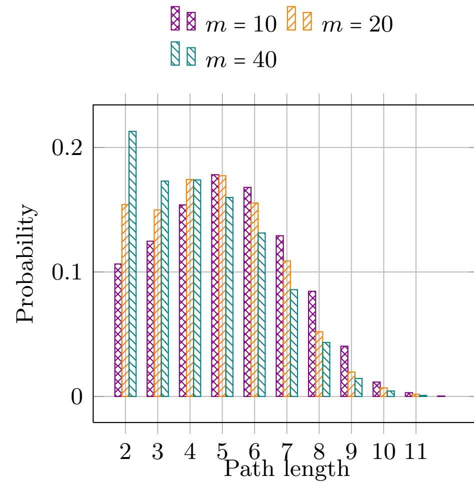


Figure 4.27: Waxman – Path length distribution for cluster sizes $\in [10, 20, 40]$ nodes with $h_{overcover}$.

Fig. 4.26 and Fig. 4.27 give an idea of the importance of the cluster size. In Fig. 4.26, we see that there is a difference of more than 100 seconds to compute a solution between a cluster of 10 nodes and cluster of 40 nodes. This is because in clusters of 10 nodes there are fewer possible routes than in the other case, approximatively 8224 routes against 33851 for a 250 nodes topology. And also, because there are fewer edges to cover. These two reasons lead to a computation time that is smaller for small clusters size.

Another thing to evaluate is the path length distribution. When the cluster size is small, the length of the paths used may be too small. Fig. 4.27 shows the path length distribution for different cluster sizes. We can see that there is an equal repartition of path lengths, i.e., that the probability of finding a path length of 2 is equivalent to the probability of finding a longer path. Conversely, in algorithm 4.4.1, there is a really bad repartition of path length when the $h_{overcover}$ is used, approximatively 80% of path of length of 2 are used. Which is not what we aspire.

To conclude with the evaluation of the cluster size, we can say that a cluster size of 20 nodes is a good compromise between time complexity and path length distribution.

Heuristics evaluation

We have seen for algorithm 4.4.1 in Sec. 4.4.4.2, that heuristic $h_{overcover}$ has a better

behavior than h_{basic}. Nevertheless for the decentralized algorithm, we defined two new heuristics. We need to evaluate them, in order to decide which is the best one.

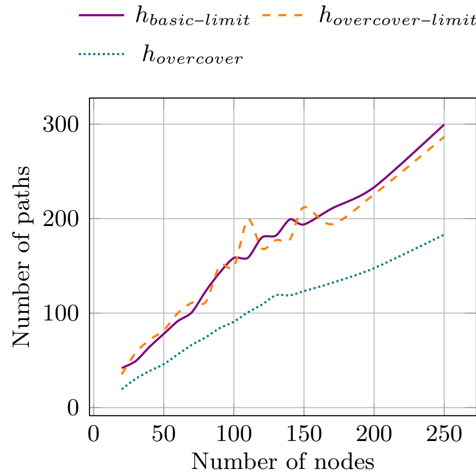


Figure 4.28: Waxman – Comparison of the number of paths when the three heuristics are used.

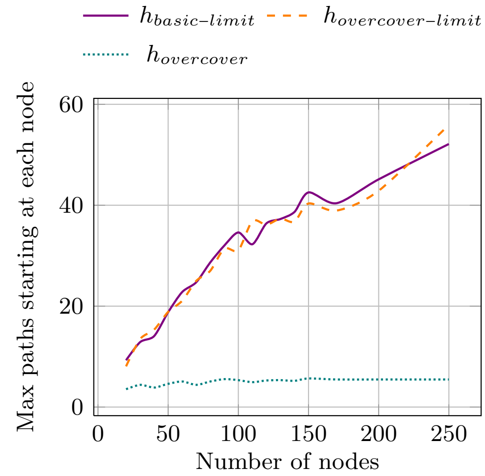


Figure 4.29: Waxman – Comparison of the Maximum number of paths starting at each node when the three heuristics are used.

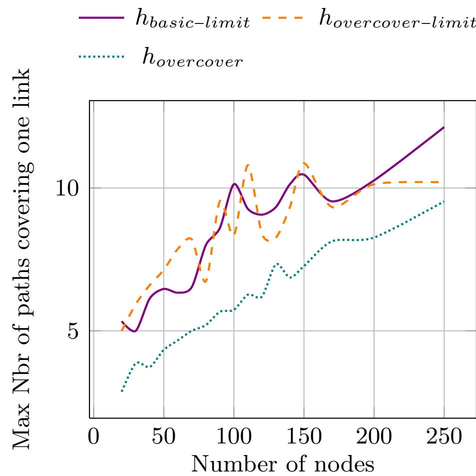


Figure 4.30: Waxman – Comparison of the Maximum number of paths covering each edge when the three heuristics are used.

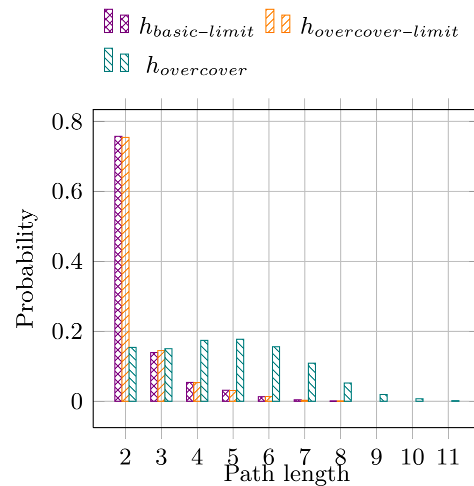


Figure 4.31: Waxman – Length distribution of path used to cover the graph.

Fig. 4.28 shows the average number of paths used to cover a Waxman topology. In this case the $h_{overcover}$ heuristic gives much better results than the two other heuristics. This is a special case because on Fig. A.5 in Appendix A, which shows

the same result but for a top-down topology, the differences between the heuristics are not so pronounced. In the first case, the number of edges is larger than in the other case, this cause that the two limited heuristics, $h_{basic-limit}$ and $h_{overcover-limit}$, choose smaller paths which result in a larger number of path in the results. However, when the number of edges is smaller, the differences between the heuristics are limited.

Fig. 4.29 and Fig. 4.30 show respectively the maximum number of paths starting at each node and the maximum over-coverage of each edge. We can see, and it is also valid for previous figures, that there are not much differences between $h_{overcover-limit}$ and $h_{basic-limit}$. It is interesting to notice that for the full mesh topology (see Fig. A.4 in Appendix A), there are no more similitudes between $h_{overcover-limit}$ and $h_{basic-limit}$ but in this case $h_{overcover-limit}$ and $h_{overcover}$ have the same behavior.

Regarding $h_{overcover}$, the results are different, in Fig. 4.29 and Fig. 4.30, it gives good results. In this latter case, for a Manhattan topology (see Fig. A.1 in Appendix A), the $h_{overcover}$ heuristic gives much worse result than the other heuristics. It seems that is continue to grow when the number of nodes is greater than 250. This result is not present in the degree-based topology family, see Fig. A.1 in Appendix A, in this case, the $h_{overcover}$ heuristic gives always best results.

Fig. 4.31 gives the probability to use a fixed length of path to cover the graph. The result is obvious, as we designed $h_{overcover-limit}$ and $h_{basic-limit}$ to limit the length of paths, it is normal to find that the distribution for these two heuristics is concentrated into small path lengths. This is not expected as it results in a larger number of paths used to cover the graph (see Fig. A.5).

Based on these results, it is difficult to conclude on the fact that one heuristic is better than the other. As it can vary from topologies, the best way is to try $h_{overcover-limit}$ and $h_{overcover}$ and use the one giving the best results. Nevertheless, the $h_{overcover}$ heuristic seems to be the best compromise, as its only problem is the over-coverage of edges, we can hypothesize that for small size networks that it is not a problem anymore.

4.6 Conclusion

The topology coverage problem is hard to compute, to simplify it we introduce simplifications such as a limitation of possible path used to cover the graph and

the use of an heuristic based algorithm. This algorithm is, however, not good enough, due to time consumption. Based on this flaw, we design an other algorithm, it decentralize and parallelize the centralized algorithm. The topology coverage becomes a set of cluster coverage. This improvement also increases the quality of the algorithm, taking now a respectable amount of time to compute, while still keeping desired behaviors.

Chapter 5

A practical application

Now that the monitoring objectives are defined (see Chap. 3, Sec. 3.2) and fully described, we can start the implementation of a practical application for verifying the theory validity. Due to the limited size of our testbed, we do not test in details all the functionalities, we rather focus on the behavior of the program in presence of a known case.

5.1 Testbed

A network testbed is a platform for experimentation like a sandbox in software engineering but, in our case, for developing network projects. In this section we discuss the hardware at our disposal and the network environment, i.e., the position of each nodes.

5.1.1 Material

The testbed is composed of *Accton MR3201A* nodes, see Fig. 5.1. The Accton MR3201A is officially supported by the *Open Mesh* project [ope08]. It is cheap, less than $50, and elegant, it is a mini-router (small dimension).

From a technical point of view, it is based on the *Atheros AR2315* chipset. It is composed of a 32MB DRAM, 8MB Flash, one 2*dbi* antenna and one Ethernet port. Thanks to the Atheros chipset, the Accton MR3201A has the possibility to create several virtual wireless interfaces, to transmit and receive data. The wireless interfaces are IEEE 802.11b/g compliant, with an output power of 60MW.

Figure 5.1: The Accton MR3201A router.

5.1.2 Environment

Fig. 5.2 depict the locations of the nodes into a real environment: the Reaumur building, 2, place Sainte Barbe, 1348 Louvain-la-Neuve (Belgium). The environment is a three-floor building, the nodes positions do not aim at efficiently covering the building.

The environment contains nine nodes, seven of them are simple routers. `5.137.144.5` is a Gateway and is directly connected to the topology server, it announces the route: `10.168.2.0/24` through OLSR HNAs. The sniffer is connected via Ethernet to `5.137.147.105`. It is configured to send information to `10.168.2.4` which is the address of the topology server.

5.2 Architecture & roles

The architecture has already been defined in the monitoring objective in Chap. 3, Sec. 3.2. The components are: the sniffer, the topology server, the leader and the pinger. We describe in details the desired behavior of each of these components.

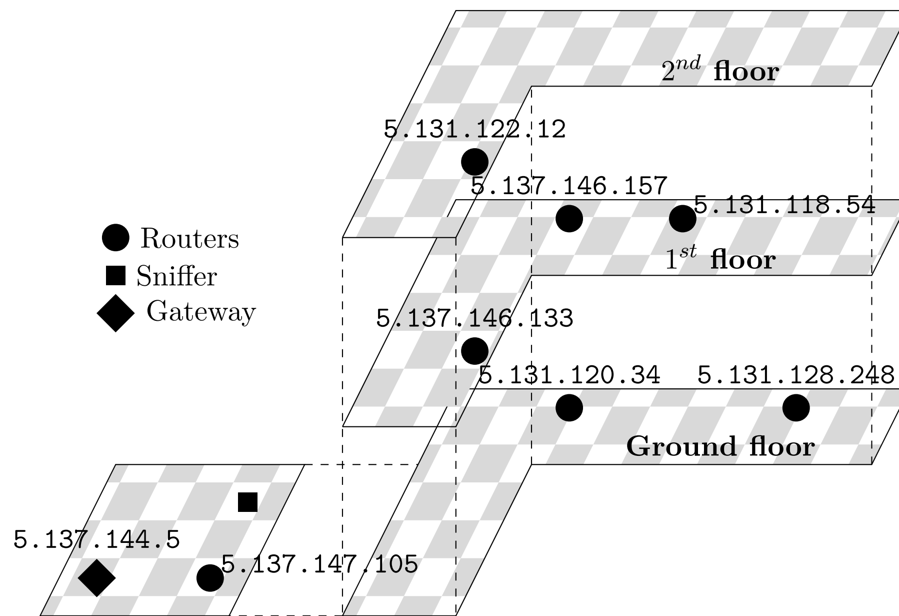


Figure 5.2: Testbed – Routers and Sniffer positioning in the Reaumur building.

5.2.1 The sniffer

The sniffer component listens to the network routing protocol (in our case OLSR) and forwards changes to the topology server. To be efficient we prefer to aggregate changes into a single message rather than sending a message every time a change occur.

Shaikh et al. already defined several message types for possible changes in the network [SG04]. The three important ones are:

Messages about a router

They define the state of routers, the messages are: RTR UP and RTR DOWN. There is one parameter to the messages, the router id, i.e., its main IP address.

Messages about an adjacency

They define the state of links in the network the messages are:

1. ADJACENCY UP and ADJACENCY DOWN, similarly as messages for a router. In this case, the parameters are the routers id that are connected on both end of the link.
2. ADJACENCY COST CHANGE, if the link cost changes. This message take one more parameter than the previous one: the current cost of the link.

3. ALL ADJACENCIES DOWN, defines that all links connected to a node are down. The only parameter is the down router id.

Messages about a prefix

In OLSR nodes can define HNAs, which can be announced and withdrawn, the messages are: ROUTE ANNOUNCED and ROUTE WITHDRAWN. Their parameters are: the prefix, the mask of the network and the router id where the route is attached.

The sniffer creates such messages based on information obtained from the routing protocol listening.

5.2.2 The topology server

As it name suggests, it maintains information about the current and past topology. It is directly connected to the sniffer and, thus, processes the messages described in the previous section. In addition to maintaining the topology, the network maintainer should be able to visualize the topology, therefore, it needs to contains a visualization module. Objective 3 described a tool to validate the network view of the routing protocol (see Sec. 3.2). In Chap. 4, we explained the validation algorithm (see algorithm 4.5.1 in Sec. 4.5). This algorithm must be implemented within the topology server, and thus must implement the following procedures:

PartitionAlgorithm implements the cluster decomposition of the graph. It returns a set of clusters. A cluster is defined by its set of nodes and edges.

LeaderElection elects a leader for each cluster. The possible implementations of this procedure are already been discussed in Sec. 4.5.

DistributeWork contacts the leader and gives him the work that needs to be done, i.e., covering its cluster with paths.

5.2.3 The leader

As explained in previous section, a leader must cover its cluster with paths. It must also implement algorithm 4.4.1. The leader component should be implemented on every nodes participating in the routing protocol, as, depending on the election implementation, every node may be selected as leader. When the coverage path set is available (after the execution of the algorithm), the leader needs to test all

the paths. Therefore, it contacts all nodes that start a path in the set, and gives it one or several destinations to ping, depending on the number of paths starting at this node. Upon the reception of all ping results, the leader sends compressed information back to the topology server.

5.2.4 The pinger

The pinger component must be implemented on all routers of the network. The pinger behavior is very simple and can be decomposed in three events:

1. It receives the target(s) to ping.
2. It executes a number of pings (depending on the query) to targets.
3. It sends back information about whether the target(s) was(were) successfully pinged or not. In addition to gives this binary result, it also sends information about the *Round-Trip delay Time* (RTT) of all successful pings.

5.3 Configuration

5.3.1 The routers

5.3.1.1 Base configuration

Originally, the routers were configured with the default firmware [ope08], which is based on the *ROBIN* distribution [Ans09] of *OpenWrt* [ope09], a Linux distribution for embedded devices.

ROBIN means ROuting Batman INside mesh network, and thus was based on the BATMAN routing protocol. Recently the maintainers of ROBIN added the support of OLSR.

ROBIN contains components that are not needed for a testbed, e.g., a captive portal. However it contains also desirable characteristics, mainly the auto-configuration (plug & play) of nodes.

For these two reasons, we modified the ROBIN distribution to create a lighter version. The routers of the testbed use a self-made distribution of OpenWrt, matching the desired characteristics.

5.3.1.2 Hardware & software Limitations

The algorithms of Chap. 4, were originally implemented in Python (see Sec. 4.4.4). However the Python library is too large for the Accton router. Several Python replacement possibilities are available, nevertheless, we chose to port the algorithms in Ruby [rub09]. Ruby is composed of several packages, which allow us to only select the required ones, and thus save memory, while the entire Python library is contained on a single package. In addition to a reduced memory footprint, the ruby language is similar to Python. Even if Ruby and Python have a very different syntax, they is similar in varying respects, allowing to easily port the implemented algorithm to Ruby.

The main components present in our self-made distribution are:

- `OpenWrt SVN r14135 (last changes: 2009-01-21)`
- `OLSR 0.5.6-r2`
- `Ruby 1.8.6-p36`

5.3.1.3 Interfaces configuration

The routers are configured to have three interfaces, two of them are virtual. The interfaces are configured as follows:

`ath0` The routing protocol wireless interface. The routers are pre-configured with a Class A IPv4 address: `5.0.0.0/8`. This is a router constructor choice. The prefix `5/8` is currently defined as unallocated by the IANA. The router final IPv4 address is constructed from its MAC address. This is simply done by this bash script:

```
hex2dec() {
  MAC=$(ifconfig eth0 | grep HWaddr | awk '{print $5}')
  let x=0x$(echo ${MAC} | cut $1)
  echo $x
}
echo 5.$(hex2dec -c10-11).$(hex2dec -c13-14).$(hex2dec -c16-17)
```

Code 1: ath0 IP address computation.

The router runs OLSR on this interface. The `ath0`'s IP address is thus the router's identifier inside the OLSR routing protocol.

`ath1` The wireless *access point* (AP) interface, used for wireless client connections. A prefix `/25` is allocated for all wireless clients. The IP address of this interface is computed as for `ath0`:

```
echo 10.$(hex2dec -c13-14).$(hex2dec -c16-17).1
```

Code 2: ath1 IP address computation.

`eth0` The ethernet interface, used for one wired client connection or as a connection to a network. In the latter case, the router is considered as a Gateway. If the interface allows a wired client, a `/25` prefix is also allocated for this interface and the IP address is the same as `ath1` but ending with `.128`.

5.3.1.4 Behavior configuration

There are several possible configurations for each router, in which the interfaces may or not be used. The configurations are:

Wireless AP

The router acts as an access point, and allocates addresses to clients via DHCP. In this case the `ath1 /25` prefix must be announced by OLSR. All the routers enable the Wireless AP configuration which broadcast the same wireless *ESSID* with the same security. This Allows handover from one Wireless AP to another.

Gateway

The router retrieves or manually configures the connected network prefix and mask, and announces it within OLSR.

Wired DHCP Server

The router acts as a wired DHCP server for the client. In this case the `eth0 /25` prefix must be announced within OLSR.

These configurations may be combined, except the Gateway and the Wired DHCP Server combination.

5.3.1.5 OLSR configuration

Code 3 contains the main part of the OLSR configuration of each nodes. This configuration comes from the ROBIN distribution.

```
1 Interface "ath0"
2 {
3         HelloInterval 2.0
4         HelloValidityTime 108.0
5         TcInterval 4.0
6         TcValidityTime 324.0
7         HnaInterval 18.0
8         HnaValidityTime 108.0
9 }
```

Code 3: Part of the OLSR configuration.

The configuration options are defined by interfaces, here on the one running OLSR: `ath0`. As their names suggest it, the `*Interval` options define the interval between two generations and transmissions of messages. The `*ValidityTime` options set the validity time to be announced in messages generated by the host on its interface.

5.3.2 The sniffers

The sniffers also contains a self-made OpenWrt distribution. There are two ways for a sniffer to connect the topology server:

1. By connecting the testbed Wireless APs.
2. By connecting the wired interface of a router, providing that it is configured as a Wired DHCP Server.

5.4 Implementation

This implementation is not a ready-to-sell program. It is a prototype that allows us to evaluate the behavior of our algorithm in a real environment.

As the testbed contains only eight routers, we do not implement the partition algorithm.

In this implementation we simplify some parts, that are explained in Sec. 5.2. For example we do not implement the changes messages for the sniffer. As the network is really small, sending the whole graph at each time does not consume to much bandwidth. For that we use Ruby XML/RPC implementation, that allows one to send objects through the network.

In addition to Ruby, the binaries and libraries used for the implementation are:

ruby-gnome2 0.16.0-10 is the Ruby binding for `Gtk`, it is used to display the topology and informations on the screen.

ruby-pcap 1.8.6 / libpcap 0.9.8 are the libraries used to capture and analyze network traffic.

ruby-xmlrpc 1.8.6 allows to create and connect an `XML/RPC` server. It supports a very nice way to create and call a remote procedure and is easy to use:

```
1 s = XMLRPC::Server.new 8080
2 s.add_handler("server.add") do |a,b|
3   a + b
4 end
```

Code 4: ruby-xmlrpc example: server side.

At line 1 an `XML/RPC` server is created on port `8080`. At line 2, a procedure is added to the server with two parameters `a` and `b`, the result of the procedure is `a+b`.

The client side is the following:

```
1 server = XMLRPC::Client.new "server_ip", "/", 8080
2 res = server.call "server.add", 4, 5
3 puts "4 + 5 = #{res}"
```

Code 5: ruby-xmlrpc example: client side.

At line 1, the server with `server_ip` IP address is connected on port `8080`. At line 2, the remote procedure `server.add` is called with the parameters `4` and `5`.

tcpdump 3.9.0 [MG09] is a packet sniffer, it allows one to intercept several types of packet being transmitted or received over the attached network.

The source-code of all the different part can be found in Appendix B.

5.4.1 Application Programming Interface (API)

Each part of the whole program defines one procedure, available via `XML/RPC`. These procedures are:

monitord.commit(graph)

Sends `graph` to the topology server.

leader.eval(cluster,routes)

Evaluate the status of the `cluster`, returns a dictionary of results.

pingd.ping(list,num)

Execute a ping, `num` times, for each destination in `list`. Returns a dictionary of results, the delays for successful pings or a value ≤ `0` for unsuccessful ones.

5.4.2 Details & functionalities

In this section we review the implementation of all the part of the whole program.

From each component detailed in Sec. 5.2 result a binary as part of the whole implementation. The binaries are: `sniffd`, `monitor`, `leaderd` and `pingd`.

The source code of this practical implementation is available in Appendix B.

The only shared component between all binaries is the representation of the topology graph: the `Graph` class. This class is serializable (it contains only one class variable: the graph adjacency array), so that a `Graph` object can be sent over `XML/RPC`.

5.4.2.1 The sniffd

There were some unresolved issues with `ruby-pcap`: it was not able to sniff OLSR packets from the network. To resolve this problem we use in parallel `tcpdump` and `ruby-pcap`, as in:

```
tcpdump -s0 -i mon0 -w - | ruby -C/usr/lib/sniffd/ sniffd.rb
```

Code 6: tcpdump with sniffd.rb.

In this script, we launch `tcpdump` to listen to the `mon0` interface and write all the packets to `STDOUT`. We launch also a ruby script called `sniffd.rb` that behaves as in Code 7.

First it opens `STDIN` and read all packets `tcpdump` writes (line 4). To collect only OLSR traffic, `ruby-pcap` contains a filtering. This filtering is configured at line 5, it limits the library to only handle UDP packets whose source and destination ports are 698, which is the default configuration of OLSR.

```
1 require 'olsr'
2 require 'pcap'
3
4 capture = Pcap::Capture.open_offline('-') # open STDIN
5 capture.setfilter 'udp port 698' # capture OLSR packets
6 olsr = Olsr.new
7 capture.each do |pkt|
8   olsr.each pkt.udp_data  do |info|
9     ...
10  end
11 end
```

Code 7: sniffd.rb main behavior.

From this, we designed a Ruby plugin: `ruby-olsr`. This plugin is implemented in `C`, and handle the OLSR traffic. Its goal is to transform OLSR messages into a simple structure to be easily handled in the Ruby environment. The structure used is an `Hash` (or dictionary), at line 8 the dictionary is the `info` variable, formated as follows:

```
info = {'ADDR' => {'networks' => { 'PREFIX' => 'MASK', ...}, ↩
       'neighbors' => {'ADDR' => COST, ...}}, ...}
```

Code 8: Formating of the ruby-olsr output.

Where the uppercase keywords vary from one OLSR message to another.

A `Graph` object, that represents the network topology, is maintained. Once, a packet has been received, the graph is updated, and if it changed, the changes are forwarded to the topology server, see Code 9.

```
1 Thread.new do
2   begin
3     s = XMLRPC::Client.new server, '/', 34587
4     s.call 'sniffd.commit', graph
5   rescue Exception => e
6     $log.error "Error while commiting to the topology server: #{e}"
7   end
8 end
```

Code 9: sniffd.rb commit of a graph to the topology server.

The operation is very simple, thanks to Ruby. First, the connection is established with the topology server (line 3) by creating a new `XMLRPC` object. To commit the graph, the `sniffd.commit` procedure is called remotely (line 4), it takes only one parameters: the `Graph` object.

5.4.2.2 The monitor

The `monitor` is composed of two components:

`display.rb` contains the `Display` class, and as its name suggests, takes care of the display of the topology and messages. To properly display the topology, we use the `neato` binary from *Graphviz* [EGH]. `neato` generates a spring-layout version of a graph, which is very interesting if we want to display it. The interface, in addition to displaying the topology, allows one to display problems on paths, e.g., useful when the coverage result fails, etc. For a better understanding of changes, `monitor` displays also newly up links/nodes in green, for a fixed period of time. For down links it displays them in red.

`monitor.rb` contains two threads: one for the reception of graphs from `sniffd` and one for the connections to leaders. Upon the reception of a graph, it places it in a `PQueue` object. The `PQueue` object is interesting as it suspends the current thread when the `pop` procedure is called. So, in the second thread, the first operation is to call the `pop` procedure to retrieve the previous graph. After this, two operations must be performed:

1. The computation of the possible routes in the graph (limiting paths to paths of length greater or equal to 2).
2. The leader election.

Here, as we have a limited testbed, we run the leader script on the topology server, so that the leader is always `localhost`.

After these two computations, information is sent to the leader via the remote `leader.eval` (via `XML/RPC`) procedure that takes two parameters: the cluster (here the whole topology graph) and the possible routes.

When new informations are available, e.g., a graph, results from the leader, etc, are displayed by the `display.rb` module.

5.4.2.3 The leaderd

The `leaderd` behavior is the following:

1. When `monitor` calls its procedure `leader.eval`, it calls the implementation of the coverage algorithm (see Algo 4.4.1, in Chap. 4, Sec. 4.4), resulting in a set of paths.

2. For each path in the paths set, it contacts the `pingd` daemon of each node starting a path, and gives it the destination to ping. For more accurate result, it performs five pings for each destination, and compute the average RTT of successful pings.

3. The summary is sent back to the `monitor` client.

5.4.2.4 The pingd

The `pingd` daemon is quite simple, it is implemented in `C` and defines a remote procedure via `XML/RPC`: `pingd.ping`. The important point to notice is that the ping timeout is set at 1 second and that the client must wait for the ping to be executed to receive data.

The `pingd` client is also simple, to test the path from `src` to `dst`, a client must do in Ruby:

```
1 server = XMLRPC::Client.new "5.130.104.5", "/RPC2", 8080
2 result = server.call("pingd.ping", ['www.google.com', ↩
3   '130.104.1.1'], 3)
```

Code 10: pingd Ruby client example.

The first operation to do is to connect the `XML/RPC pingd` server (line 1). At line 2, it calls the remote procedure `pingd.ping` to execute `3` pings to two destinations (`www.google.com` and `130.104.1.1`).

The procedure returns a dictionary of delays for each destination, such as in Code 11. The `0` result means that the destination was not reached within 1 second.

```
{"www.google.com" => [0, 3456, 1432], "130.104.1.1" => [3986, ↩
    4512, 6789]}
```

Code 11: Example results from the pingd.ping procedure.

5.5 Review & tests

Now that we have designed and implemented the monitoring application on our testbed, we can evaluate the behavior of the network by doing some tests. These tests do not aim to fully evaluate the accurate state of the network. However, we can observe the evolution of the topology according to several possible cases:

Node down

Nodes may need to be rebooted, e.g., to update software, etc. The monitoring tool needs to quickly represent the changes in the topology.

Packet loss

Nodes may suffer from congestion. We simulate a 70% packet loss on a single node.

Delay

It is possible, due to the mobility of nodes and the possibility to have interferences in the wireless connectivity, that the delay may increase. We simulate a 100ms and 1000ms delay on a single node.

In the following section, we explain the results obtained when these three cases appends.

The *traffic control* (`tc`) [Hub03] tool is used for the congestion and delay tests. It allows one to manipulate traffic control settings in the Linux kernel, by applying traffic shaping, scheduling, policing and dropping.

5.5.1 Evaluation

Before performing tests, we must make sure that the monitoring program successfully represents the topology. The physical topology was detailed in Sec. 5.1.2, the monitoring program gives us the state of the network in Fig. 5.3. We can see that all nodes are represented and they are fully connected (in fact almost a full mesh topology). This is due to the fact that we placed nodes randomly and do not care about a good space utilization and coverage.

It is also interesting to notice that the two nodes that are the farthest of each other are connected, `5.131.118.54` and `5.137.144.5`, but not `5.137.147.105` and `5.131.118.54`, this is surely due to perturbation, as in the building there are others functional AP and wireless devices.

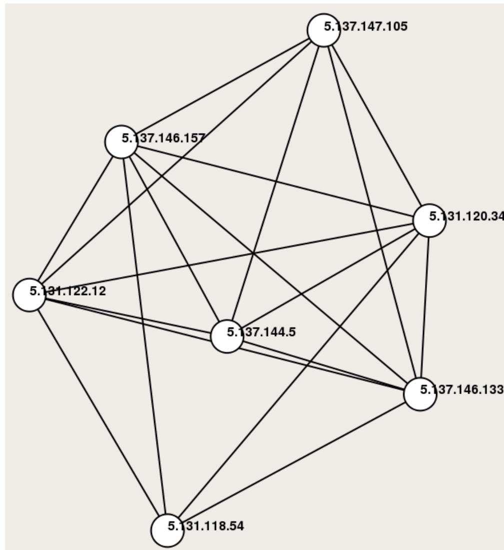


Figure 5.3: State of the testbed view by the application.

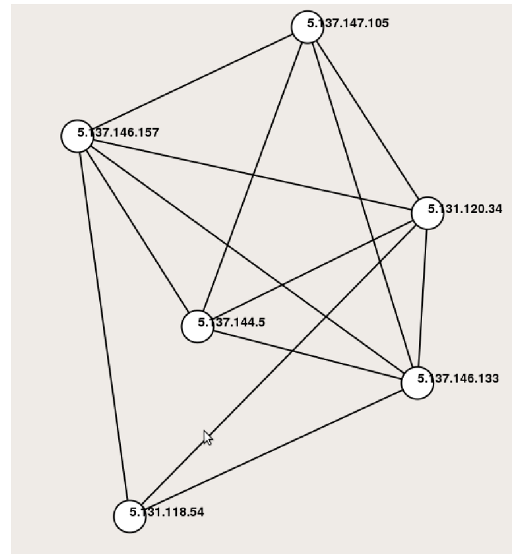


Figure 5.4: State after the reboot of one node.

As the graphical representation of the network seems to be accurate, we now discuss the tests.

5.5.2 Node down

In this test, we simulate the most common change in networks, a node going down. The cases can be multiple:

1. Reboot for maintenance.
2. Problem with or failure of the power supply.
3. Wireless interferences resulting in the node becoming isolated.

From the topology state, see Fig. 5.3, we simulate the reboot of node `5.131.122.12`. We observe the following timing (the nodes and the topology server have the same clock synchronization):

State	Time
Beginning of the reboot	11:06:20
Observation of the down state	11:08:18
Observation of the re-up state	11:11:53

Table 5.1: Node down simulation: timing.

There are approximatively 118 seconds between the rebooting and the observation of the change by the monitoring tool. This result can be explained based on the OLSR configuration of the routers, see Sec. 5.3.1.5.

The `HelloValidityTime` configuration defines the time, in second, between the reception of the last Hello message and the detection of a down neighbor. This means that if a node is unavailable, its neighbors are still sending TC messages containing it, during the next 108 seconds. The other 10 seconds are likely due to delays of messages, as `5.131.122.12` is relatively far from the sniffer (there are separated by two floors).

We can observe also that there are approximatively 333 seconds between the reboot time and the reappearance of the node. This can be explained by the `TcValidityTime` configuration of the OLSR configuration. This configuration defines the validity time to be announced in TC messages generated by this host, here 324 seconds. Therefore, when the router reboots, it need to wait this amount of time before its neighbors accept its new TC messages.

5.5.3 Packet loss

We simulate a 70% packet loss on node `5.131.120.34`, this means that 70% of packets that go through this node are dropped. The `tc` command executed to apply this feature is:

```
tc qdisc add dev ath0 root netem loss 70%
```

Code 12: Command to apply a 70% packet loss on a node.

We can see the result of this simulation on Fig. 5.5. This simulation is interesting to see if the implementation of the coverage algorithm is functional. As most of the Hello messages still pass, the routing protocol may not see the difference.

The topology contains only few paths of length greater than 2. In Fig. 5.5, 3 paths are used to cover the graph. All those paths pass through node `5.131.120.34`. We can see that the ping tests failed on all the paths, with a success rate of 80% in the best case (on a sample of 5 ping tests). This is the result expected by this simulation, as those paths are used within the routing protocol but not functional.

A way to avoid this routing protocol issue is to reconfigure it by putting a smaller `HelloValidityTime`. In this case the routing protocol may detect the loss of Hello packets.

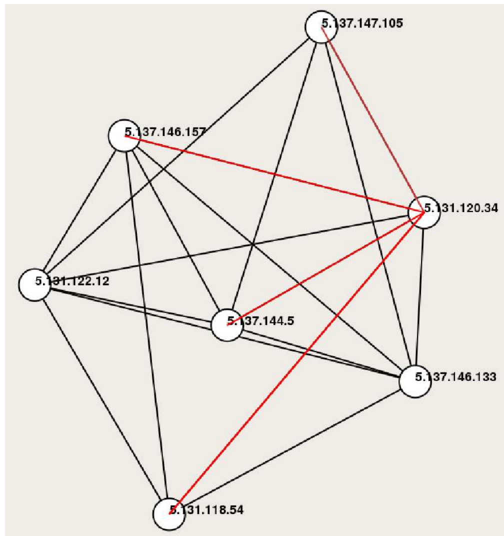


Figure 5.5: State after the simulation of 70% packet loss.

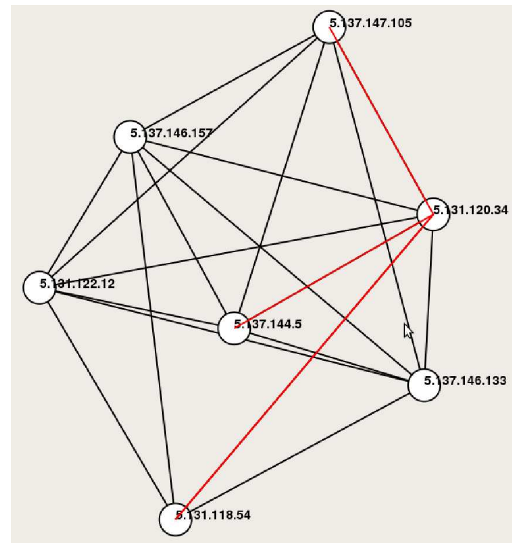


Figure 5.6: State after the simulation of an increasing of 1000ms in the delay on one node.

5.5.4 Delay

The last test is similar to the previous one, we simulate a increase in the delay of all packet passing through a node.

```
tc qdisc add dev ath0 root netem delay 1000ms
```

Code 13: Command to apply an increase of 1000ms delay on packets passing through a node.

Results for the 1000ms delay in node `5.131.120.34` can be viewed in Fig. 5.6. In this case, the interesting part is to see how much ping have failed. Two paths are used to cover (they both contain the node) the graph and both result in a 100% loss of the ping tests. The same correction to the OLSR configuration, as for the packet loss test, can be made in order to eradicate this problem, i.e., the decreasing of the `HelloValidityTime` configuration.

Results for the 1000ms delay in node `5.131.128.248` can be viewed in Fig. 5.7. It shows the evolution of pings RTT delay. At approximatively 400 seconds the delay in node `5.131.128.248` is increased by 100ms. Results after 400 seconds confirm that the probing technique works as it shows an increase of approximatively 100ms in the probing delays.

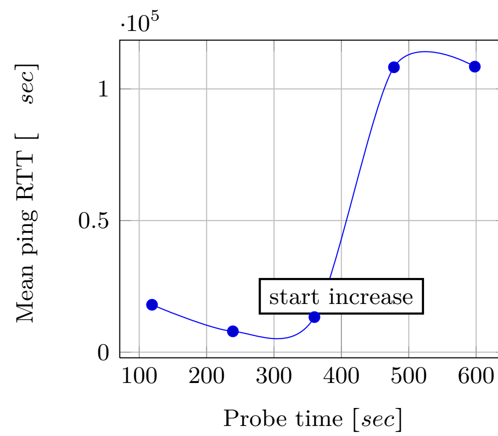


Figure 5.7: Probing delay evolution when an increase of 100ms delay is applied (at 400sec).

5.6 Conclusion

In this chapter, we have tested an application developed to meet our three objectives. We also test this application with possible WMN bad behavior. It comforts ourselves in the assumption that we made, in Chap. 3, about the behavior of routing protocols in WMNs. Even though, we cannot conclude that their are fully respected, as a larger testbed would be useful to completely validate our solution.

Chapter 6

Conclusion and future work

The main objective of this thesis was to develop a way for WMN maintainer to be aware of problems, in order to be able to solve them. At the end of the work, we think this objective is fulfilled. The topology coverage can indeed detect issues in a network, even when routing protocols cannot.

The dynamic character of WMNs involves that we need to take it into account in the objectives decision. Before deciding the main objectives we decided to limit ourselves to the monitoring of OLSR, which is the most commonly used in WMN. We defined three main objectives.

The first one is to provide real-time and efficient monitoring of routing protocols behaviors, this is the basis of all network monitoring. The solution chosen to solve this objective is to include a special node in the network that listens to the exchanged messages, in order to build its own view of the topology. Some research [SGG+02, SG04], on OSPF, shows that this solution to be the best available for link state routing protocols, which is the case of OLSR.

Our second objective is to limit the amount of monitoring messages. Of course the available bandwidth on wireless devices is smaller than in wired router. To avoid interference with the WMNs client traffic, while still keeping an efficient monitoring of the network, the amount of message must be small. Small enough such that we minimize the bandwidth consumption while we have enough information on the network status. To avoid monitoring overhead, there is only one solution: playing

with the monitoring frequency. A high frequency allows one to have an almost perfect view of the state of the network, but it also increase the bandwidth usage. A small frequency allows one to limit interferences, however, the network image is not accurate.

Our last objective was the main part of this thesis: accurately validating the network state viewed by the routing protocol. Due to the highly dynamic characteristic of the WMNs, routing protocols may not accurately view the topology. They may take path where issues arise, such as a link where the nodes are far from each other, i.e., where the loss rate is high. We propose a new theory: the *topology coverage*. Its goal is to provide a way to tests links in order to detect failure, or malfunctioning path, etc. From the graph theory we defined our topology coverage problem. This problem is to find a subset of path from a set of possible route through the network. This subset has to cover the topology, i.e., each links needs to be covered by at least one path of the subset. From this problem we firstly define one centralized algorithm. This algorithm is based on simple heuristic informed search algorithm. The idea is to start from an empty set of path and increase its size with the set of possible route until a coverage is found. We simulate several cases in order to validate this algorithm. The algorithm works good but has a major flaw: it is time greedy. For a large-scale network, it takes too much time to compute the coverage of the graph. As WMNs are dynamic we want to design an algorithm such that no or less changes my happen between while the computation.

Because of this flaw, we defined another algorithm, this time not purely centralized but almost distributed. It cannot be considered as a distributed algorithm as it still depends on a single node to do a simple computation: the graph partitioning. This algorithm is in fact a decentralized way of computing the coverage. The graph is partitioned in cluster, such that, in each cluster, a node (called the leader) runs the centralized algorithm in order to cover its own cluster. As the time greediness of the centralized algorithm is caused by the highly number of nodes and edges in the network, a smaller cluster coverage takes a smaller time. We evaluate this latter algorithm based on different possible topologies. As a result of the cluster coverage, the time is effectively decreased. It now takes an acceptable amount of time to cover the graph. The cluster size has also an impact on the quality and the rapidity of the algorithm. If the cluster size is small, a solution will quickly be found, however

the path used will be too small, resulting on a bad coverage. The cluster size must then be taken with care, we define a 20 nodes cluster size to be a good trade-off between quality and speed. We, finally, evaluate the algorithm ability of coverage, and we conclude that it has almost the same characteristic behavior as the centralized algorithm, without the time computation problem.

We define a practical implementation of these objectives. This application has not the goal to fully test the expected behavior of the network, but it is a start for an heavy battery of tests. The main outcome of the tests is the validation of third objective expected. When the network is changing, we simulate a 70% packet loss on a node, the routing protocol does not detect the failure and keeps using the malfunctioning links. However, the coverage algorithm allows us to detect the failure and to remedy, e.g., by moving the incriminated node.

6.1 Further work

There might be several topics where the topology coverage can be improved. Overloading is a big issue in the algorithm. Even though it is not too important, it must be problematic on most sensible nodes, such as a Gateway. It could be interesting to develop an heuristic taking into account this type of issues, by, e.g., have the possibility to give a cost at each node, in order to give more attribution to the ones that have a small cost, etc.

Another improvement of the algorithm is of course making it purely distributed, such that the cluster partitioning and leader election are done automatically.

One of the objective of this work was not to build a commercial application but to build the foundation of possible WMN monitoring systems. The objective is fulfilled and so this contribution could be a good starting point for further work on this subject.

Bibliography

[A. 09] A. Tønnesen and T. Lopatic and H. Gredler and B. Petrovitsch and A. Kaplan and S.-O. Tücke and others. OLSR – an ad-hoc wireless mesh routing daemon, 2009. http://www.olsr.org.

[AB00] R. Albert and A.-L. Barabási. Topology of Evolving Networks: Local Events and Universality. *Physical Review Letters*, 85(24):5234–5237, December 2000.

[Ans09] A. Anselmi. ROBIN - Open Source Mesh Network Project, 2009. http://www.blogin.it.

[ASNN07] J. Abley, P. Savola, and G. Neville-Neil. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095, Internet Engineering Task Force, December 2007.

[AWD04] M. Abolhasan, T. Wysocki, and E. Dutkiewicz. A review of routing protocols for mobile ad hoc networks. *Ad Hoc Networks*, 2:1–22, January 2004.

[AWW05] I. F. Akyildiz, X. Wang, and W. Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445–487, March 2005.

[bri07] BRITE: Boston University Representative Internet Topology gEnerator, 2007. http://www.cs.bu.edu/brite/.

[BT02] T. Bu and D. Towsley. On distinguishing between Internet power-law topology generators. In *Proc. IEEE INFOCOM*, June 2002.

[CFSD90] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157, Internet Engineering Task Force, 1990.

[CJ03] T. H. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626, Internet Engineering Task Force, October 2003.

[DL93] M. Doar and I. M. Leslie. How bad is naive multicast routing? In *Proc. IEEE INFOCOM*, March 1993.

[Don80] A. Donald. An upper bound for the path number of a graph. *Journal of Graph Theory*, 4(2):189–201, 1980.

[EGH] J. Ellson, E. Gansner, and Y. Hu. Graphviz - Graph Visualization Software. http://www.graphviz.org.

[ER59] P. Erdos and A. Renyi. On Random Graphs. I. In *Pub. Math. Debrecen*, volume 6, pages 290–297, 1959.

[Flo62] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[GMCN08] D. Gupta, P. Mohapatra, and C. Chen-Nee. Efficient monitoring in wireless mesh networks: Overheads and accuracy trade-offs. *Proc. IEEE Mobile Ad Hoc and Sensor Systems (MASS)*, October 2008.

[gti00] Modeling Topology of Large Internetworks, 2000. http://www.cc.gatech.edu/projects/gtitm/.

[Hag08] Aric Hagberg. NetworkX v0.99, 2008. http://networkx.lanl.gov/.

[Hop00] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, Internet Engineering Task Force, November 2000.

[Hub03] B. Hubert. Linux Advanced Routing and Traffic Control, 2003. http://www.lartc.org.

[IF05] L. Iannone and S. Fdida. MeshDV: A Distance Vector mobility-tolerant routing protocol for Wireless Mesh Networks. In *Proc. IEEE Workshop on Multi-hop Ad hoc Networks (RealMAN)*, July 2005.

[ips09] Cisco IOS IP Service Level Agreements (SLAs), 2009. http://www.cisco.com/en/US/products/ps6602/products_ios_protocol_group_home.html.

[JS03] J. Jun and M. L. Sichitiu. The nominal capacity of wireless mesh networks. *IEEE Wireless Communications*, 10(5):8–14, October 2003.

[Kar08] G. Karypis. Metis - Family of Multilevel Partitioning Algorithms, 2008. http://glaros.dtc.umn.edu/gkhome/views/metis.

[KL70] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, February 1970.

[LH98] T. Larsson and N. Hedman. Routing Protocols in Wireless Ad-hoc Networks - A Simulation Study. Master's thesis, Luleå University of Technology, Stockholm, Sweden, 1998.

[MG09] L. MG. tcpdump/libpcap, 2009. http://www.tcpdump.org.

[MqCh06] Z. Ming-qing and L. Chang-hong. Path Decomposition of Graphs with Given Path Length. *Acta Mathematicae Applicatae Sinica (English Series)*, 22(4):633–638, october 2006.

[NALW08] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich. Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.). Internet Draft (Work in progress) draft-wunderlich-openmesh-manet-routing-00, Internet Engineering Task Force, April 2008.

[NBB+07] D. Naudts, S. Bouckaert, J. Bergs, A. Schoutteet, C. Blondia, I. Moerman, and P. Demeester. A wireless mesh monitoring and planning tool for emergency services. In *Proc. IEEE Workshop on End-to-End Monitoring Techniques and Services*, May 2007.

[NK08] S. Nanda and D. Kotz. Mesh-Mon: A Multi-Radio Mesh Monitoring and Management System. *IEEE Computer Communications*, 31(8):1588–1601, May 2008.

[ope08] Open-Mesh Network, 2008. http://www.open-mesh.com.

[ope09] OpenWrt - Wireless Freedom, 2009. http://openwrt.org.

[Pac09] Packet Design Inc. Route Explorer, 2009. http://www.packetdesign.com/products/rex.htm.

[PBRD03] C. E. Perkins, E. M. Belding-Royer, and S. R. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, Internet Engineering Task Force, July 2003.

[Pos81] J. Postel. Internet Control Message Protocol. RFC 792, Internet Engineering Task Force, September 1981.

[Pyb96] L. Pyber. Covering the edges of a connected graph by paths. *J. Comb. Theory Ser. B*, 66(1):152–159, January 1996.

[pyt09] Python Programming Language, 2009. http://www.python.org.

[Quo09] B. Quoitin. iGen - Topology generation through network design heuristics, 2009. http://www.info.ucl.ac.be/~bqu/igen/.

[RN03] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[Rou08] P-E. Le Roux. MeshDV. Technical Report 2.0, University Paris 6, Pierre et Marie Curie, France, January 2008.

[rub09] Ruby Programming Language, 2009. http://www.ruby-lang.org.

[SFQ+07] F. Sailhan, L. Fallon, K. Quinn, P. Farell, S. Collins, D. Parker, S. Ghamri-Doudane, and Y. Huang. Wireless Mesh Network Monitoring: Design, Implementation and Experiments. In *Proc. IEEE Distributed Autonomous Network Management workshop (DANMS)*, November 2007.

[SG04] A. Shaikh and A. Greenberg. OSPF Monitoring: Architecture, Design and Deployment Experience. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[SGG+02] A. Shaikh, M. Goyal, A. Greenberg, R. Rajan, and K.K. Ramakrishnan. An OPSF Topology Server: Design and Evaluation. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(4):746–755, May 2002.

[SIG+02] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. A Case Study of OSPF Behavior in a Large Enterprise Network. In *Proc. ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2002.

[TGJ+02] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Network Topology Generators: Degree-Based vs. Structural. In *In Proc. ACM SIGCOMM*, Augustus 2002.

[TR] N. Teypaz and C. Rapine. Graph decomposition into paths under length constraints. BQR INPG 'Optimisation du transport de fret par l'utilisation de plateformes logistiques'.

[Tø04] A. Tønnesen. Implementing and extending the Optimized Link State Routing Protocol. Master's thesis, UniK University Graduate Center, University of Oslo, Norway, August 2004.

[Var96] Y. Vardi. Network Tomography: Estimating Source-Destination Traffic Intensities from Link Data. *Journal of the American Statistical Association*, 91(433):365–377, 1996.

[Wax88] B. M. Waxman. Routing of Multipoint Connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, December 1988.

[wif07] IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Technical report, 2007.

[wim04] IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 16: Air Interface for Fixed Broadband Wireless Access Systems. Technical report, 2004.

[WJL03] D. Watson, F. Jahanian, and C. Labovitz. Experiences With Monitoring OSPF on a Regional Service Provider Network. In *ICDCS '03:*

Proceedings of the 23rd International Conference on Distributed Computing Systems, page 204, Washington, DC, USA, 2003. IEEE Computer Society.

Appendix A

Decentralized Algorithm: Evaluation

This appendix contains complementary figures for the evaluation of the distributed algorithm 4.5.1, in Chap. 4, Sec. 4.5.5.2.

The figures are here for information, they are not fully described.

A.1 Heuristics evaluation

To evaluate which heuristic is the best one, we need to compare them. The following sections display simulation results for three topology families.

We can see that there are few or no differences between the heuristic $h_{basic-limit}$ and $h_{overcover-limit}$, as they try to limit the path length used. $h_{overcover}$ gives good results in each family, excepted for the max number of paths for each edges. This behavior is normal, as it do not limit the path length, it has no information about the other clusters and thus may chose one path that contains an edge already covered in another cluster.

A.1.1 Regular networks

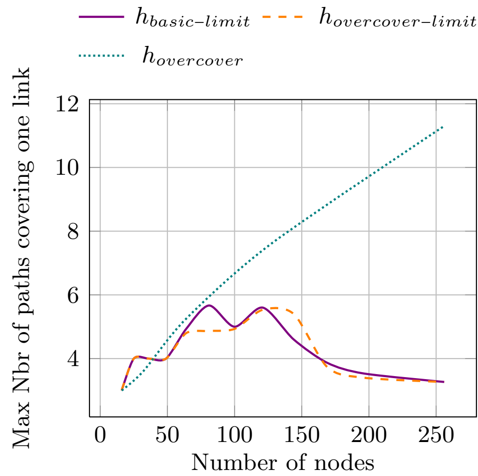


Figure A.1: Manhattan – Comparison of the Maximum number of paths covering each edge when the three heuristics are used.

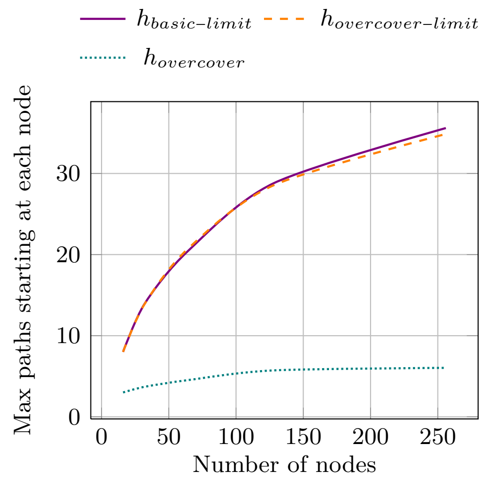


Figure A.2: Hypercube – Comparison of the Maximum number of paths starting at each node when the three heuristics are used.

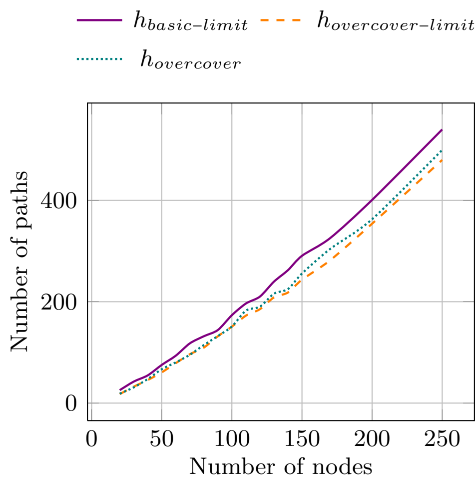


Figure A.3: Full mesh – Comparison of the number of paths when the three heuristics are used.

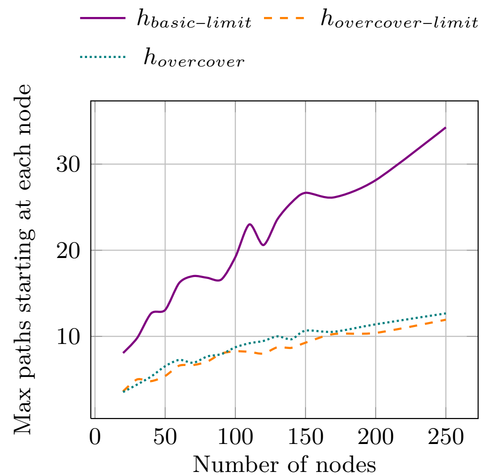


Figure A.4: Full mesh – Comparison of the Maximum number of paths starting at each node when the three heuristics are used.

A.1.2 Structural generators

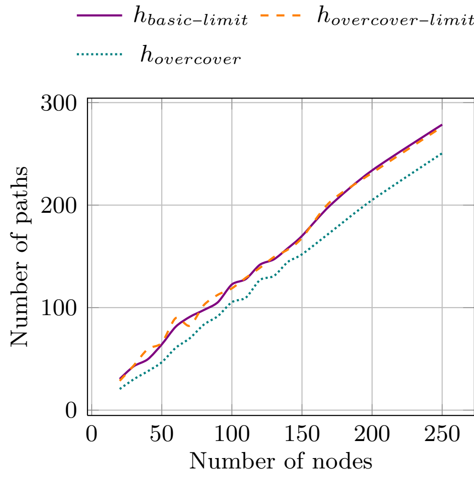


Figure A.5: Hierarchical Top-Down – Comparison of the number of paths when the three heuristics are used.

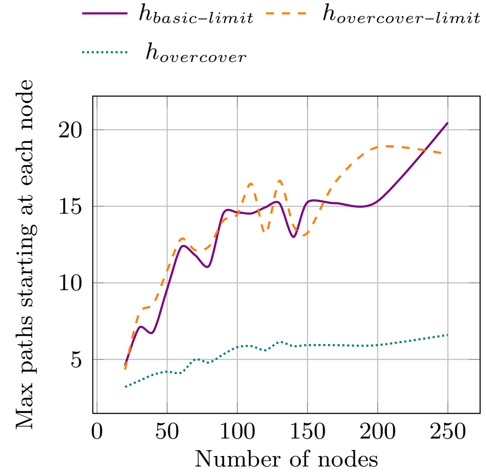


Figure A.6: Hierarchical Top-Down – Comparison of the Maximum number of paths starting at each node when the three heuristics are used.

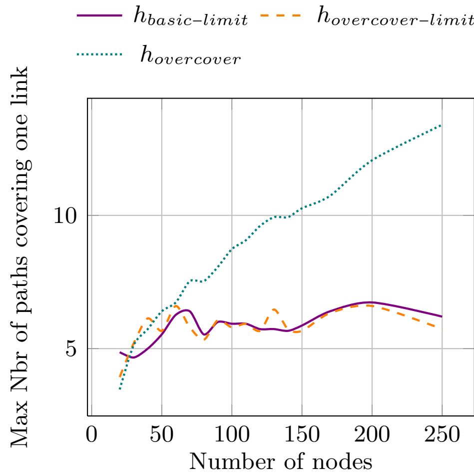


Figure A.7: Hierarchical Top-Down – Comparison of the Maximum number of paths covering each edge when the three heuristics are used.

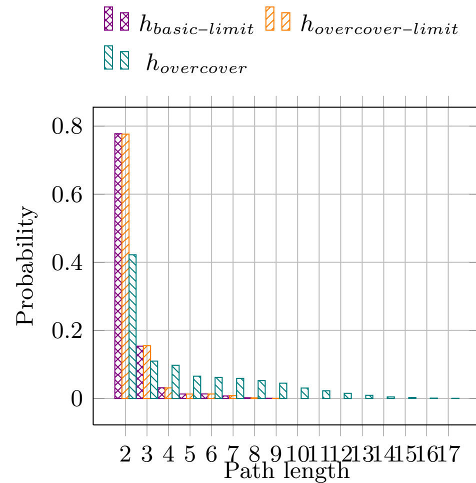


Figure A.8: Hierarchical Top-Down – Length distribution of path used to cover the graph.

A.1.3 Degree-based generators

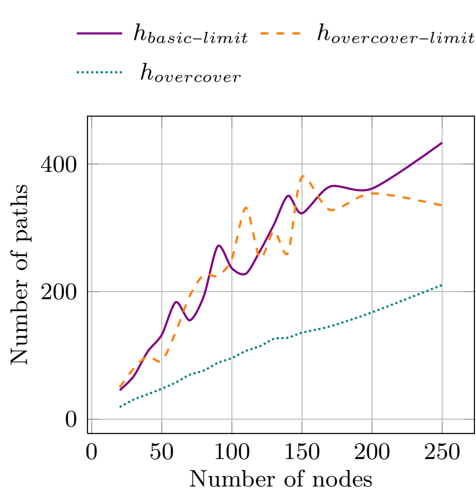


Figure A.9: BA – Comparison of the number of paths when the three heuristics are used.

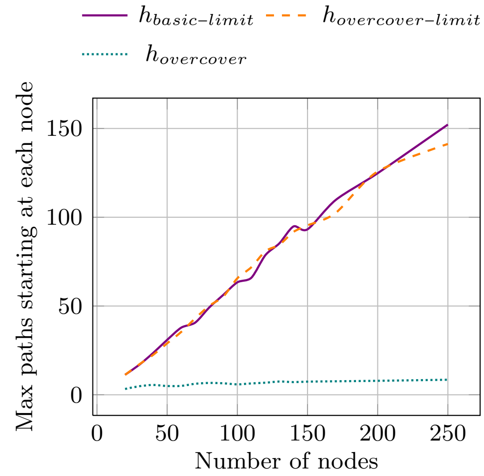


Figure A.10: GLP – Comparison of the Maximum number of paths starting at each node when the three heuristics are used.

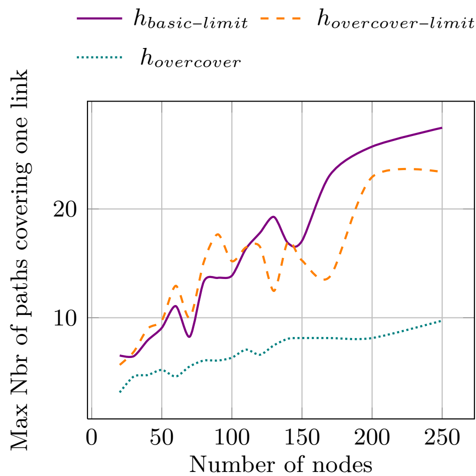


Figure A.11: GLP – Comparison of the Maximum number of paths covering each edge when the three heuristics are used.

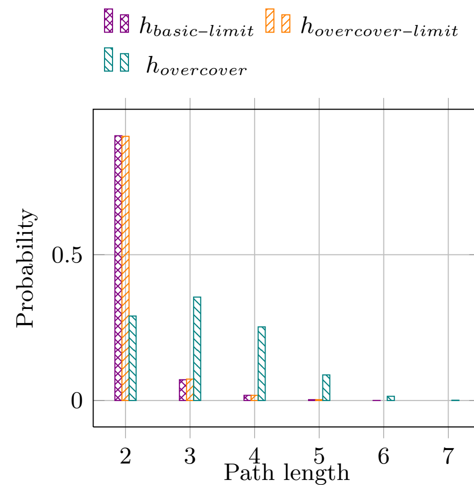


Figure A.12: GLP – length distribution of path used to cover the graph.

A.2 Coverage quality

The main evaluation of the algorithm is to measure the ability to cover the topology graph. The behavior is very similar to the one developed in Chap. 4, Sec. 4.5.5.2.

A.2.1 Regular networks

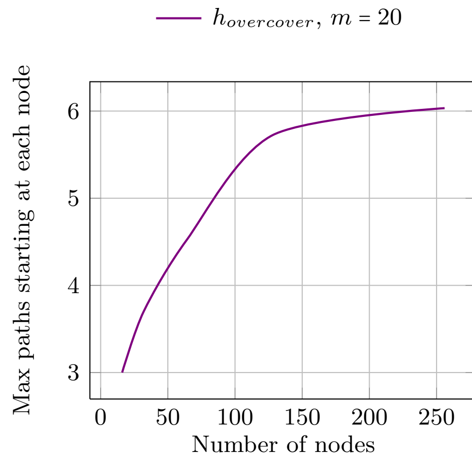


Figure A.13: Manhattan – Maximum number of different paths starting at each node.

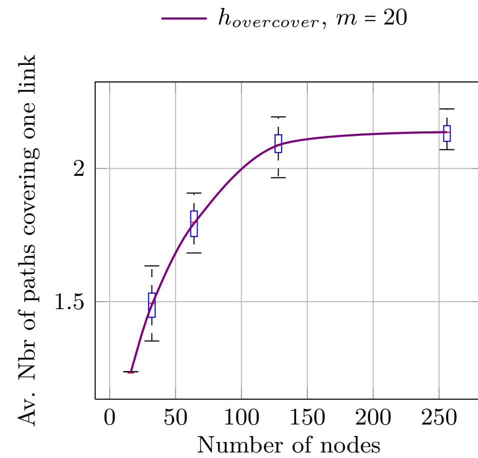


Figure A.14: Hypercube – Average number of times each edges is over-covered.

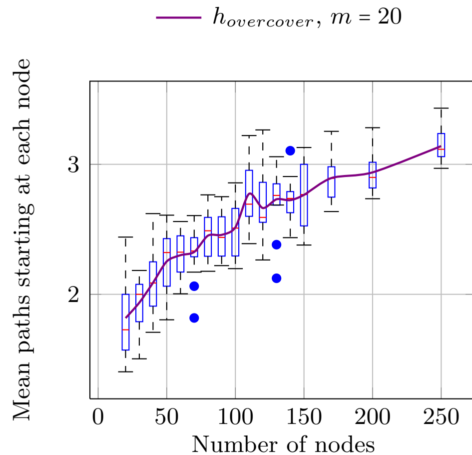


Figure A.15: Full mesh – Average number of different paths starting at each node.

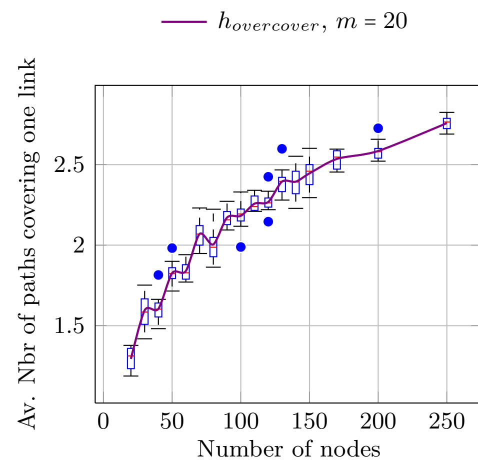


Figure A.16: Full mesh – Average number of times each edges is over-covered.

A.2.2 Structural generators

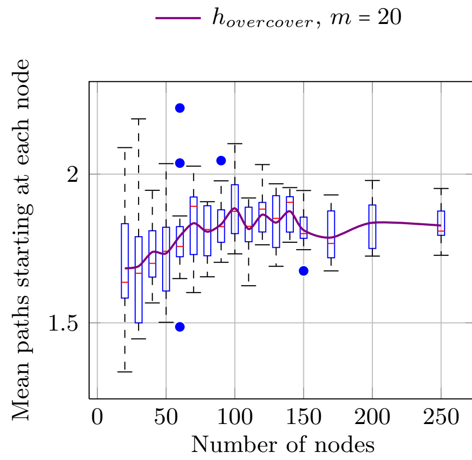


Figure A.17: Hierarchical Top-Down – Average number of different paths starting at each node.

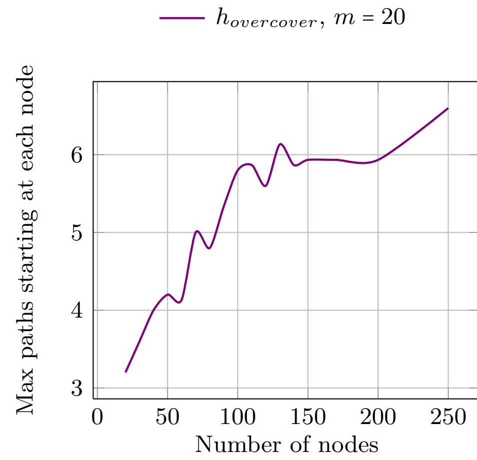


Figure A.18: Hierarchical Top-Down – Maximum number of different paths starting at each node.

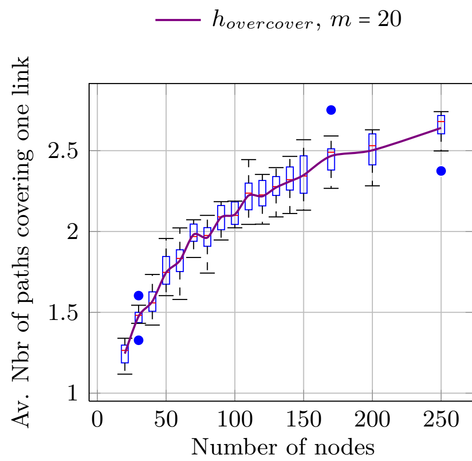


Figure A.19: Hierarchical Top-Down – Average number of times each edges is over-covered.

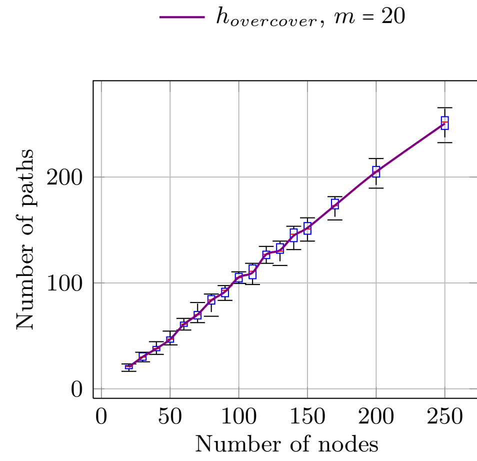


Figure A.20: Hierarchical Top-Down – Number of paths used to cover the topology graph.

A.2.3 Degree-based generators

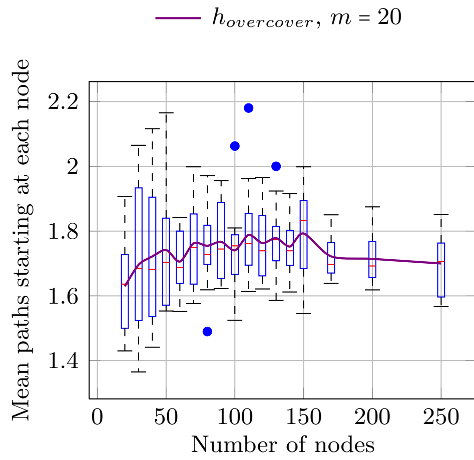


Figure A.21: BA – Average number of different paths starting at each node.

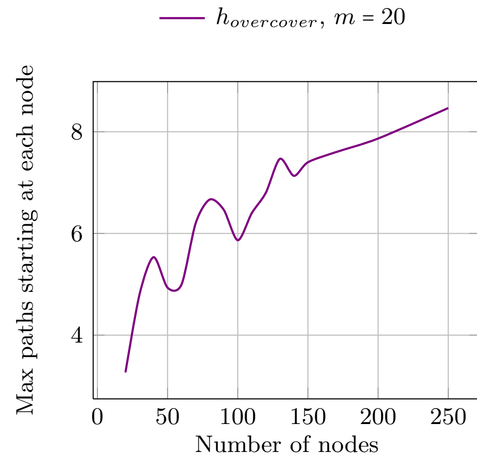


Figure A.22: GLP – Maximum number of different paths starting at each node.

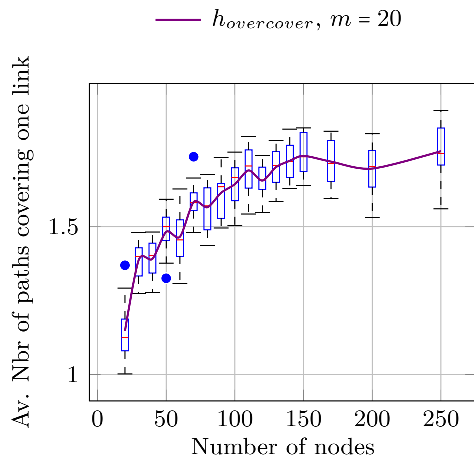


Figure A.23: GLP – Average number of times each edges is over-covered.

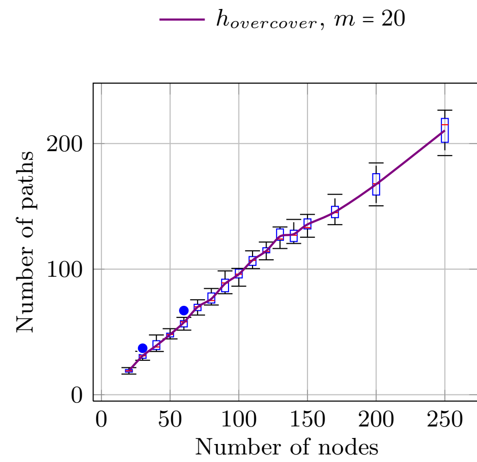


Figure A.24: BA – Number of paths used to cover the topology graph.

Appendix B

A practical application & simulation: Source code

The source code is available and fully described in the CD-ROM provided with this document.