Sapienza University of Rome

Department of Computer Science

PhD Thesis

Network Performance Analysis through Boolean Network Tomography and Parallelization of Fundamental Operations in Numerical Linear Algebra

Candidate:
Viviana Arrigoni

Advisor:
prof. Annalisa Massini

In partial fulfillment of the requirements for the degree of Doctor of Philosophy - XXXIII cycle
Academic year 2019-2020

Thesis defended on 8th July 2021
in front of a Board of Examiners composed by:

Prof. Maurizio Bonuccelli (chairman)
Department of Computer Science
University of Pisa, Italy

Prof. Dario Catalano
Department of Mathematics and Computer Science
University of Catania, Italy

Prof. Andrea Marin
Department of Environmental Sciences, Informatics and Statistics
Ca' Foscari, University of Venice, Italy

External reviewers:

Prof. Francesco Lo Presti
Civil Engineering and Computer Engineering Department
Tor Vergata University of Rome, Italy

Prof. Simone Silvestri
Computer Science Department
University of Kentucky, Lexington, KY, USA

Prof. Murat Manguoğlu
Department of Computer Engineering
Middle East Technical University, Ankara, Turkey

Thesis committee:

Prof. Annalisa Massini (Advisor)
Prof. Novella Bartolini
Prof. Adolfo Piperno

Network Performance Analysis through Boolean Network Tomography and Parallelization of Fundamental Operations in Numerical Linear Algebra

Ph.D. thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX.

Version July 2021

Author's email: arrigoni@di.uniroma1.it

to my family

Abstract

This thesis collects, in a unified framework, two cores, reflecting the dual nature of my research activity. During my Ph.D., I had the chance to explore different branches of knowledge in Computer Science, and this thesis focuses on the two disciplines where my work was more fertile, that are respectively Boolean Network Tomography and Numerical Linear Algebra and High Performance Computing. Despite these two branches are orthogonal to one another in the fields of application of this thesis, they share a common ground as numerical Linear Algebra is often evoked for solving problems in Optimization, Graph Theory and Compressed Sensing, that are in turn exploited in Boolean Network Tomography with the scope of analysing network performance. In addition, both two disciplines share a multi-disciplinar background; the first one, in terms of the combinatorial and probabilistic analysis that is usually required to interpret data acquired through Boolean Network Tomography techniques, the second one for its vast field of application, including Engineering and scientific modelling of complex systems.

Contents

Introduction

This thesis is divided into two parts. In the first part, we propose novel results in the context of Boolean Network Tomography. We briefly introduce them in Section I, and we show further detailed insights in Chapter 1. The second part of this thesis provides algorithmic and implementation solutions to fundamental problems of numerical Linear Algebra and High Performance Computing. We devote Section II to presenting the research scenario where these results belong, and Chapter 5 to a thorough introduction of such topics

I Network Performance Analysis through Boolean Network Tomography

Network management is a challenging task due to lack of centralized control and as a consequence of heterogeneous technologies employed in networks. In decentralized networks, commercial and administrative factors often preclude different organizations from sharing topology and performance information, causing limitations to the possibility of cooperating for an effective network management able to guarantee Quality of Service (QoS), Service Level Agreement (SLA) verification, malicious traffic prevention, [99]. On the other hand, as distributed systems are a pillar framework for telecommunication networks, network applications, real-time process control and parallel computation, it is crucial to guarantee their reliability. However, this is usually a difficult endeavour, since their complex and heterogeneous architectures and communication mechanisms hinder a comprehensive ability to deploy and monitor the underlying networks. Network Tomography refers to the methodology of inferring properties of a network by means of end-to-end measurements. In analogy with tomographic techniques used in medicine, Network Tomography uses end-to-end measurements scanning the inner part of the network to retrieve information about the internal nodes and links without direct inspections [141]. End-to-end measurements do not rely on administrative access privileges, and are agnostic to network heterogeneity, [83]; as a matter of fact, they do not hinge on specific operating systems, hardware, communication protocols etc., that instead represent constraining specifications for classical monitoring systems. For these reasons, data provided by end-to-end measurements is easily accessible and may be collected passively (by observing existing traffic flow data) or actively (by injecting trackable monitoring packets which are served as regular users traffic). Network tomography techniques can be classified depending on the objective metrics inference: in the first part of this thesis we focus on *Boolean Network Tomography (BNT)*, a field of network tomography where link, node and path metrics have a binary classification. In particular, we tackle the problem of node identification and failure detection through binary measurements taken along paths between peripheral nodes. Chapter 1 introduces this topic in detail and affords background notions that will recur throughout the first part of this thesis.

By means of simple examples, we highlight challenges and goals related to obtaining and interpreting performance data acquired through techniques of BNT. Specifically, we show that the number of possible admissible classifications is potentially exponential. In this context, we will introduce a key concept in Boolean Network Tomography, *identifiability*,

that refers to the possibility of inferring with certainty the binary status (failed/working) of a node by means of end-to-end paths. Driven by these motivations, in Chapters 2 and 3, we provide fundamental bounds to the number of identifiable nodes in a network, taking into account QoS requirements, including routing and path length. The bounds proved in these chapters allow to evaluate the performance of a given monitoring scheme, to size the monitoring system in terms of the number of resources to allocate, and to extract guidelines for designing the most suitable topologies for failure localization. In addition, we also define an algorithmic approach for path deployment that meets the bounds tightly.

Our topology-agnostic bounds are fundamental descriptors of the theoretical limits of network identifiability when specific resources are available. In Chapter 4 instead, we move on to the problem of detecting failed nodes in a network by optimizing the available resources. We pursue this goal by defining a stochastic maximization problem that can be synthesized as follows: given a network and paths defined by an uncontrollable routing scheme, find the maximum number of defective nodes with the least number of path probes (i.e., limiting the injected traffic as much as possible). To solve this problem, we define a greedy algorithm based on a Bayesian analysis that is able to retrieve all the information that can be possibly obtained using Boolean Network Tomography (i.e., probing all available paths), by scheduling only a small portion of path probes. In addition, our algorithm uses the probabilistic analysis that is carried out throughout the scheduling phase, to return a failure probability distribution on the nodes of the given network.

II Parallelization of Fundamental Operations in Numerical Linear Algebra

The second part of this thesis is devoted to my research in the field of Numerical Linear Algebra and super-computing. Numerical Linear Algebra algorithms are often evoked in the most disparate fields (including Computer Vision, Computer Graphics and Geometry processing, as well as in simulations of complex scientific and engineering systems) as they provide fast solutions to problems whose symbolic counterparts have prohibitive computational costs. Despite its long history, dating back to the 1940s, many researchers still devote their efforts to devising efficient numerical algorithms, and to providing fast, parallel implementations that should have the ability to adapt to the growing industry of High Performance Computing (HPC) architectures. In this thesis, we focus on the parallel implementation of fundamental operations of numerical Linear Algebra: matrix multiplications and linear solvers. These operations often represent a bottleneck when they are included in more complex frameworks, specially when realized exploiting distributed architectures, because of the strong dependence between data, that implies inherent communication overhead. Part II of this thesis is dedicated to this topic, that is introduced in detail in Chapter 5, where we also provide fundamental background knowledge on the problems that shall be tackled in the second part of this thesis.

In Chapter 6, we extensively study solutions for the matrix multiplication $\mathbf{A}^T\mathbf{A}$. This operation appears as an intermediate step in several applications, including base orthonormalization, least-square problem and singular value decomposition. The chapter collects a number of significant results: first of all, we introduce AtA, a sequential algorithm, applicable to any matrices, whose computational complexity is reduced with respect to other known algorithms. In addition, we describe in detail a shared-memory and a distributed-memory implementations for $\mathbf{A}^T\mathbf{A}$, optimizing memory storage requirements and minimizing the communication cost.

In Chapter 7, we provide an efficient solution for specific linear systems arising from simulations of complex systems modelled by Finite Element Methods. Their evolutionary process is captured by large systems of differential equations, that are not solvable analyti-

cally. To study their behaviour, numerical approaches integrating multiple linear systems are usually exploited. In this chapter, we provide an implementation of a MPI/OpenMP hybrid solver for systems that we call *Quasi-Block Diagonal (QBD)*. We take advantage of the shape of these systems in order to wisely distribute input data, so that communication is maximally reduced, making the solver almost embarrassingly parallel.

All the results collected in this thesis are formally proved and validated by means of experimental comparisons with state-of-the-art benchmark solutions. Most of them have been published in prestigious venues ([16], [5], [6], [8], [9]), and another one has been recently accepted as a conference paper ([7]). A summary on the literature related to each topic faced in this thesis is reported in every chapter.

Acronyms

BNT Boolean Network Tomography.

FaCeGreedy Failure Centrality Greedy.

GE Gaussian Elimination.

HPC High Performance Computing.

HPJ hybrid preconditioned Jacobi.

ICA Incremental Crossing Arrangement.

PDEs Partial Differential Equations.

PoPGreedy Posterior Probability Greedy.

QBD Quasi-Block Diagonal.

QoS Quality of Service.

SLA Service Level Agreement.

Part I

Network Performance Analysis through Boolean Network Tomography

Chapter 1

Introduction to Part I

Access to the Internet has dramatically changed the way people live worldwide. In every moment, it is possible to check on one's bank account, communicate with people living thousands of kilometers away, share contents, read the latest news, buy and sell items from e-commerce sites. The Internet has evolved from a small tightly controlled network serving only a few users in the late 1970s to be an immense multilayered collection of heterogeneous platforms in recent years, [99]. The evolution is still an active process: it is estimated that in 2019 the Internet users were 4.1 billions, reflecting a 5.3% increase with respect to 2018, [22]. In the same year, the number of active, mobile-broadband subscriptions per 100 inhabitants continued to grow strongly, with an 18.4% year-on-year growth, and with almost the entire world population living within reach of a mobile network, [22]. Behind this stands a massive network whose progressive and enormous growth is characterized by lack of centralized control and heterogeneous hardware and software components. Therefore, the Internet can be seen as a *network of networks*, each belonging to some commercial or government company and connected (directly or indirectly) with the others by means of several kinds of devices and communication technologies. Such companies and organizations have administrative access to only a small fraction of the network's internal nodes, whereas commercial factors often prevent them from sharing internal properties data (e.g., network topology and performance), [2]. As a consequence, making quantitative assessment of a network performance at a wider level is impracticable, but at the same time it is a crucial task for guaranteeing quality of service (QoS), verifying Service Level Agreements (SLAs), improving network management, enabling dynamic routing, and filtering anomalous or malicious traffic. The fact that many researches working for online retailers and advertisement companies (including Google and YouTube), in addition to research centres (e.g., CAIDA - Cooperative Association for Internet Data Analysis, website [26]) are devoted to developing Internet maps and networking tools is emblematic, [40, 75, 101].

Network Tomography was born with the wider goal of assessing topological and performance properties of a network by means of end-to-end measurements that do not rely on administrative access privileges, and hence, that are easily available. The term *tomography* was first introduced in this context by Y. Vardi in [141], because of the analogy between network inference through end-to-end paths scanning the internal components of the network and medical tomography[1]. Network Tomography takes up the challenge coming with the heterogeneous and largely unregulated structure of the Internet, and with the reluctance of cooperation of individual servers and routers to share structural and performance information,

[1] tomography: any of several techniques for creating three-dimensional images of the internal structure of a solid object by analyzing the propagation of waves of energy, such as x-rays or seismic waves, through the object, [110]

to infer the internal behavior and topology of the network.

Applications of Network Tomography include network topology discovery [29,32,42,52], estimation of the complete set of end-to-end measurements from an incomplete set, [30], derivation of path-level network parameters from measurements made on individual links/nodes, [141], and inference of link and node metrics through end-to-end measurements, that is the subject of this thesis. Possible link and node metrics that can be inferred through Network Tomography include delays due to congested links/nodes ([43, 44, 115]), packet losses ([45, 59, 117, 118]), due to defectiveness of network links or nodes, links bandwidth estimations ([87, 90, 126]). We talk about delay tomography, loss tomography or bandwidth tomography, respectively, [77]. Network tomography principles have been intelligently adopted in [147] to monitor city traffic and localize vehicles without GPS. Conversely, it exploits just a limited number of cameras placed at road intersections to measure car end-to end traveling times. Depending on whether explicit monitoring measurements are required, Network Tomography can be classified as *active* or *passive*; in the first case, monitoring messages are probed in the network in order to measure the characteristic of monitoring paths. In the second case, this is done through the analysis of passive traffic data flows.

In this thesis, we use Boolean Network Tomography (BNT) for detecting defective nodes in a network. In Boolean Network Tomography, pioneered by Duffield et. al. [46], a binary value is assigned to the outcome of measurement path probes and to links/nodes of the network. BNT is not only conveniently applied for inherently binary metrics inference (as for the scenario envisioned in this thesis where a node is either defective or working), but also for continuously valued metrics (such as link delay inference) where a threshold might be conveniently defined in order to divide the domain into two intervals corresponding to a binary classification (as for example in [74, 106]). This application of Network Tomography does not only provide solutions to problems related to the Internet network, but also represents a powerful tool for network maintenance, that allows to avoid expensive and possibly dangerous direct inspections on nodes when network cover vast geographical areas or when natural cataclysms cause node faults. Nodes and link failure detection through Boolean Network Tomography also represents a robust alternative to other network monitoring systems: complex and heterogeneous networks (including the Internet, hybrid optical/copper networks, future cellular networks, and distributed cloud networks) come with bugs and configuration errors in various customer software and network functions that often induce “silent failures”. Such phenomena are hardly detected by traditional network monitoring approaches based on pervasively deployed monitoring agents (e.g., SNMP) or pervasively supported network protocols (e.g., traceroute), and are only detectable from end-to-end connection states, [83]. In addition, in multi-domain systems some nodes might not be cooperative, and they can block frequent traceroute requests. Despite managers of autonomous networks have direct access to all the internal devices of the systems, they can still use Network Tomography for detecting and localizing possible vulnerabilities that a network outsider might infer via end-to-end paths.

In the following section, we introduce basic notions in Boolean Network Tomography. In particular, we introduce the concept of *node identifiability*, that is crucial for the purpose of this thesis. Together with providing fundamental insights on BNT, we show the challenges related to gaining a comprehensive knowledge of the state of the nodes (failed/working) of a network by means of measurement paths.

1.1 Preliminaries in Boolean Network Tomography

The goal of BNT is to use end-to-end measurement paths to assess the binary state of the internal nodes of a network. A network can be modelled with a graph $G = (V, E)$ where V

is the set of nodes and E is the set of edges. Measurements are taken by probing monitoring packets through paths that traverse nodes in V. We call P the set of all monitoring paths. Each path $p \in P$ can be represented as the set of nodes it traverses. In Boolean Network Tomography, the outcome of a path probe has a binary value, *working* or *failed*. A path is working if packets probed along it are correctly received by the end node of the path. In contrast, we say that a path is failed if packet losses occur within its nodes. A path is working if all the nodes it traverses are working, whereas it fails if at least one of the nodes it traverses is defective. This concept can be formally described as a system of Boolean equations, as follows. Let us give a Boolean value to the state of paths and nodes; in particular, we use 0 for 'working' and 1 for 'failed'. We denote with y_i the binary value of the outcome of path p_i, i.e.:

$$y_i = \begin{cases} 0 & \text{if path } p_i \text{ is working,} \\ 1 & \text{otherwise} \end{cases}$$

Similarly, we denote with x_j the binary value of the state of node v_j. We define the path matrix of a network the matrix $M \in \{0,1\}^{|P|\times|V|}$ such that $m_{i,j} = 1$ if node v_j belongs to path p_i, $m_{i,j} = 0$ otherwise. Hence we can define a system of Boolean linear equations relating paths and nodes' states together as follows:

$$y_i = \bigvee_{j=1}^{|V|} m_{i,j} x_j, \quad \forall i \in \{1, \dots, |P|\} \tag{1.1}$$

where $\vee$ is the logical OR operator. The system in Equation 1.1 is consistent (i.e., admits one and only one solution) if and only if all equations are linearly independent. If this is not the case, the number of solutions is potentially exponential, and in particular it is given by the number of failing paths (i.e., paths $p_i \, s.t. \, y_i = 1$) multiplied by the exponential of the number of nodes that each such path traverses and that do not appear in any other equation where the known term is 0. Concisely, the number of possible solutions is $O(\sum_{i=1}^{|P|} y_i \cdot 2^{|p_i|})$. This number can be limited if we have knowledge on the number of failed nodes (i.e., if we are also given the equation $\sum_{j=1,\dots,|V|} x_j = k$, for some natural $k \leq |V|$). The simple network in Figure 1.1 can help us understand the growth of the solution space of the system in Equation 1.1. The system 1.1 for this example is the following:

$$\begin{cases} y_1 = x_1 \vee x_5 \vee x_6 \vee x_2 \\ y_2 = x_1 \vee x_5 \vee x_6 \vee x_9 \vee x_{11} \vee x_3 \\ y_3 = x_1 \vee x_5 \vee x_8 \vee x_7 \vee x_4 \\ y_4 = x_2 \vee x_{10} \vee x_3 \\ y_5 = x_2 \vee x_6 \vee x_8 \vee x_{12} \vee x_4 \\ y_6 = x_3 \vee x_{11} \vee x_{12} \vee x_4 \end{cases}$$

We shall consider different failing scenarios.

Scenario 1. Firstly, let us assume the simple case where we have knowledge of the number of failed nodes, and let us consider the case where only one node fails. If we probe all available paths, $p_1, \dots, p_6$, and if all paths work except for path p_3, then we know that all nodes are working, except for node v_7. This is because v_7 is the only node that is being traversed by p_3 alone. If any other node laying on path p_3 failed, we would have observed failure of other paths, too.

Scenario 2. Again, let us consider the same scenario where we know that only one node in the network is defective, and assume that by probing all available paths, we observe that paths p_1, p_2 and p_3 fail. In this case, we can limit the possible failure set to nodes v_1 and v_5, but we cannot claim with certainty which of the two nodes failed.

Scenario 3. The capability of detecting failed nodes may be further limited when multiple failures occur concurrently. Assume we know that two nodes are defective, and again paths p_1, p_2 and p_3 fail. Then we have multiple possible failing scenarios that equivalently justify the observations, and that are given by the following sets of possibly failed nodes: $\{v_1, v_5\}$, $\{v_1, v_7\}$, $\{v_1, v_9\}$, $\{v_5, v_7\}$, $\{v_5, v_9\}$.
Scenario 4. Finally, assume that knowledge on the number of defective nodes is not given, and again assume that paths p_1, p_2 and p_3 fail. The family of the possible failure sets (i.e., set of possibly failed nodes) that implies failure of these paths is: $\{v_1\}$, $\{v_5\}$, $\{v_1, v_5\}$, $\{v_1, v_7\}$, $\{v_1, v_9\}$, $\{v_5, v_7\}$, $\{v_5, v_9\}$, $\{v_1, v_5, v_7\}$, $\{v_1, v_5, v_9\}$, $\{v_1, v_7, v_9\}$, $\{v_5, v_7, v_9\}$ and $\{v_1, v_5, v_7, v_9\}$. If all nodes are equally likely to fail, and if there is no evidence on the most probable number of failures, all these possible solutions are equally likely to be correct.

The concept of *identifiability*, introduced in [65], represents the capability of assessing without ambiguity the state of the nodes in a network. More specifically, we say that a node is k-identifiable if, when it fails, its state can be inferred by means of end-to-end paths when k failures occur in the network. We give the following formal definitions:

Definition 1.1.1. *A node v is* 1-identifiable *with respect to P if the set of paths that traverse it is different from the set of paths traversing every other node.*

Definition 1.1.2. *Given a set of measurement paths P, two failure set F_1 and F_2 are* distinguishable *with respect to P if exists at least a path $p \in P$ that fails under only one failure set, either F_1 or F_2.*

Definition 1.1.3. *A node v is* k-identifiable *with respect to P is for any couple of failure sets F_1 and F_2 with $|F_i| \leq k$, $i = 1, 2$ and $F_1 \cap \{v\} \neq F_2 \cap \{v\}$ (i.e., one including v and one not including it), F_1 and F_2 are distinguishable with respect to P.*

These concepts will be recalled in Chapter 2, where we will provide equivalent definitions based on combinatorial properties of identifiable nodes. Observe that 1-identifiability is a special case of k-identifiability, and that verifying k-identifiability requires exploring the subset of 2^V that includes sets of paths of size from 1 to k. Furthermore, it is easy to see that k-identifiability implies l-identifiability for all $l \in \{1, \ldots, k\}$.

Node identifiability is hardly satisfied by network topologies and routing schemes. In the example of Figure 1.1, all nodes, except for v_1 and v_5, are 1-identifiable, whereas no node is k-identifiable, with $k \geq 2$. Also knowledge on the number of failures is an information that is hardly available, despite such knowledge narrows down the solution space, as we have seen in the small example reported in this section. Nevertheless, even when

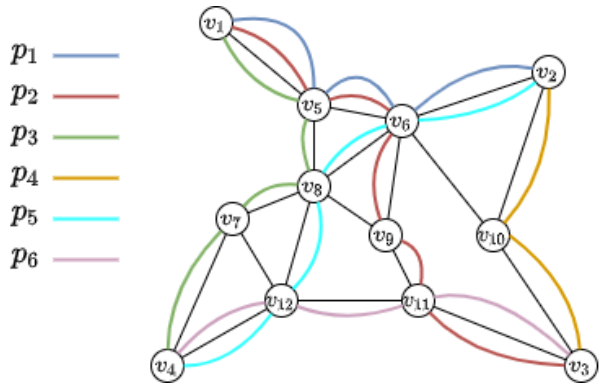


Figure 1.1

knowledge on the number of failed nodes is provided, the system in Equation 1.1 admits possibly exponentially-many solutions. The solution space can be limited by deploying more monitoring paths, nevertheless this is not always feasible. Furthermore, in [65], He et. al. prove that optimal monitor placement for node identifiability maximization is a NP-hard problem. For all these reasons, Boolean Network Tomography comes with great challenges when it is applied to networks where there is no room for active interventions, and when prior observations on the nodes' probability of failure are not available.
In this thesis, we provide theoretical bounds on network identifiability with different constraints on network resources and properties. Afterwards, we shall study how nodes state inference can be achieved with an intelligently chosen subset of all available paths. We also study a probabilistic model to expand BNT potential informativeness.
In this chapter and in the chapters that follow we refer to node failure identifiability and detection. We point our that all definitions and results collected in this thesis are also valid for link identifiability and link failure localization.

Chapter 2

Fundamental Identifiability Bounds in Boolean Network Tomography

In this chapter, we provide upper-bounds on the maximum number of identifiable nodes, given the number of monitoring paths and different constraints on the network topology, the routing scheme, and path length. These upper bounds represent a fundamental limit on identifiability of failures via Boolean Network Tomography. In this chapter, we also provide an algorithmic approach to the design of network topologies and path deployment that meet the discussed limits, under various network settings. Through analysis and experiments we demonstrate the tightness of the bounds and efficacy of the design insights for engineered as well as real networks. The contributions reported in this chapter are published in [16], where we extended the results published in [17].

2.1 Motivations

The capability to assess the states of network nodes in the presence of failures is fundamental for many functions in network management, including performance analysis, route selection, and network recovery. In modern networks, the traditional approach of relying on built-in mechanisms to detect node failures is no longer sufficient, as bugs and configuration errors in various customer software and network functions often induce "silent failures" that are only detectable from end-to-end connection states [83]. Boolean Network Tomography (BNT) [46] is a powerful tool to infer the states of individual nodes of a network from binary measurements taken along selected paths. We consider the problem of Boolean Network Tomography in the framework of graph-constrained group testing [31]. Classic group testing [11, 39] studies the problem of identifying defective items in a large set S by means of binary measurements taken on subsets $S_i \subseteq S$ $(i = 1, \ldots, m)$. Close to the problem of group testing, Boolean Network Tomography aims at identifying defective network items, i.e. nodes or links, in a large set S including all the network components, by performing binary measurements over subsets S_i, i.e., monitoring paths. As in graph-based group testing, the composition of the testing sets conforms to the structure of the network.

In this chapter, we consider the problem of maximizing the number of nodes whose states can be uniquely determined from binary measurements on a given number of monitoring paths. While the literature presents several works addressing the above optimization by considering different ways to deploy the monitors [65, 93] or routing packets [31, 93], researchers agree on the computational hardness of the general problem and propose heuristic approaches

providing lower bounds. Unlike previous work, we focus on deriving easily computable theoretical upper bounds. In formulating these bounds, we consider that monitoring paths are constrained not only by the network topology, but also by the routing scheme adopted in the network, and by additional requirements in case of passive monitoring, i.e. monitoring paths coinciding with service related paths. Knowledge of such theoretical bounds allows us to: (i) evaluate the performance of a given monitoring system and the room of improvement in a specific network setting, (ii) size the monitoring system (make decisions concerning how many monitors, how many paths, and related length) and (ii) extract guidelines for designing the most suitable topologies for failure localization. In addition, in this Chpater we give a formalization the Incremental Crossing Arrangement (ICA) procedure to generate monitoring schemes and underlying topologies that meet the bounds tightly, giving insights on which topology is the most suitable for failure localization.

The main contributions of this chapter are the following:

- We upper-bound the maximum number of identifiable nodes with a given number of monitoring paths, in the following scenarios: (1) paths between arbitrary nodes under arbitrary routing (Theorem 2.4.1); (2) paths between arbitrary nodes under consistent routing (Theorem 2.4.2).
- We provide the Incremental Crossing Arrangement (ICA) algorithm, which allows topology design meeting the proposed bounds.
- We give insights on the design of topologies and monitoring schemes to approximate the bounds, grounded upon the bound analysis.
- We demonstrate the tightness of the upper bounds by providing constructive approaches and comparisons with the results of known heuristics [65] on engineered as well as real network topologies.
- Through experiments, we compare the bounds in different scenarios to evaluate the impact of the routing scheme, the number of monitoring paths, and the maximum path length on the number of identifiable nodes.

2.2 Related work

The early works in Boolean Network Tomography focused on best-effort inference. For example, Duffield et al. [41, 46] and Kompella et al. [83] aimed at finding the minimum set of failures that can explain the observed measurements, and Nguyen et al. [106] aimed at finding the most likely failure set that explains the observations.

Later, the identifiability problem attracted attention. Ma et al. characterized in [92] the maximum number of simultaneous failures that can be uniquely localized, and then extended the results in [94] to characterize the maximum number of failures under which the states of specified nodes can be uniquely identified as well as the number of nodes whose states can be identified under a given number of failures. The work described in this chapter provides topology specific relationship of inclusions for the set of identifiable nodes. In contrast to [94], we provide fundamental bounds that are topology agnostic, i.e., only based on the number of monitoring paths and high level routing consistency properties.

The related optimization problems have also been studied. The problem of optimally placing monitors to detect failed nodes via round-trip probing was introduced and proven to be NP-hard by Bejerano et al. in [18]. The work by Cheraghchi et al. [31] aimed at determining bounds on the minimum number of monitoring paths to uniquely localize failures, where paths are defined by random walks in the network graph and the maximum number of simultaneous failures (also called identifiability index) is constrained. These studies are orthogonal to ours, as we aim at bounding the number of identifiable nodes,

Notation	Description
P	Set of m monitoring paths $P = \{p_1, \ldots, p_m\}$
p, $\hat{p} \in P$	Monitoring path as a set or a list of nodes, respectively
$b(v)$	Binary encoding of node v with respect to P
$b(v)\|_i$	i-th element of $b(v)$ (equal to 1 if and only if $v \in p_i$, to 0 otherwise)
$\chi(v)$	Crossing number of node v with respect to a set of paths P
P_F	Incident set of paths of a failure set F
$\mathcal{I}(p)$	Set of identifiable nodes traversed by path p
$M(\hat{p})$	Path matrix of path $\hat{p}$
$\mathcal{B}(k)$	$\{b \in \{0,1\}^m,\ s.t.\ \sum_{i=1}^{m} b\|_i = k\}$, $k = 1, \ldots, m$
B_V	Set of binary encodings of a set of nodes V

Table 2.1. Notation table.

within a given identifiability index, given the number of monitoring paths. Moreover we emphasize the impact of the routing scheme on the achievable failure identifiability.

For monitoring paths that start/end at monitors, Ma et al. [93] proposed polynomial time heuristics to deploy a minimum number of monitors to uniquely localize a given number of failures under various routing constraints. When monitoring is performed at the service layer, He et al. [65] proposed service placement algorithms to maximize the number of identifiable nodes by monitoring the paths connecting clients and servers.

Differently from Boolean Network Tomography, robust network tomography aims at inferring fine-grained performance metrics (e.g., delays) of non-failed links under failures. For robust network tomography, Tati et al. [135] proposed a path selection algorithm to maximize the expected rank of successful measurements subject to random link failures, and Ren et al. [119] proposed algorithms to determine which link metrics can be identified and where to place monitors to maximize the number of identifiable links, subject to a bounded number of link failures.

Robust network tomography has also been studied under settings not limited to failures [66, 89], to study the identifiability of additive link metrics under topology changes.

2.3 Problem formulation

We model the network as an undirected graph $G = (V, E)$, where V is a set of nodes, and E is the set of links. According to the needs of the discussion, a path p defined on G is represented as either a *set* of nodes p, or as an ordered *sequence* of nodes $\hat{p}$.

Each node may be in working or failed state. Without loss of generality, we assume that links do not fail and model network links through logical nodes so that a link failure corresponds to the failure of a logical node. The set of *all* failed nodes, denoted by $F \subseteq V$, defines the state of a network, and is called *failure set*. We assume that node states cannot be measured directly, but only indirectly via *monitoring paths*. The state of a path is working if and only if all traversed nodes (including endpoints) are in working state. Let P be a given set of m monitoring paths. We call the *incident set* of v the set of paths traversing v and denote it with $P_v \subseteq P$. We define with $\chi(v) \triangleq |P_v|$, the *crossing number* of node v, i.e., is the number of monitoring paths traversing v. We also denote the incident set of paths of a failure set F with $P_F \triangleq \cup_{v_i \in F} P_{v_i}$.

The characteristic vector[1] of P_v with respect to the set of paths $P = \{p_1, \ldots, p_m\}$, is

[1] A *characteristic vector* (or *indicator vector*) of a subset S of an ordered set of m elements

hereby denoted with $b(v)$ and called the *binary encoding* of v. It holds that $v \in p_i$ if and only if the i-th element of its binary encoding is equal to 1, i.e., $b(v)|_i = 1$. Note that multiple nodes may have the same binary encoding.

Observation 2.3.1. *The crossing number $\chi(v)$ of node v is equal to the number of ones in $b(v)$, i.e., $\chi(v) = \sum_{i=1}^{m} b(v)|_i$.*

2.3.1 Identifiability

The concept of identifiability refers to the capability of inferring the state of individual nodes from the state of the monitoring paths. Informally, we say that a node v is 1-identifiable with respect to a set of paths P, if its failure and the failure of any other node w cause the failure of different sets of monitoring paths in P, i.e. v and w have different incident sets. This concept can be extended to the case of concurrent failures of at most k nodes, where a node is k-identifiable in P if any two sets of failures F_1 and F_2 of size at most k, which differ at least in v (i.e., one contains v and the other does not), cause the failures of different paths, namely F_1 and F_2 have different incident sets, i.e. $P_{F_1} \neq P_{F_2}$. In the following, we reformulate the concept of k-identifiability (reported in Definition 1.1.3 and firstly introduced by He et al. in [65]) in terms of the concepts introduced in this chapter:

Definition 2.3.1. *Given a set of paths P and a node $v_i \in V$, v_i is called* k-identifiable *with respect to P when for any failure sets F_1 and F_2 such that $F_1 \cap \{v_i\} \neq F_2 \cap \{v_i\}$, and $|F_j| \leq k$ ($j \in \{1, 2\}$), it holds that[2]: $\bigvee_{v_s \in F_1} b(v_s) \neq \bigvee_{v_z \in F_2} b(v_z)$.*

The following Lemma derives from Definition 2.3.1, considering the special case of $k = 1$.

Lemma 2.3.1. *A node v_i is 1-identifiable with respect to P if and only if $b(v_i) \neq \mathbf{0}$, and $\forall v_j \neq v_i$, $b(v_j) \neq b(v_i)$, i.e., its binary encoding is not null and not identical with that of any other node.*

We clarify that, in agreement with Lemma 2.3.1, a node with null encoding cannot be 1-identifiable, because it is impossible to assess its status, working or failed, based only on the status of the monitoring paths.

2.3.2 Bounding identifiability

The set of monitoring paths P is usually the result of design choices related to topology, monitor placement, routing scheme, etc. Given a collection of candidate path sets $\mathcal{P}$ under all possible designs[3], the question is: how well can we monitor the network using path measurements in $\mathcal{P}$ and which design is the best? Using the notion of k-identifiability, we can measure the monitoring performance by the number of nodes that are k-identifiable with respect to $P \in \mathcal{P}$, denoted by $\phi_k(P)$, and formulate this question as an optimization: $\psi_k(\mathcal{P}) \triangleq \max_{P \in \mathcal{P}} \phi_k(P)$.

Although extensively studied [18,31,65,93], the optimal solution is hard to obtain due to the (exponentially) large size of $\mathcal{P}$, and heuristics are used to provide lower bounds. There is, however, a lack of general upper bounds. In this chapter we establish upper bounds on $\psi_k(\mathcal{P})$ in representative scenarios.

Note that, as discussed in [65], Definition 2.3.1 implies that if v is k-identifiable with respect to P for any $k \geq 1$, then v is also 1-identifiable with respect to P. It follows that

$P = \{p_1, p_2, \ldots, p_m\}$ is a binary vector with '1' only in the positions of the elements of P that are included in S.

[2] With "$\bigvee$" we refer to the element-wise logical `OR`.

[3] For example, $\mathcal{P}$ may be the class of path sets of given cardinality, or paths of a given length between given sources and each of multiple candidate destinations.

$\psi_1(\mathcal{P}) \geq \psi_k(\mathcal{P})$. Therefore, in the sequel, we look for upper bounds on $\psi_1(\mathcal{P})$, simply denoted by $\psi(\mathcal{P})$, where we will replace $\mathcal{P}$ by specific parameters in each network setting. Knowledge of these upper bounds is key to understanding the fundamental limits of Boolean Network Tomography, and gives insights on the optimal network design to facilitate network monitoring.

In the following, we shortly call the 1-identifiable nodes "identifiable".

2.4 General network monitoring

In this section we study the number of identifiable nodes of a general network of which we know the number of nodes and monitoring paths, without assuming any specific interconnecting topology. The following study provides bounds of general validity, although more refined bounds can be provided when more topological details are available. After the introduction of these bounds we give guidelines for the design of high identifiability topologies. We remark that in providing these guidelines we neglect other requirements (besides those related to routing and hop count for QoS), for example cost, technology constraints and more advanced performance requirements of such networks, which can be considered in a more technology and topology specific future study. In the following, we consider a collection $\mathcal{P}$ of candidate path sets of known cardinality $m \geq 1$, between any endpoints. We analyze $\psi(\mathcal{P})$ in two cases: (i) arbitrary routing and (ii) consistent routing.

2.4.1 Arbitrary routing

Identifiability bound

We hereby consider bounds on node identifiability which are valid regardless of the specific routing scheme being adopted. We refer to this scenario as *arbitrary routing*.

Proposition 2.4.1. *Given a network with n nodes, and m monitoring paths p_i, $i = 1, \ldots, m$, we denote with $\mathcal{I}(p_i)$ the set of identifiable nodes traversed by p_i and with $d_i \leq n$ the length of p_i in number of nodes. It holds that $|\mathcal{I}(p_i)| \leq \min\{d_i;\ 2^{m-1}\}$.*

Proof. By Lemma 2.3.1, in order for a node to be identifiable, its binary encoding must be unique. Since all the nodes traversed by path p_i have a '1' in the i-th position of their binary encoding, the number $|\mathcal{I}(p_i)|$ of identifiable nodes traversed by p_i is upper-bounded by the number of different sequences of m bits (binary encodings), where the i-th bit is a '1', which is 2^{m-1}, and of course, by the length d_i. □

Theorem 2.4.1 (Identifiability under arbitrary routing with known average path length)**.** *Given a network with n nodes, and m arbitrary routing paths with average length $\bar{d} \leq n$, the maximum number of identifiable nodes in the network satisfies:*

$$\psi^{AR}(m, n, \bar{d}) \leq \min\left\{\sum_{i=1}^{i_{max}} \binom{m}{i} + \left\lfloor \frac{N_{max} - \sum_{i=1}^{i_{max}} i \cdot \binom{m}{i}}{i_{max}+1} \right\rfloor ; n\right\},$$

$$\text{where } i_{max} = \max\{k \mid \textstyle\sum_{i=1}^{k} i \cdot \binom{m}{i} \leq N_{max}\},$$
$$\text{and}^{4}\ N_{max} = m \cdot \min\{\bar{d};\ 2^{m-1}\}.$$

Proof. The number $|\mathcal{I}(p_i)|$ of identifiable nodes traversed by a path p_i of length d_i, $i \in \{1, \ldots, m\}$, is bounded as described by Proposition 2.4.1. Consequently, the number of identifiable nodes is also bounded from above as follows: $|\cup_{i=1}^{m} \mathcal{I}(p_i)| \leq \sum_{i=1}^{m} |\mathcal{I}(p_i)| \leq \sum_{i=1}^{m} \min\{d_i;\ 2^{m-1}\} \leq m \cdot \min\{\bar{d};\ 2^{m-1}\} = N_{\texttt{max}}$.

[4] By definition $N_{\texttt{max}}$ is an integer number.

Since we used the union bound to calculate $N_{\max}$, this value considers some encodings multiple times when the related node belongs to more than one path. This happens, according to Observation 2.3.1, $\chi(v)$ times for each node v.

It follows that the number of distinct encodings is maximized when we minimize the number of encoding replicas and therefore the crossing number of the related nodes. This is achieved, within the limits of the path length, when we have $\binom{m}{1}$ nodes with crossing number equal to 1 (counted only once in $N_{\max}$), $\binom{m}{2}$ nodes with crossing number equal to 2 (counted twice in $N_{\max}$), and so forth, until the total number of encodings (counting the replicas) is $N_{\max}$.

More formally, let $i_{\max} = \max\{k \mid \sum_{i=1}^{k} i \cdot \binom{m}{i} \leq N_{\max}\}$. For each $i \leq i_{\max}$, we have $\binom{m}{i}$ nodes with crossing number equal to i, i.e., traversed by i paths. Considering that the remaining $N_{\max} - \sum_{i=1}^{i_{\max}} i \cdot \binom{m}{i}$ encodings will have at least $(i_{\max}+1)$ digits equal to '1' and thus are counted at least $(i_{\max}+1)$ times in $N_{\max}$, the number of distinct encodings out of the $N_{\max}$ encodings is upper-bounded by:

$\psi^{\mathrm{AR}}(m,n,\bar{d}) \leq \sum_{i=1}^{i_{\max}} \binom{m}{i} + \left\lfloor \frac{N_{\max} - \sum_{i=1}^{i_{\max}} i \cdot \binom{m}{i}}{i_{\max}+1} \right\rfloor$.

Considering also that the number of identifiable nodes cannot exceed n, we have the final bound of the theorem.

□

We underline that Theorem 2.4.1 provides a topology-agnostic bound, i.e., a theoretical limit which is valid for any topology and only considers the number of nodes, the number of monitoring paths, and the average path length[5]. When the monitoring paths are known, the bound of Theorem 2.4.1 is useful to quantitatively evaluate the margin for improvement of the identifiability using another set of monitoring paths with the same cardinality and average length.

We also observe that knowledge of the path lengths is not a requirement for the use of the proposed bound. More specifically when path lengths are unknown, we have $N_{\max} = m \cdot 2^{m-1}$, and $i_{\max} = m$, and Theorem 2.4.1 reduces to the following corollary for unbounded path length.

Corollary 2.4.1 (Identifiability under arbitrary routing and unbounded path length)**.** *Given a network with n nodes and m monitoring paths, the maximum number of identifiable nodes satisfies:*

$$\psi^{AR}(m,n) \leq \min\{n; 2^m - 1\}.$$

Despite the simplicity of the bound of Corollary 2.4.1, we underline that the bound value is achieved tightly by specifically designed topologies for which routing and path lengths are not constrained. Even in the case in which the average length of monitoring paths is not known a priori, the routing scheme or QoS requirements limiting the hop count of the monitoring paths may imply an upper bound on the path length, $d_{\max}$. For example, in the case of shortest path routing, the value of $d_{\max}$ is upper bounded by the diameter of the network. The value of $d_{\max}$ can be used to derive a variation of the bound, $\psi^{\mathrm{AR}}(m,n,d_{\max})$, on the number of identifiable nodes, since $\bar{d} \leq \max_i\{d_i\} \leq d_{\max}$.

Corollary 2.4.2 (Identifiability under arbitrary routing and bounded maximum path length)**.** *Given a network with n nodes, and m arbitrary routing paths with maximum length d_{max}, the maximum number of identifiable nodes in the network, $\psi^{AR}(m,n,d_{max})$, is upper-bounded as in Theorem 2.4.1, except that N_{max} is now defined as: $N_{max} = m \cdot \min\{d_{max};\ 2^{m-1}\}$.*

[5] As the constraints imposed by the topology of the network and path routing are not taken into account in this theorem, its validity holds also for any group testing problem where m groups of known average size are used to inspect the state of n elements.

2.4.2 Design via Incremental Crossing Arrangement (ICA)

The proof of Theorem 2.4.1 suggests a technique to build a network topology $G = (V, E)$ and related monitoring paths P with maximum identifiability, where $|P| = m$. We call this technique *Incremental Crossing Arrangement* (ICA).

ICA, the idea. The technique works by generating node encodings in increasing order of crossing number with respect to the m monitoring paths in use, until the number of generated encodings reaches the bound defined in Theorem 2.4.1. Monitoring paths must be designed so as to traverse nodes according to the generated encodings: path p_i traverses any node v for which $b(v)|_i = 1$, $\forall i \in \{1, \dots, m\}$. The network topology is then constructed by considering a node for each of the generated Boolean encodings, and adding links between any pair of nodes appearing sequentially in any path.

ICA in details. Algorithm 1 formalizes the incremental crossing arrangement design, used to determine the binary encodings of the identifiable nodes.

As we consider m paths, the node encodings will be sequences of m bits in $\mathcal{B} \triangleq \{0,1\}^m$. We also denote with $\mathcal{B}|_i \subset \mathcal{B}$ the set of m-digits binary encodings having a 1 in the i-th position, i.e., $\mathcal{B}|_i = \{b \in \mathcal{B}\ s.t.\ b|_i = 1\}$. The nodes corresponding to encodings of $\mathcal{B}|_i$ will be monitored (at least) by path p_i. Moreover, we denote with $\mathcal{B}(k) \subset \mathcal{B}$ the set of all binary encodings having exactly k digits equal to 1, therefore $\mathcal{B}(k) \triangleq \{b \in \mathcal{B}\ s.t.\ \sum_{i=1}^{m} b|_i = k\}$. The nodes corresponding to encodings in $\mathcal{B}(k)$ have crossing number equal to k.

Finally, given a generic set of binary encodings $B \subseteq \mathcal{B}$, we denote with $\ell_i(B)$ the number of encodings of B having a one in the i-th position: $\ell_i(B) \triangleq |B \cap \mathcal{B}|_i|$. The value of $\ell_i(B)$ represents the length of a path p_i traversing all the nodes in $B \cap \mathcal{B}|_i$, exactly once.

Without loss of generality, we consider paths of balanced length, i.e. we set the length d_i of path p_i to a value $d_i \in \{\lfloor \bar{d} \rfloor, \lfloor \bar{d} \rfloor + 1\}$ (**lines 2 - 4**).

The incremental crossing arrangement approach incrementally generates the solution set B_V by including all the encodings of $\mathcal{B}(i)$, $i = 1, \dots, i_{\max} + 1$ corresponding to nodes with crossing number lower than or equal to $i_{\max}$. It then considers some encodings with $(i_{\max} + 1)$ digits equal to one. For this purpose it generates a family $\mathcal{F}$ of subsets in $\mathcal{B}(i_{\max} + 1)$, i.e., $\mathcal{F} \subseteq 2^{\mathcal{B}(i_{\max}+1)}$ (**line 7**) whose elements B are such that $\ell_k(B \cup B_V) \leq d_k$. The algorithm then looks for a maximal cardinality set B^* in the family $\mathcal{F}$ and adds it to the solution B_V, s.t. $B_V = \cup_{k=1}^{i_{\max}+1} \mathcal{B}(k) \cup B^*$. Notice that the maximality of the cardinality of B^* implies that no encoding with $(i_{\max} + 1)$ digits equal to one can be added to the set B_V without violating the path length constraint $\ell_k(B_V) \leq d_k$ for some path $k = 1, \dots, m$, or without removing at least one encoding already in B_V.

The procedure described so far is sufficient to produce a network topology with m paths of average path length $\bar{d}$ meeting the bound of Theorem 2.4.1. In the produced topology, there can be values of $k \in \{1, \dots, m\}$ for which $\ell_k(B_V) < d_k$ and, more precisely, given the balanced path length, $\ell_k(B_V) = d_k - 1$, corresponding to paths longer than strictly necessary to identify n nodes, i.e., overlength paths. Overlength paths cannot traverse nodes with the same encoding without compromising the achievement of maximum identifiability. Therefore, in order for the average path distance to meet the value $\bar{d}$, we proceed as follows, with a procedure that we call *Path Completion*. First, we observe that under ICA, the bound on the minimum number of monitoring paths can sometimes be met tightly even when the average path length is slightly lower than the given $\bar{d}$. This condition is verified when the ratio inside the floor operator of the bound expression of Theorem 3.3.1 is not integer. Nevertheless, the same bound can still be met tightly with the exact average length provided as input, by operating as follows: let $S \subset \{1, \dots, m\}$ be the set of overlength path indexes, namely $S \triangleq \{k,\ s.t.\ \ell_k(B_V) = d_k - 1\}$. It holds $|S| = \left[m \cdot D - \sum_{i=1}^{i_{\max}} i \cdot \binom{m}{i} \mod (i_{\max} + 1) \right]$, hence the number of overlength paths is lower than or equal to $i_{\max}$.

We choose an encoding $b' \in B_V \cap \mathcal{B}(i_{\max} + 1 - |S|)$ such that $b'|_k = 0, \forall k \in S$, and such

that $\left(\bigvee_{k\in S} \mathbf{e_k} \vee b'\right) \notin B_V$, where $\mathbf{e}_k$ is an m-dimensional identity vector with all zeroes but a one in the k-th position[6]. Then we remove b' from the solution set B_V and replace it with $b'' \triangleq \bigvee_{k\in S} \mathbf{e_k} \vee b'$, i.e., with a new encoding b'' such that $b''|_k = 1, \forall k \in S$, and $b''|_k = b'|_k$ otherwise.

Algorithm 1 Incremental Crossing Arrangement

Input: m and $\bar{d}$.

Output: A set of encodings B_V which can be mapped onto a topology graph $G = (V, E)$, with m paths of average length $\bar{d}$, such that $\psi^{\mathrm{AR}}(m, \bar{d}$ corresponding nodes are identifiable.

1: Compute $N_{\mathtt{max}}$ and $i_{\mathtt{max}}$ according to Theorem 2.4.1 and $\psi^{*\mathtt{AR}}(m, \bar{d}) \triangleq \sum_{i=1}^{i_{\mathtt{max}}} \binom{m}{i} + \left\lfloor \frac{N_{\mathtt{max}} - \sum_{i=1}^{i_{\mathtt{max}}} i \cdot \binom{m}{i}}{i_{\mathtt{max}}+1} \right\rfloor$

2: Compute $m_1 \triangleq m \cdot (\bar{d} - \lfloor \bar{d} \rfloor)$;

3: **for** $i = 1, \ldots, m_1$ **do** set $d_i = \lfloor \bar{d} \rfloor + 1$

4: **for** $i = m_1 + 1, \ldots, m$ **do** set $d_i = \lfloor \bar{d} \rfloor$

5: $B_V = \emptyset$

6: **for** $i = 1, \ldots, i_{\mathtt{max}}$ **do** $B_V = B_V \cup \mathcal{B}(i)$

7: Calculate the family $\mathcal{F}$ defined as
$\mathcal{F} \triangleq \{B : \ B \subseteq \mathcal{B}(i_{\mathtt{max}} + 1) \wedge \ell_k(B \cup B_V) \in [d_k - 1, d_k], \ \forall k = 1, \ldots, m\}$

8: Choose $B^* = \arg\max_{B\in\mathcal{F}} |B|$

9: $B_V = B_V \cup B^*$

10: **if** $\exists k \in \{1, \ldots, m\}$ s.t. $\ell_k(B_V) = d_k - 1$ **then**

11: Perform *path completion* and update B_V

12: Return B_V

ICA: Example A (where path completion is not necessary). Figure 2.1 shows an example of a topology generated by means of incremental crossing arrangement. We are given $m = 4$ nodes and $\bar{d} = 4.75$. Applying Algorithm 1, we have $N_{\mathtt{max}} = m \cdot \bar{d} = 19$ and $i_{\mathtt{max}} = 2$. We set $d_i = 5, \forall i = 1, 2, 3$ and $d_4 = 4$ (**lines 2 - 4**). According to ICA, we first generate all the encodings of $\mathcal{B}(1)$ and in $\mathcal{B}(2)$ and set $B_V = \{1000, 0100, 0010, 0001, 1100, 1010, 1001, 0110, 0101, 0011\}$ (**line 6**). Then we start generating some encodings in $\mathcal{B}(3) = \mathcal{B}(i_{\mathtt{max}} + 1)$ until no other encoding can be added without violating the path length constraint (**lines 7 - 9**), obtaining $B_V = \{1000, 0100, 0010, 0001, 1100, 1010, 1001, 0110, 0101, 0011, 1110\}$, where each encoding corresponds to a node of the graph G. The number of generated encodings is 11, that is the value of ψ^{AR} for the given input. Then we define the corresponding monitoring paths, by letting path p_i traverse all the nodes whose encoding has a 1 in the i-th position, in arbitrary order, $\forall i \in \{1, \ldots, m\}$. Finally, we design the underlying topology by connecting each pair of nodes appearing in a sequence in any of the paths, as shown in Figure 2.1.

ICA: Example B (with path completion). Figure 2.2 shows another example of a topology generated by means of incremental crossing arrangement. We are given $m = 4$ and $\bar{d} = 5$. To meet the requirement on average length, we set $d_i = 5 \, \forall i = 1, \ldots, 4$ (**lines 2 - 4**). According to ICA (**line 6**), we first generate all the encodings of $\mathcal{B}(1)$ and $\mathcal{B}(2)$ and set $B_V = \{1000, 0100, 0010, 0001, \underline{1100}, 1010, 1001, 0101, 0011\}$. Then we choose one of the possible B^* ((**lines 7 - 9**)), for instance $B^* = 1110$, obtaining $B_V = \{1000, 0100, 0010, 0001, \underline{1100},$

[6] We can always find an encoding b' with the described properties because B_V contains all the encodings of $\mathcal{B}(i_{\mathtt{max}} + 1 - |S|)$ and not all the encodings b of the set $\mathcal{B}(i_{\mathtt{max}} + 1)$ for which $b|_i = 1, \forall i \in S$.

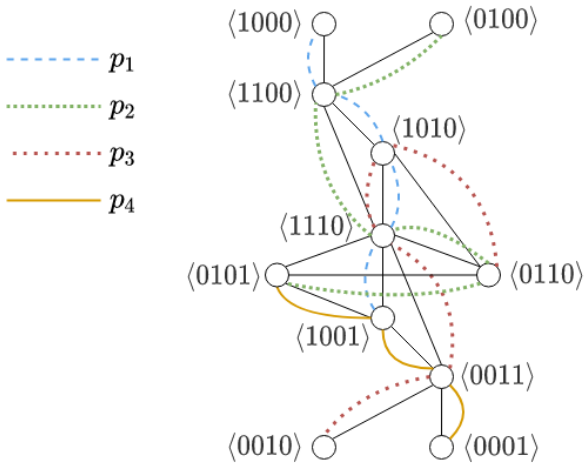


Figure 2.1. ICA execution on Example A. Solid lines represent graph edges.

1010, 1001, 0101, 0011, 1110}

Finally, we observe that $\ell_4(B_V) = 4 < d_4$. We then perform the path completion procedure (**line 11**) and choose one of the encodings b' in $B_V \cap \mathcal{B}(i_{\max} + 1 - |S|) = \mathcal{B}(2)$ for which $b'|_4 = 0$ and $b' \vee \mathbf{e_4} \notin B_V$. One encoding that satisfies this condition is $b' = 1100$. We replace b' with $b'' = 1101$ and obtain the set of encodings {1000, 0100, 0010, 0001, 1101, 1010, 1001, 0101, 0110, 0011, 1110}, each corresponding to a node of the graph G. Again, the number of generated encodings corresponds to ψ^{AR} in Theorem 2.4.1 for the given input. Then we define the corresponding monitoring paths, by letting path p_i traverse all the nodes whose encoding has a 1 in the i-th position, in arbitrary order, $\forall i \in \{1, \ldots, m\}$. Finally, we design the underlying topology by connecting each pair of nodes appearing in a sequence in any of the paths, obtaining the topology of Figure 2.2.

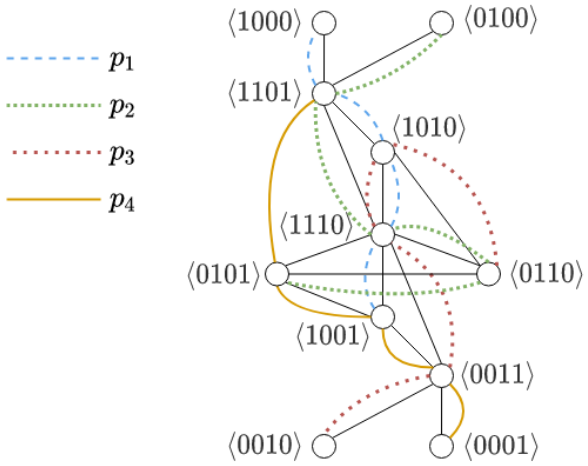


Figure 2.2. ICA execution on Example B. Solid lines represent graph edges.

It is worth observing the following.

Observation 2.4.1. *ICA produces a network topology and related monitoring paths such that all nodes have a crossing number lower than or equal to* $(i_{max}+1)$.

When knowledge of $\bar{d}$ is not available, and instead we limit the paths length to be at most $d_{\max}$, Algorithm 1 can be simplified allowing the following changes: the number of paths m is computed as in Theorem 3.3.1 (**line 1**). To all nodes, we initially assign $d_i = d_{\max}$ (**lines 2-4**). The algorithm continues as it is until the condition that some paths satisfy $\ell_k(B_V) = d_{\max} - 1$ is met (**line 11**). When $d_{\max}$ is used, path completion is not required, and for such paths we simply set $d_k = \ell_k(B_V)$. The returned set of encodings B_V can be mapped onto a topology graph where all n nodes are identifiable by using m paths with maximum path length $d_{\max}$.

Tightness of the bound on identifiability under arbitrary routing

In this section we show that the bound given by Theorem 2.4.1 can be achieved tightly for a specific family of topologies, namely those constructed via ICA.

Proposition 2.4.2 (Tightness of Theorem 2.4.1)**.** *For any number m of monitoring paths with average length $\bar{d} \in (0, 2^{m-1}]$, there exists a set P of m paths, such that the number of identifiable nodes equals the bound given in Theorem 2.4.1:*

$$\psi^{*AR}(m,\bar{d}) = \sum_{i=1}^{i_{max}} \binom{m}{i} + \left\lfloor \frac{N_{max} - \sum_{i=1}^{i_{max}} i \cdot \binom{m}{i}}{i_{max}+1} \right\rfloor .$$

Proof. We give an existence proof by showing that we can build the paths of the proposition by means of the ICA technique. We need to show that the number of identifiable nodes obtained through ICA is equal to the one provided by the bound of Theorem 2.4.1. We denote with $\mathcal{B}(i)$ the set of m-digit encodings with i digits equal to one. ICA initially generates all the encodings of $\mathcal{B}(i)$, for $i = 1, \ldots, i_{\max}$. As a consequence, notice that each path will traverse at least $d(i_{\max}) \triangleq \sum_{i=0}^{i_{\max}-1} \binom{m-1}{i}$ identifiable nodes. In fact, the encodings of the nodes of $\mathcal{I}(p_i)$ (identifiable nodes traversed by path p_i), must have a '1' in the i-th position. Therefore the number of distinct encodings corresponding to nodes of $\mathcal{I}(p_i)$ is at least equal to the number of binary sequences of $(m-1)$ elements, with up to $(i_{\max}-1)$ ones, which is $d(i_{\max})$.

Under ICA, each path also traverses other nodes with crossing number equal to $(i_{\max}+1)$. Each of these nodes will appear in exactly $(i_{\max}+1)$ paths. The number of such nodes is therefore given by $\left\lfloor \frac{\sum_{k=1}^{m}(d_k - d(i_{\max}))}{(i_{\max}+1)} \right\rfloor$.

Hence, ICA generates the set of node encodings B_V including the following:

- $\binom{m}{i}$ encodings corresponding to nodes with crossing number equal to i, for $i = 1, \ldots i_{\max}$, and
- $\left\lfloor \frac{\sum_{k=1}^{m}(d_k - d(i_{\max}))}{(i_{\max}+1)} \right\rfloor$ encodings corresponding to nodes with crossing number equal to $(i_{\max}+1)$.

ICA constructs the set B_V in a way that each encoding corresponds to a unique node, and the nodes are traversed by paths of average length $\bar{d}$, guaranteeing identifiability of all the nodes corresponding to the generated encodings.

In order to show that the number of identifiable nodes is equal to the one provided by the bound of Theorem 2.4.1, we need to prove that $\left\lfloor \frac{\sum_{k=1}^{m}(d_k - d(i_{\max}))}{(i_{\max}+1)} \right\rfloor = \left\lfloor \frac{N_{\max} - \sum_{i=1}^{i_{\max}} i \cdot \binom{m}{i}}{(i_{\max}+1)} \right\rfloor$, which holds because $\sum_{k=1}^{m} d_k = m \cdot \bar{d} = N_{\max}$, and $m \cdot d(i_{\max}) = m \cdot \sum_{i=0}^{i_{\max}-1} \binom{m-1}{i} = \sum_{i=1}^{i_{\max}} i \cdot \binom{m}{i}$, which can easily be proven by expanding the binomial coefficients.

Notice that the proposition requires $\bar{d} \leq 2^{m-1}$ as a higher value would require the generation of at least a path to traverse different nodes with the same encoding, losing identifiability with respect to the bound value. □

2.4.3 Consistent routing

Identifiability bound

As we discussed in Section 2.4.1, when monitoring paths can be routed arbitrarily, the number of identifiable nodes may be exponential in the number of monitoring paths. In contrast, the adoption of specific routing schemes may affect the identifiability of nodes.

In the following we study the impact of consistent routing schemes.

Definition 2.4.1. *A set of paths P is* consistent *if $\forall p, p' \in P$ and any two nodes u and v traversed by both paths (if any), p and p' follow the same sub-path between u and v.*

Note that routing consistency implies that paths are cycle-free. Figure 2.2 is an example of non-consistent routing. For example paths p_1 and p_3 choose different routes to go from node 1110 to node 1010, across nodes 1001 and 0011, respectively. An example of consistent routing of monitoring paths is instead given in Figure 2.3.

We observe that routing consistency is satisfied by many practical routing protocols, including but not limited to shortest path routing (where ties are broken with a unique deterministic rule), and by many emerging routing techniques. For instance, routing consistency may be the result of QoS policies, aiming at balancing traffic along several lines, such as in the case of fat-tree based data center topologies, largely is discussed in [16].

We define the *path matrix* of $\hat{p}_i$ as a binary matrix $M(\hat{p}_i)$, in which each row is the binary encoding of a node on the path, and rows are sorted according to the sequence $\hat{p}_i$. Notice that by definition $M(\hat{p}_i)|_{*,i}$ has only ones, i.e., $M(\hat{p}_i)|_{r,i} = 1,\ \forall r$.

Lemma 2.4.1. *Under the assumption of consistent routing, if any two different rows of the matrix $M(\hat{p}_i)$ are equal, then the corresponding nodes are not 1-identifiable.*

Proof. Under consistent routing, the path $\hat{p}_i$ cannot contain any cycle, so every row of $M(\hat{p}_i)$ corresponds to a different node. If two different nodes have the same binary encoding, by Lemma 2.3.1, the two nodes are not identifiable. □

Definition 2.4.2. *A column $M(\hat{p})|_{*,k}$ $(k = 1, \ldots, m)$ of a path matrix $M(\hat{p})$ has* consecutive ones *if all the "1"s appear in consecutive rows, i.e., for any two rows i and j $(i < j)$, if $M(\hat{p})|_{i,k} = M|_{j,k} = 1$, then $M|_{h,k} = 1$ for all $i \leq h \leq j$.*

Lemma 2.4.2. *Under the assumption of consistent routing, all the columns in all the path matrices have consecutive ones.*

Proof. The assertion is true for $M(\hat{p}_i)|_{*,i}$ since it contains only ones. Let us consider column $M(\hat{p}_i)|_{*,j}$, with $j \neq i$. Assume by contradiction that there are two rows $k_1 < k_2$ s.t. $M(\hat{p}_i)|_{k_1,j} = M(\hat{p}_i)|_{k_2,j} = 1$ but there is a row h with $k_1 < h < k_2$ for which $M(\hat{p}_i)|_{h,i} = 0$. Let v_1, v_2, and v_h be the nodes with encodings $M(\hat{p}_i)|_{k_1,*}$, $M(\hat{p}_i)|_{k_2,*}$, and $M(\hat{p}_i)|_{h,*}$, respectively. Then the paths $\hat{p}_i$ and $\hat{p}_j$ traverse both nodes v_1 and v_2 following different paths, of which only $\hat{p}_i$ traverses node v_h, in contradiction with consistent routing. □

Lemma 2.4.3. *Given $m > 1$ consistent routing paths, each path p_i having length d_i, the maximum number of different encodings in the rows of $M(\hat{p}_i)$ is upper-bounded by $\min\{d_i; 2 \cdot (m-1)\}$.*

Proof. While the number of different encodings appearing in the rows of $M(\hat{p}_i)$ is trivially bounded by d_i, it can even be lower. By considering each column of $M(\hat{p}_i)$ separately we observe the following. First, column $M(\hat{p}_i)|_{*,i}$ contains only ones. Second, for any column $M(\hat{p}_i)|_{*,j}$ with $j \neq i$, it holds, by Lemma 2.4.2, that it has consecutive ones.

We say that column k has a *flip* in row r if $M(\hat{p}_i)|_{r-1,k} \neq M(\hat{p}_i)|_{r,k}$. Due to Lemma 2.4.2 any column of $M(\hat{p}_i)$ can have up to two flips or it would interrupt a sequence of ones, violating Lemma 2.4.2. In fact, if the column starts with a '0' in the first row, it can flip from '0' to '1' in row r_1 and then back in row r_2, with $r_2 > r_1$, but if it flips from '1' to '0' it cannot flip back in a successive column. If instead the column starts with a '1' in the first row, it can only flip once. In order to have a change in the encoding contained in any two successive rows $r-1$ and r of the matrix $M(\hat{p}_i)$, i.e., $M(\hat{p}_i)|_{r-1,*} \neq M(\hat{p}_i)|_{r,*}$, there must be at least a column that flips in r. The number of columns that can flip is $m-1$ and each of them can flip at most twice. The number of different rows of $M(\hat{p}_i)$ is therefore upper-bounded by the smallest between the path length d_i and $2 \cdot (m-1)$. □

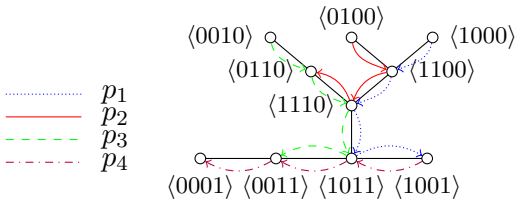


Figure 2.3. Consistent routing paths identifying all nodes of the network.

In the example of Figure 2.3 routing is consistent and each column flips at most twice, so the number of different rows is lower than, or equal to $2 \cdot (m-1) = 6$. For instance, considering $M(\hat{p}_3)$, we observe only $4 < 6$ flips:

$$M(\hat{p}_3) = \begin{array}{c} \text{flips} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} b_1 & b_2 & b_3 & b_4 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Theorem 2.4.2 (Identifiability with consistent routing)**.** *Given n nodes, and a set P of $m > 1$ consistent routing paths, with average path length $\bar{d}$, the maximum number of identifiable nodes ψ^{CR}, for any G and any location of the path endpoints, is upper-bounded as in Theorem 2.4.1,*

$$\psi^{CR}(m, n, \bar{d}) \leq \min\left\{ \sum_{i=1}^{i_{max}} \binom{m}{i} + \left\lfloor \frac{N_{max} - \sum_{i=1}^{i_{max}} i \cdot \binom{m}{i}}{i_{max}+1} \right\rfloor ; n \right\},$$

where $i_{max} = \max\{k \mid \sum_{i=1}^{k} i \cdot \binom{m}{i} \leq N_{max}\}$, except that N_{max} is now defined as: $N_{max} = m \cdot \min\{\bar{d}\,; 2 \cdot (m-1)\}$.

Proof. The proof follows the same arguments used to prove Theorem 2.4.1, concerning the minimization of the number of ones in the node encodings to reduce the number of repeated encodings among the different path matrices. Nevertheless, due to routing consistency, the value of $N_{\max}$ is now the sum of the bound value of Lemma 2.4.3, for each path. □

We underline that the setting of $N_{\max}$ is the fundamental aspect which makes Theorem 2.4.2 and Theorem 2.4.1 different from each other. While in Theorem 2.4.1 $N_{\max}$ can be as high as $m \cdot 2^{(m-1)}$, i.e., exponential in the number of paths m, in Theorem 2.4.2 $N_{\max}$ can never be higher than $2m \cdot (m-1)$, i.e. quadratic in the number of paths.

When the individual path length is not known, nor is the average path length, but the path length is upper-bounded by $d_{\max}$, we have the following Corollary of Theorem 2.4.2.

Corollary 2.4.3 (Identifiability under consistent routing, and bounded maximum path length)**.** *Given a network and a set P of $m > 1$ consistent routing paths with maximum length d_{max}, the maximum number of identifiable nodes in the network is upper-bounded as in Theorem 2.4.1, except that N_{max} is now defined as: $N_{max} = m \cdot \min\{d_{max};\ 2 \cdot (m-1)\}$.*

We remark that prior knowledge of the monitor path length is not a necessary requisite for the usage of the bounds of Theorem 2.4.2 and Corollary 2.4.3. When unknown, the value of $d_{\max}$ can be replaced with any upper bound, for instance with the number of nodes in the network or with the diameter of the network, in the case of shortest path consistent routing.

Tightness of the bound and design insights

It must be noted that in generating node encodings ICA does not ensure the existence of a consistent routing solution, therefore, differently from the case of arbitrary routing, it is not always applicable to produce tight topologies. Nevertheless, some topologies generated by ICA, for certain values of m, n and $\bar{d}$, achieve the bound of Theorem 2.4.2 also in the case of consistent routing.

For example, we use ICA to generate the topology shown in Figure 2.4, that we name *half-grid*.

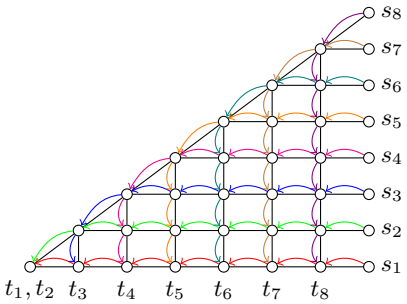


Figure 2.4. An example of half-grid graph.

We consider $m = 8$ paths. The figure highlights the source s_i and destination t_i of any path p_i, $i = 1, \ldots, m$, where $d_i = 8$ for all paths, hence $\bar{d} = m = 8$. Observe that the half-grid satisfies the condition of routing consistency, and all the $n = \binom{8}{1} + \binom{8}{2} = 36$ nodes are identifiable. In agreement with Observation 2.4.1, the maximum crossing number in this topology is equal to $i_{\max} = 2$. Such a topology can be constructed for any m paths, $n = m \cdot (m+1)/2$ nodes and $d_i = m$. In the resulting half-grid, routing is consistent and all the nodes are identifiable. Moreover, we observe that some *modified half-grid* topologies, using path length in the range $[m, m+3]$, with average $\bar{d} \leq \frac{m^2+3m-6}{m}$, may still meet the bound when we add some new nodes (up to $(m-2)$) with crossing number equal to 3 along the diagonal of the grid[7] . More specifically, in Figure 2.5, we modified the half-grid of

[7]The requirement $\bar{d} \leq \frac{m^2+3m-6}{m}$ comes from a simple counting argument on the modified half-grid topology with the maximum number of additional nodes along the diagonal. The top and bottom path of the half grid would have length $m+1$, the second and second to last paths would have

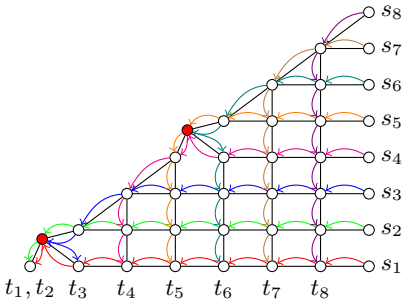


Figure 2.5. An example of half-grid graph with two additional nodes.

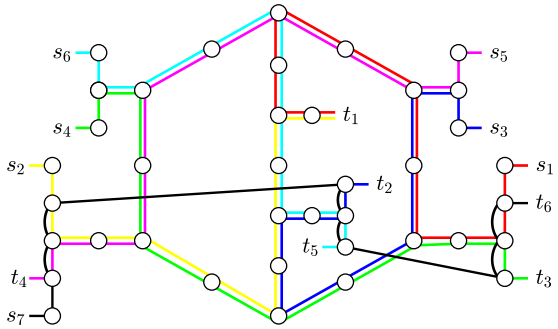


Figure 2.6. A topology that meets the bound of Theorem 2.4.2 with $m = 7$ and $\bar{d} = \frac{82}{7}$, and $d_{\texttt{max}} = 12$.

Figure 2.4, by adding two new nodes (the two red nodes of the figure) using $m = 8$ paths, numbered as above, and $d_1 = \ldots = d_6 = 9$, $d_7 = d_8 = 8$, meaning that $\bar{d} = \frac{70}{8} = 8.75$. Again, it holds that routing is consistent and that the bound of Theorem 2.4.2 is achieved tightly, as $\psi^{\text{CR}} = \binom{8}{1} + \binom{8}{2} + \lfloor \frac{6}{3} \rfloor = 38$.

It remains open to find the general family of topologies that can achieve the bound in Theorem 2.4.2. However, half-grid based topologies are not the only ones that can achieve the bound. In fact, we observe that there are other topologies meeting the bound tightly, for settings where the average length of the paths is $\bar{d} > \frac{m^2+3m-6}{m}$ (a requirement for the tightness of modified half-grid topologies). An example is given in Figure 2.6 where ICA is used for $m = 7$ consistent routing paths, each with length 12, except for one that has length 10, thus $\bar{d} = \frac{82}{7}$ and $d_{\texttt{max}} = 12$. All the 39 nodes in the figure are identifiable, and so the bound of Theorem 2.4.2 is achieved tightly and with nodes whose crossing number is always lower than or equal to 3. We observe that under this setting of number of paths and path length, i.e., where $\bar{d} \geq m + 3$, a (modified) half-grid could not meet the bound tightly, unless violating routing consistency.

length $m + 2$ and all the other paths would have length $m + 3$ thus enabling the identification of the $m - 2$ additional nodes along the diagonal.

2.5 Performance evaluation

To evaluate the tightness of the proposed upper bounds, we compare them with lower bounds obtained by known heuristics on real network topologies.

2.5.1 Bound Analysis

We analyze the tightness of the upper bound in Theorems 2.4.1 and 2.4.2. In Figure 2.7 the upper bounds (UB) computed as in Theorems 2.4.1 and 2.4.2 are shown together with a lower bound (LB) obtained by considering a half-grid, and placing monitoring endpoints as in Section 2.4.3. We remark that in the computation of the bounds for these plots, we consider $n = 78$ nodes, paths of equal length, such that $\bar{d} = d_{\max} = 12$, while we vary the number of paths m.

Notice that the upper bounds given by Theorems 2.4.1 and 2.4.2 for $d_{\max} = 12$ are the same for $m = 2,\ 3$, that is when $\min\{d_i; 2^{m-1}\} = \min\{d_i; 2 \cdot (m-1)\}$, and for $m \geq 7$, that is the threshold above which it holds that $\min\{d_i; 2 \cdot (m-1)\} = d_i = 12$. This result highlights how consistent routing reduces the maximum number of identifiable nodes.

The figure also shows the identifiability of the modified half-grid topology, (see Figures 2.4 and 2.5). Notice that, as we pointed out in Section 2.4.3, the bound on the number of identifiable nodes under the assumption of consistent routing (Theorem 2.4.2) is tight on the modified half grid topologies when m satisfies $\frac{m^2+3m-6}{m} \geq d_i$ (that in this example is when $m \geq 10$). The green triangle in the figure represents the number of identifiable nodes for the topology shown in Figure 2.6.

In Figure 2.8, we also consider a scenario with $n = 78$ nodes and we show how the bound of Theorem 2.4.2 varies with the number of monitoring paths m and the maximum path length $d_{\max}$. For small values of $d_{\max}$ the bound has an almost linear growth with m. For larger values of $d_{\max}$ the bound shows two regions: an initial super-linear growth for small values of m, and a linear growth for large values of m. The figure also shows that while the number of paths m has a major impact on the number of identifiable nodes, the length of the monitoring paths has a significant impact only when $d_{\max}$ is small, and diminishing impact otherwise. It must be noted that the bound of Theorem 2.4.2 increases with the value of $d_{\max}$, as long as we have $d_{\max} \leq 2 \cdot (m-1)$ because the optimal number of identifiable nodes is also increasing.

2.5.2 Tightness Evaluation on Real Topologies

For the next experiment we consider two real physical layer networks, US Signal and Uninett, both available in the Topology Zoo archive [82]. US Signal is a fiber optical

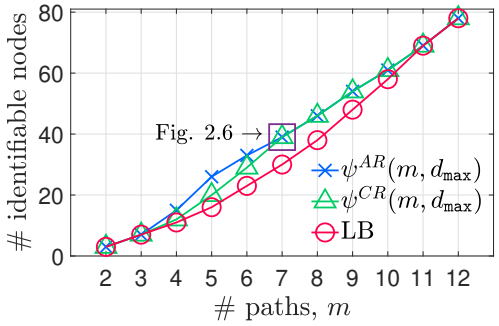


Figure 2.7. Bounds of Th. 2.4.1 and Th. 2.4.2, and LB for $n = 78$, $d_{\max} = 12$.

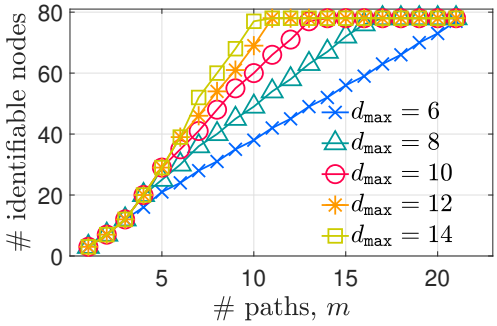


Figure 2.8. Bound of Th. 2.4.2, for $n = 78$, and different values of $d_{\max}$.

network in the USA ($n = 63$ nodes and 133 edges), whereas Uninett is an existing Internet topology located in Norway ($n = 69$ nodes and 98 edges). We underline that the choice of sampling the network at any layer of the protocol stack influences the capability of BNT to identify failures. In particular, BNT techniques identify node failures at the same level of representativeness of the topology sampling. Namely, if a node of the topology hides details of the underlying network layer topology, the identification of a failure on that node may reflect one or more failures in the portion of the lower layer topology. Notice also that communication network topologies may be only partially available when obtained by means of measurements such as *traceroute*, as underlined in [1]. Also in this case, BNT techniques ensure failure isolation, under the limits described above, to the extent in which the same paths are used for both topology discovery and failure detection. In the experiments of Figures 2.9 and 2.10 we simulate a server positioning problem where we use the heuristic *Greedy for Identifiability* (GI) proposed in [65] to determine the location of the server which optimizes node identifiability, given the position of m clients. By considering one client per service, GI deploys a fixed number of paths m such that at each step the number of identifiable nodes is maximized. In the experiments, we run 20 trials for each value of m. We remark that this experiment provides a lower bound, possibly loose, on the achievable number of identifiable nodes, for varying m. We compare this lower bound to the upper bounds given by Theorems 2.4.1 and 2.4.2 with both $d_{\mathtt{max}}$ and $\bar{d}$, where the value of $d_{\mathtt{max}}$ has been set based on the paths chosen by the GI heuristic (i.e., in a range between 2 and 5). The paths placed by GI follow a deterministic shortest path routing scheme, that is a special case of consistent routing.

2.6 Conclusions

In this chapter, we consider the problem of maximizing the number of nodes whose states can be identified via Boolean Network Tomography. By combinatorial analysis we derive upper bounds on the number of identifiable nodes under different assumptions, including general routing (arbitrary and consistent) as well as QoS requirements (path lenghts). These bounds show the fundamental limits of failure identifiability via Boolean Network Tomography in both real and engineered networks. The bound analysis gives new insights for the design of topologies with high identifiability in different network scenarios, and in particular we propose a polynomial-time algorithm, called ICA, that takes into consideration such constraints to design a network that meets the bounds for the case of arbitrary routing. Through analysis and experiments we evaluate the tightness of the bounds and demonstrate the efficacy of the design insights for engineered as well as real networks.

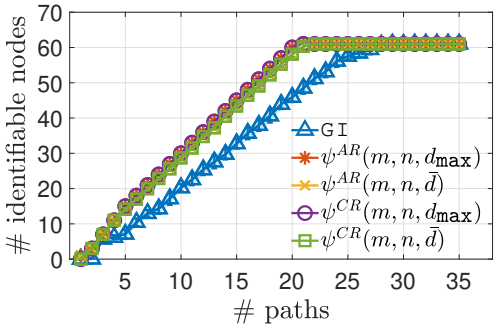


Figure 2.9. Upper bounds of Th.s 2.4.1 and 2.4.2 and lower bound of GI on US Signal topology.

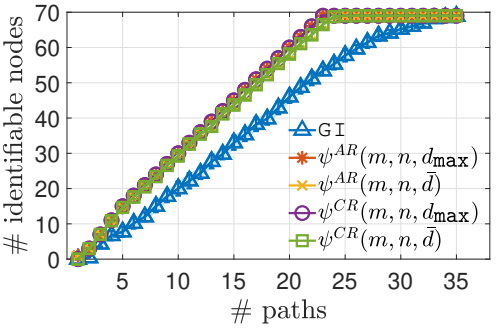


Figure 2.10. Upper bounds of Th.s 2.4.1 and 2.4.2 and lower bound of GI on Uninett topology.

Chapter 3

Resources Bounds for Identifiability in Boolean Network Tomography

In Boolean Network Tomography (BNT), node identifiability is a crucial property that reflects the possibility of unambiguously classifying the state of the nodes of a network as *working* or *failed* through end-to-end measurement paths. In this chapter, we extend the analysis introduced in Chapter 2, and we provide theoretical bounds on the minimum number of measurement paths that are necessary to guarantee identifiability of a given number of nodes. The bounds take into consideration two different classes of routing schemes (arbitrary and consistent routing) as well as quality of service (QoS) requirements. We formally prove the tightness of such bounds for the arbitrary routing scheme, and we show that the algorithmic approach for network topology design introduced in Chapter 2 works in this context, too. Due to the computational complexity of the optimal solution, we evaluate the tightness of our lower bounds by comparing their values with an upper bound, obtained by a state-of-the-art heuristic for node identifiability. For our experiments we run extensive simulations on both synthetic and real network topologies, for which we show that the two bounds are close to each other, despite the fact that the provided lower bounds are topology agnostic.
The results shown in this chapter are published in [5].

3.1 Introduction

With the massive growth of the Internet, localizing node failures has become a crucial task. Single organizations have direct access only to limited portions of the internal nodes of the network, and they hardly collaborate in sharing internal performance observations because of commercial conflicts. With similar motivations as those listed in Chapter 2, in the work described in this chapter, we provide topology-agnostic lower-bounds to the minimum number of measurement paths which are necessary to guarantee identifiability to a desired number of nodes. Such bounds represent the dual solution to the optimization problem studied in Chapter 2 ([16]), where we introduced upper-bounds to the maximum number of identifiable nodes given a number of monitoring paths. In contrast with existing literature, we propose theoretical lower-bounds that cannot be violated, independently of the specific characteristics of the topology. The bounds formulations are only based on the number of nodes to identify, on high level routing consistency properties (*arbitrary* and *consistent routing*), and on QoS requirements, expressed in terms of maximum allowed path

Notation	Description
P	Set of m monitoring paths $P = \{p_1, \ldots, p_m\}$
$p \in P$	Monitoring path as list of nodes
$b(v)$	Binary encoding of node v wrt P
$b(v)\|_i$	i-th element of $b(v)$ (equal to 1 iff $v \in p_i$, to 0 otherwise)
$\chi(v)$	Crossing number of node v wrt a set of paths P
P_F	Incident set of paths of a failure set F
$\mathcal{I}(p)$	Set of identifiable nodes traversed by path p
$M(p)$	Path matrix of path p
$\mathcal{B}(k)$	$\{b \in \{0,1\}^m,\ s.t.\ \sum_{i=1}^{m} b\|_i = k\}$, $k = 1, \ldots, m$
B_V	Set of binary encodings of a set of nodes V

Table 3.1. Notation table.

length. Motivated by the need to complement the analysis of [16], our bounds are a useful tool to measure the capability of a monitored topology to efficiently identify the status of its components. Implementing a monitoring system comes with the cost of installing monitors on the nodes of a network and of traffic caused by path probing; with this work, we aim at providing fundamental guidelines and minimal requirements for achieving the desired level of network identifiability (e.g., number of identifiable nodes).
We hereby list the major contributions presented in this chapter.

- We study theoretical bounds on the minimum number of paths to deploy in a network for identifying a desired number of nodes. The bounds do not depend on specific network topologies (i.e., they are topology agnostic bounds).
- We adapt the Incremental Crossing Arrangement (ICA) algorithm (Algorithm 1), to the scenario studied in this chapter, and we use it to prove tightness of the proposed bounds.
- We evaluate the tightness of our bounds on both synthetic and real network topologies. For this purpose we compare the bounds with the results of a state-of-the-art greedy algorithm, hereby referred to as Greedy for Identifiability (GI), for maximizing network identifiability by means of client-to-server probing paths [65].

3.2 Problem formulation

We represent a network as a undirect graph $G = (V, E)$, where V is the set of the nodes of G and E is the set of its edges. Each node v is either in *working* or *failed* state. In Table 3.1 we sum up the notation that will be used throughout this chapter. The state of the nodes is assessed indirectly by a set of *monitoring paths*, $P := \{p_1, \ldots, p_m\}$, each being represented as the ordered sequence of nodes it traverses. Node failures cause paths disruption: when a path traverses a failed node, its communication is interrupted. On the other hand, paths traversing only working nodes are working. Each node v may be labeled with a binary encoding of length m, $b(v) \in \{0,1\}^m \setminus 0^m$, where $b(v)|_i = 1$ if v is traversed by path p_i, $b(v)|_i = 0$ otherwise. We call *crossing number* of a node v, $\chi(v)$, the number of paths that traverse v, i.e., the number of 1s in its binary encoding ($\chi(v) = \sum_{i=1}^{m} b(v)|_i$). For each path p_i we define a *path matrix* as a binary matrix $M(p_i)$, in which each row is the binary encoding of a node on the path, and rows are sorted according to the sequence p_i. Notice that by definition $M(p_i)|_{*,i}$ has only ones, i.e., $M(p_i)|_{r,i} = 1,\ \forall r$. We call the

incident set of v the set of paths traversing v and denote it with $P_v \subseteq P$.
We call *failure set* of a network, F, the set of all failed nodes. Here, we assume that nodes fail one at a time, and therefore that $|F| = 1$. In such a context, we focus on the property of 1-identifiability. With reference to [17], we give the following definition following from Lemma 2.3.1:

Definition 3.2.1. *A node v_i is 1-identifiable with respect to $P = \{p_1, \ldots, p_m\}$ if $b(v_i) \neq 0^m$ and if for all $v_j \neq v_i$, $b(v_i) \neq b(v_j)$, i.e., its binary encoding is not null and not identical to that of any other node.*

Node identifiability allows non ambiguous node state assessment by means of end-to-end measurement paths. We highlight that in order to be identifiable, a node must be monitored at least by one path. For this reason, a node whose binary encoding is null cannot be identifiable. In this chapter, we give bounds on the minimum number of paths that are needed for letting $n \leq |V|$ nodes be 1-identifiable under arbitrary and consistent routing schemes.

3.3 General network monitoring

Similarly to how we did in Chapter 2, we now study the number of paths to place in a general network of which we know the number of nodes and monitoring paths, without assuming any specific interconnecting topology. In the following, we consider a collection $\mathcal{P}$ of candidate path sets of known cardinality $m \geq 1$, between any endpoints. We analyze $\psi(\mathcal{P})$ in two cases: (i) arbitrary routing and (ii) consistent routing.

3.3.1 Arbitrary Routing

In this section we study the minimum number of monitoring paths that can be employed to identify n nodes in a network under arbitrary routing. We say that paths follow an arbitrary routing scheme if they do not traverse a node more than once, but they can cross each other non-restrictively.
In Section 3.2, we explained that nodes can be represented with binary encodings depending on what paths traverse them. In addition, we noticed that, in order for nodes in a network to be identifiable, they must have all different encodings. Since the number of different binary encodings of length m, excluding the string 0^m, is $2^m - 1$, the following holds:

Proposition 3.3.1. *The minimum number of monitoring paths to place in order to identify n nodes under arbitrary routing is $m_{min}^{AR} = \lceil \log_2(n+1) \rceil$.*

The bound represented by $m_{\mathtt{min}}^{AR}$ does not take into consideration the length of the paths involved. The length of a path p_i is the number of nodes it traverses, d_i ($d_i = |\{v \in V : b(v)|_i = 1\}|$). When constraints to the paths length are given, for instance by defining an upper bound to the maximum length, $d_i \leq d_{\mathtt{max}}$, or to the average path length, $\frac{1}{m}\sum_i d_i \leq \bar{d}$, the bound of Proposition 3.3.1 may change. In order to discuss the bound on the minimal number of paths under path length constraints, we observe the following facts:

Observation 3.3.1. *The number of distinct binary strings in $\{0,1\}^m$ with k 1s and $m-k$ 0s (with $0 < k \leq m$) is $\binom{m}{k}$. Out of them, there are $\binom{m-1}{k-1}$ strings where the $i-th$ digit is 1. In our context, this means that a path can traverse at most $\binom{m-1}{k-1}$ nodes having crossing number k in order to guarantee identifiability, that is when all encodings are distinct.*

Observation 3.3.2. *The maximum length d_i of a path p_i is $d_i = \sum_{i=0}^{m-1} \binom{m-1}{i} = 2^{m-1}$. In fact, the maximum number of identifiable nodes using m path under arbitrary routing is $n = 2^m - 1$ (see Proposition 3.3.1). The statement follows from Observation 3.3.1.*

These observations are illustrated in Figure 3.1. In this simple example we have $n = 7$ nodes. Assuming arbitrary routing, $m = \log_2(8) = 3$ monitoring paths are enough to identify all nodes. Each path p_1, p_2, p_3 traverses $\binom{m-1}{k-1} = \binom{2}{k-1}$ nodes v with crossing number $\chi(v) = k \in \{1, \ldots, m\}$, and the length of each path is $2^{m-1} = 4$.

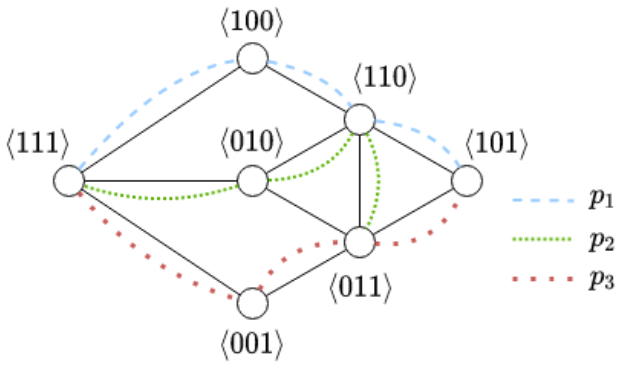


Figure 3.1. Arbitrary routing example.

In Theorem 3.3.1 we provide a lower bound to the number of paths to place in order to identify n nodes in a network, in the case a path length constraint is given.

Theorem 3.3.1. *The minimum number of paths $m_{min}^{AR,d_{max}}$ of maximum path length d_{max} to identify n nodes under arbitrary routing is the solution of the following problem:*

$$\min m \; s.t. \left\lfloor \frac{l_{(i_{max}+1)} \cdot m}{i_{max}+1} \right\rfloor + \sum_{i=1}^{i_{max}} \binom{m}{i} \geq n, \tag{3.1a}$$

$$where \; i_{max} = \max \left\{ j : \sum_{i=1}^{j} i \cdot \binom{m}{i} \leq m \cdot D \right\}, \tag{3.1b}$$

$$D = \min \left\{ d_{max}, 2^{m-1} \right\}, \tag{3.1c}$$

$$and \; l_{(i_{max}+1)} = D - \sum_{i=0}^{i_{max}-1} \binom{m-1}{i}. \tag{3.1d}$$

Proof. In order to minimize the number of paths, we want to have as many distinct encodings as possible with the minimum number of 1s. This fact translates into a strategy that consists in incrementally increasing the crossing number of the monitored nodes until the fixed average path length $d_{\max}$ allows it or until there is no way that paths traverse more nodes without violating identifiability, equation (3.1c) (see Observation 3.3.2).
The quantity $i_{\max}$ in Equation (3.1b) says that paths can be placed in such a way that the nodes they traverse are all distinct nodes with crossing number $o < \chi(v) \leq i_{\max}$. The quantity $m \cdot D$ is a loose upper-bound to the maximum number of nodes traversed by m paths, where nodes with crossing number j are counted j times, as j paths traverse them. Depending on the value $l_{(i_{\max}+1)}$ in Equation (3.1d), some more paths with crossing number $(i_{\max}+1)$ may be traversed. Notice that $l_{(i_{\max}+1)}$ represents the number of nodes of crossing number $(i_{\max}+1)$ that each path p can traverse considering that it traversed all distinct nodes with crossing number $\leq i_{\max}$ (see Observation 3.3.1). Extended to all m paths, $\left\lfloor \frac{l_{(i_{\max}+1)} \cdot m}{(i_{\max}+1)} \right\rfloor$ in Equation (3.1a) is the number of distinct nodes with crossing number $(i_{\max}+1)$ that can be traversed by m paths of length $l_{(i_{\max}+1)} + \sum_{i=0}^{i_{\max}-1} \binom{m-1}{i}$.

□

Constraints on paths length are usually imposed by QoS requirements and influence substantially the minimum amount of paths needed to identify a certain number of nodes. In a network where shortest path routing schemes are applied, the value of $d_{\max}$ is the diameter of the network. Differently, multi-service networks serve for more than a single service, to which a number of clients access. Each service is characterised by a different service level agreement (SLA) that regulates the routing scheme to adopt as well as the reserved portion of the network. QoS requirements may also vary for each service. In this scenario, paths lengths may be different from one another depending on what service a path belongs to. Information about paths lengths for different services in multi-service networks justifies the introduction of the notion of average path length of a network, $\bar{d}$. Furthermore, average path length can be easily computed when the network topology and the routing scheme implemented on it are known.

Corollary 3.3.1. *The bound of Theorem 3.3.1 holds also when we consider the average path length, $\bar{d}$, or an upper-bound to it, instead of d_{max}. The statement of Theorem 3.3.1 only changes in Equation (3.1c), where the value of $\bar{d}$ is to be substituted to d_{max}. We call such bound $m_{min}^{AR,\bar{d}}$.*

The bound of Theorem 3.3.1 suggests that the number of nodes that m monitoring paths can identify grows with the path length, (Equation (3.1b)). Nevertheless, we can show that the growth stops for $d_{\max} > 2^{m-1}$.

Corollary 3.3.2. *The number of nodes that m paths can identify grows with d_{max} as long as $d_{max} \leq 2^{m-1}$.*

Proof. In Observation 3.3.1 we point out that, given a set of m paths, each of them can traverse at most $\binom{m-1}{k-1}$ nodes with crossing number $\chi(v) = k$. The maximum value for $\chi(v)$ is m, and therefore the maximum path length for a path is $\sum_{i=0}^{m-1} \binom{m-1}{i} = 2^{m-1}$. This fact motivates the expression in Equation (3.1b). □

We highlight that knowledge of path length does not necessarily imply explicit knowledge of the paths - in terms of what nodes they traverse.

Adapting Incremental Crossing Arrangement (ICA)

In this section, we see how ICA in Algorithm 1 can be adapted to the scenario considered in this chapter, where we are not given the number of measurement paths, but instead we have as a input parameter the number of nodes we want to identify. We shall then use the adapted version of ICA to show the tightness of the bounds of Theorem 3.3.1.

In order to adapt Algorithm 1 to design a network topology and routing scheme where the number of nodes to identify is given, it is enough to apply the following changes:

- **Input**: n, number of nodes to identify, and $\bar{d}$, average path length.
- **Output**: A set of encodings B_V which can be mapped onto a topology graph $G = (V, E)$ where all $|V| = n$ nodes are identifiable by using $m_{\min^{AR,d_{\max}}}$ paths with average length $\bar{d}$.
- **line** 1: Compute m according to Theorem 3.3.1 and Corollary 3.3.1.

Tightness of the bound under arbitrary routing In this section we show that the bound given by Theorem 3.3.1 can be achieved tightly for a specific family of topologies constructed via Incremental Crossing Arrangement (ICA) (Algorithm 1).

Proposition 3.3.2. *[Tightness of Theorem 3.3.1]*
For any $n \in \mathbb{Z}^+$ (positive integer) and $\bar{d} > 0$, there exists a set P of m monitoring paths with average length $\bar{d}$, such that m is the solution of the problem in Equations (3.1a) to (3.1d).

Proof. We recall that the Incremental Crossing Arrangement (ICA) algorithm builds a topology by creating nodes with unique encodings, in increasing order of crossing number, up to possibly $(i_{\max}+1)$.

To prove the proposition, we need to show that the minimum number of monitoring paths required to identify n nodes is provided in Theorem 3.3.1. ICA initially generates all the encodings of $\mathcal{B}(i)$, for $i = 1, \dots, i_{\max}$. As a consequence, it follows from Observation 3.3.1 that each path will traverse at least $d(i_{\max}) \triangleq \sum_{i=0}^{i_{\max}-1} \binom{m-1}{i}$ identifiable nodes. In fact, the encodings of the nodes of $\mathcal{I}(p_i)$ (identifiable nodes traversed by path p_i), must have a "1" in the i-th position. Therefore the number of distinct encodings corresponding to nodes of $\mathcal{I}(p_i)$ is at least equal to the number of binary sequences of $(m-1)$ elements, with up to $(i_{\max}-1)$ ones.

Under incremental crossing arrangement, each path also traverses other nodes with crossing number equal to $(i_{\max}+1)$. Each of these nodes will appear in exactly $i_{\max}+1$ paths. The number of such nodes is therefore given by $\left\lfloor \frac{\sum_{k=1}^{m}(d_k - d(i_{\max}+1))}{i_{\max}+1} \right\rfloor$.

In conclusion, with this construction, ICA generates the following number of node encodings:

- $\binom{m}{i}$ encodings corresponding to nodes with crossing number equal to i, for $i = 1, \dots, i_{\max}$, and
- $\left\lfloor \frac{\sum_{k=1}^{m}(d_k - d(i_{\max}))}{i_{\max}+1} \right\rfloor$ encodings corresponding to nodes with crossing number equal to $(i_{\max}+1)$.

The number of generated encodings does not change if ICA applies the path completion procedure, which consists in a replacement of an encoding $b' \in \cup_{i=1}^{i_{\max}} \mathcal{B}(i)$ with an encoding $b'' \in \mathcal{B}(i_{\max}+1)$. In both cases, ICA constructs the set B_V in a way that each encoding corresponds to a unique node, and the nodes are traversed by paths of average length $\bar{d}$, guaranteeing identifiability of all the nodes corresponding to the generated encodings.

In order to show that the number of paths provided by Theorem 3.3.1 is enough to identify at least n nodes in the network, we need to prove that $\left\lfloor \frac{\sum_{k=1}^{m}(d_k - d(i_{\max}1))}{i_{\max}+1} \right\rfloor = \left\lfloor \frac{l_{(i_{\max}+1)} \cdot m}{i_{\max}+1} \right\rfloor$. This holds because $\sum_{k=1}^{m}(d_k - d(i_{\max})) = \sum_{k=1}^{m} d_k - m \cdot d(i_{\max})$, that is equal to $l_{(i_{\max}+1)} \cdot m$ in Equation (3.1d), being $\bar{d} = \frac{1}{m}\sum_{k=1}^{m} d_k$.

□

Notice that Proposition 3.3.2 requires $\bar{d} \leq 2^{m-1}$ as having longer paths would require at least a path to traverse different nodes with duplicate encodings, losing identifiability with respect to the bound value.

While Proposition 3.3.2 gives a characterization of sufficient conditions for building a network topology achieving the bound, we note that there exist topologies that do not meet the conditions, but still achieve the bound.

Observation 3.3.3. *The statement of Proposition 3.3.2 holds also when d_{max} is given instead of $\bar{d}$. In fact this simply translates into the more general scenario where initially $d_i = d_{max}$ is assigned to all paths and where path completion is not performed. Indeed, path completion does not serve for identifiability increase, but only to meet the input condition on the average path length.*

3.3.2 Consistent routing

As we have seen in Theorem 3.3.1, given a number of nodes to identify, the number of required paths can be logarithmic in the number of nodes. Nevertheless the bound of Theorem 3.3.1 is achieved only when the routing scheme allows paths to traverse arbitrary sequences of nodes.

If routing needs to meet additional requirements, the theoretical bound given by Theorem 3.3.1 can be increased.

We now consider the impact of the routing scheme on the identifiability of nodes via Boolean tomography. In the sequel, we assume that paths satisfy the following property of *routing consistency*.

Definition 3.3.1. *A set of paths P is* consistent *if $\forall p, p' \in P$ and any two nodes u and v traversed by both paths (if any), p and p' follow the same sub-path between u and v.*

We remark that routing consistency is satisfied by many practical routing protocols, including but not limited to shortest path routing (where ties are broken with a unique deterministic rule). Note that routing consistency implies that paths are cycle-free. In the following, we recall from [16] necessary for the proof of Theorem 13.3.2, where we provide the bound on the minimum number of paths under consistent routing.

Definition 3.3.2. *A column $M(p)|_{*,k}$ $(k = 1, \ldots, m)$ of a path matrix $M(p)$ has* consecutive ones *if all the "1"s appear in consecutive rows, i.e., for any two rows i and j $(i < j)$, if $M(p)|_{i,k} = M|_{j,k} = 1$, then $M|_{h,k} = 1$ for all $i \leq h \leq j$.*

We recall, as proved in Lemma 2.4.2 in Chapter 2, that under the assumption of consistent routing, all the columns in all the path matrices have consecutive ones. Furthermore, in Lemma 2.4.3, we showed that given $m = |P| > 1$ consistent routing paths, each path p_i having length d_i, the maximum number of different encodings in the rows of $M(p_i)$ is upper-bounded by $\min\{d_i; 2 \cdot (m-1)\}$.

We show these results, formally proved in Chapter 2 ([16]), by the simple example of Figure 3.2, where all nodes are 1-identifiable under consistent routing. Routing consistency is verified as the i-th column of all path matrices $M(p_i)$ is $\mathbf{1}$.

$$M(p_1) = \begin{bmatrix} \mathbf{1} & 0 & 0 \\ \mathbf{1} & 1 & 0 \\ \mathbf{1} & 0 & 1 \end{bmatrix} \quad M(p_2) = \begin{bmatrix} 0 & \mathbf{1} & 0 \\ 1 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 1 \end{bmatrix} \quad M(p_3) = \begin{bmatrix} 0 & 0 & \mathbf{1} \\ 0 & 1 & \mathbf{1} \\ 1 & 0 & \mathbf{1} \end{bmatrix}$$

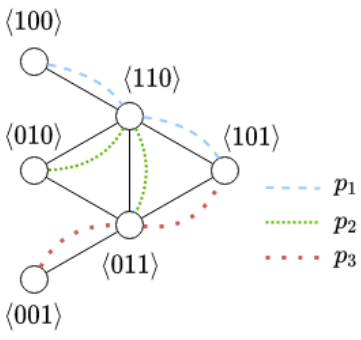


Figure 3.2. Consistent routing example.

Theorem 3.3.2. *The minimum number of paths $m_{min}^{CR,d_{max}}$ of maximum path length d_{max} to identify n nodes under consistent routing is the solution of the problem of Theorem 3.3.1, where Equation (3.1c) reads:*

$$D = \min\{d_{max}, 2 \cdot (m-1)\}. \tag{3.2}$$

Proof. The proof is analogous to the one of Theorem 3.3.1. The only difference in this case is that $D = \min\{d_{\max}, 2 \cdot (m-1)\}$, because of Lemma 2.4.3. □

The same considerations on the knowledge of the average path lengths for Theorem 3.3.1 hold for Theorem 3.3.2:

Corollary 3.3.3. *When $\bar{d}$ is known, the optimal solution $m_{min}^{AR,d_{max}}$ of the problem in Theorem 3.3.1, is a lower bound to the minimum number of paths to identify n nodes, if we substitute d_{max} with $\bar{d}$ in Equation (3.2). We call $m_{min}^{CR,\bar{d}}$ the bound computed with $\bar{d}$.*

Corollary 3.3.4. *The bound provided in Theorem 3.3.2 may be achieved by allowing the maximum value for the crossing number of a node to be 3.*

Proof. We need to prove that the maximum value of $i_{\max}$ is 2 under the assumption of consistent routing. Let us assume that $\bar{d} \geq 2 \cdot (m-1)$, and therefore that $D = 2 \cdot (m-1)$. Recall that $i_{\max} = \max\left\{j : \sum_{i=1}^{j} i \cdot \binom{m}{i} \leq m \cdot D\right\}$. For $j = 2$ and $\forall m \geq 2$, it holds that

$$\sum_{i=1}^{2} i \cdot \binom{m}{i} = m + 2\frac{m(m-1)}{2} = m^2 < 2m \cdot (m-1),$$

whereas for $j = 3$:

$$\begin{aligned}\sum_{i=1}^{3} i \cdot \binom{m}{i} &= m^2 + 3\frac{(m-2)(m-1)m}{6} \\ &= \frac{m^3 - m^2}{2} + m > 2m \cdot (m-1) \quad \forall m.\end{aligned}$$

Since $\sum_{i=1}^{N} i \cdot \binom{m}{i}$ is a growing function of N, $i_{\max}$ is at most 2.

□

Case of study: grid networks

The bound provided in Theorem 3.3.2 is tight on square grid networks with n^2 nodes, using $2n-1$ paths of maximum length $d_{\max} = n$. By contradiction, assume $d_{\max} = n$ and $m_{\min}^{CR,d_{\max}} = 2n-2$. It is easy to see that $i_{\max} = 1$ $\forall n \in \mathbb{N}$, $n > 2$. Therefore $l_{(i_{\max}+1)} = n-1$ and the number of nodes that may be identified with $m = 2n-2$ paths is $(n-1)^2 + 2n - 2 = n^2 - 1 < n^2$. An example of such topology and of paths placement is in Figure 3.3.

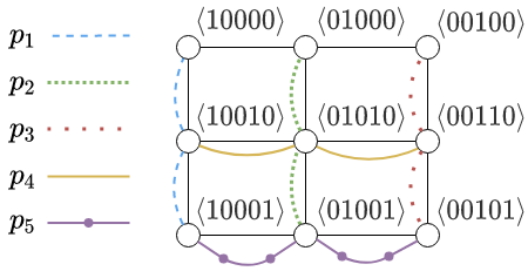


Figure 3.3. 3×3 grid network.

3.4 Experimental Results

We evaluate the tightness of the bounds proposed in the previous sections in comparison to with the results obtained by a state-of-the-art heuristic ([65]). For this purpose we run experiments on synthetic as well as real network topologies, implemented in Matlab. First, in Figures 3.4 and 3.5 we show the trend of the bounds on the minimum number of paths for the identification of n nodes, for the two cases of arbitrary and consistent routing, respectively (i.e., bounds of Theorems 3.3.1 and 3.3.2), under varying n, and path length $d_{\texttt{max}}$. Observe that the dependence on the path length of the values of $m_{\texttt{min}}^{AR,d_{\texttt{max}}}$ and $m_{\texttt{min}}^{CR,d_{\texttt{max}}}$ is stronger for smaller values of $d_{\texttt{max}}$. This is an expected behaviour, as in Equation (3.1d), it holds that $D = 2^{m-1}$ for all values of $d_{\texttt{max}} \geq 2^{m-1}$. As a result, for all such values of $d_{\texttt{max}}$, the minimum number of paths needed to identify n nodes is the same. This phenomenon is more evident in the case of consistent routing (Figure 3.5) because $D = \min\{d_{\texttt{max}}, 2 \cdot (m-1)\}$ (see Equation (3.2)), and therefore $D = 2 \cdot (m-1)$ for all $d_{\texttt{max}} \geq 2 \cdot (m-1)$.

3.4.1 Topologies

We hereafter list the topologies (synthetic and real) used in our evaluation:

1. **Random Geometric (RG) graphs.** RG graphs are synthetic topologies [64] built as follows: n nodes are placed in a unit square and a link is added between any pair of nodes whose distance is lower than or equal to a threshold parameter $\rho > 0$. In our experiments, we generate random coordinates (x_i, y_i) for each node v_i and we vary the value of ρ. This model well simulates ad-hoc wireless networks.
2. **Jellyfish topology.** Introduced in [129], Jellyfish are emerging data centre topologies which offer high throughput and capacity, high scalability and failure resiliency. The internal nodes of the Jellyfish (nodes with degree strictly greater than one) are network switches, whereas leaf nodes are servers.
3. **US Signal.** This is the real topology of a fiber optical network in the USA. This topology was made available in the Topology Zoo archive [82], and is composed of 63 nodes and 133 edges.
4. **Uninett.** This is an existing Internet topology located in Norway. It has 69 nodes and 98 edges. It was also taken from the Topology Zoo archive [82].

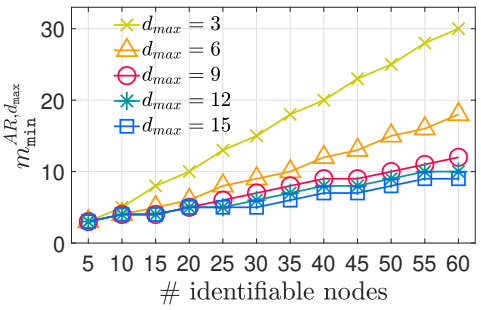


Figure 3.4. Bound of Th. 3.3.1 $m_{\texttt{min}}^{AR,d_{\texttt{max}}}$, varying paths lengths.

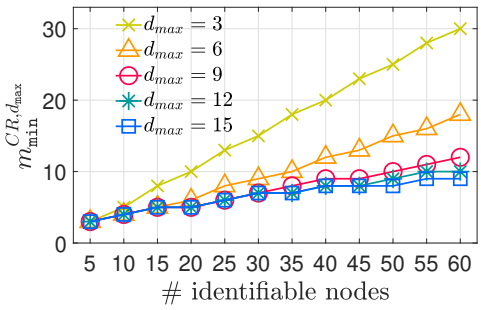
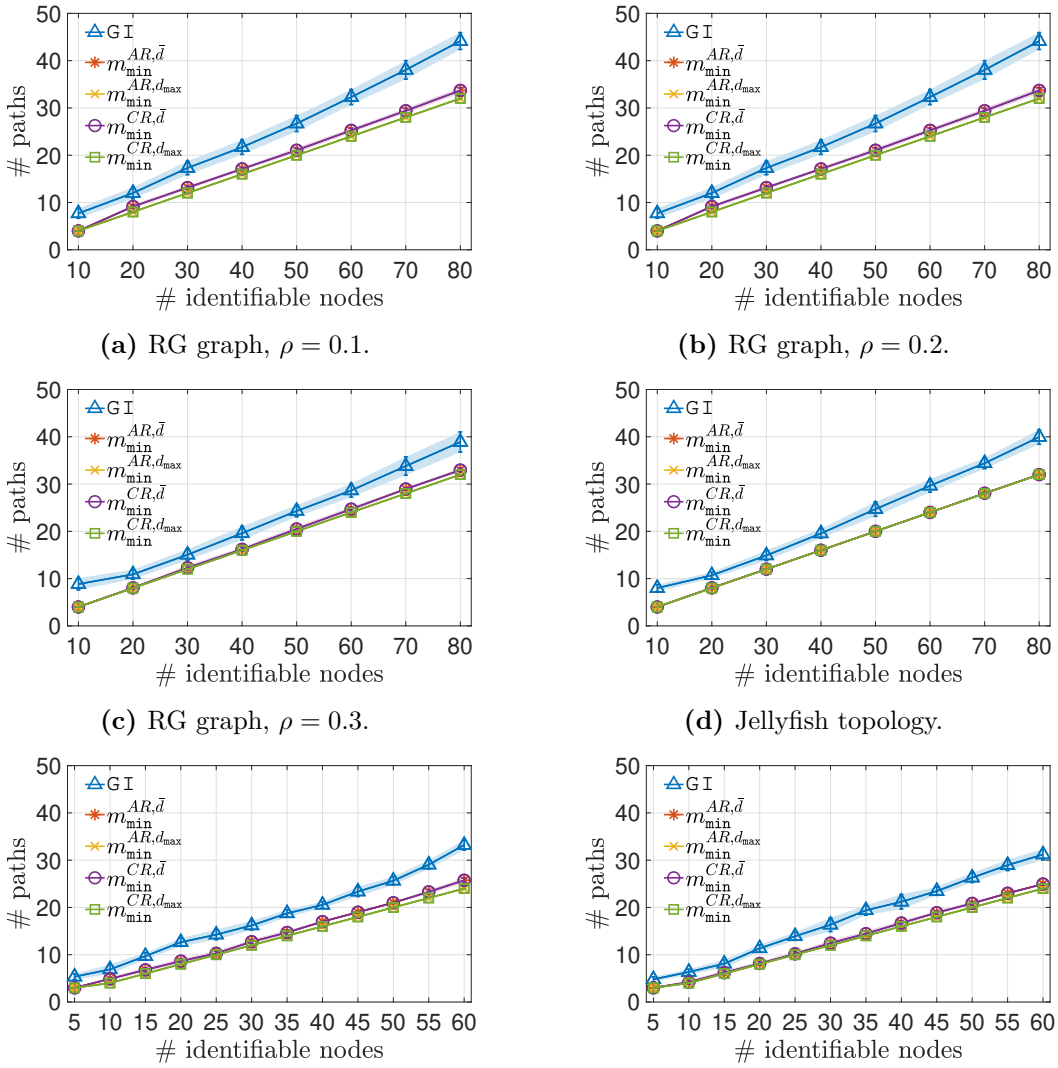


Figure 3.5. Bound of Th. 3.3.2 $m_{\texttt{min}}^{CR,d_{\texttt{max}}}$, varying paths lengths.



(a) RG graph, $\rho = 0.1$.

(b) RG graph, $\rho = 0.2$.

(c) RG graph, $\rho = 0.3$.

(d) Jellyfish topology.

(e) US Signal.

(f) Uninett.

Figure 3.6. Number of paths to identify variable numbers of nodes on different topologies. $d_{\max} = 4$.

3.4.2 Benchmark heuristic

In order to evaluate the tightness of our bound, we use a state-of-the-art greedy for identifiability (GI) proposed in [65], as a benchmark for comparisons, and we adapt it to our scenario. GI was originally proposed as an algorithm to place servers for addressing Quality of Service (QoS) and failure identifiability requirements in a joint manner. Given multiple services, and related client sets, the algorithm finds the most suitable server location, among those satisfying QoS requirements, to optimize failure identifiability. The selected server locations are such that the client-to-server paths form several intersecting trees, one for each service, where servers are located at the roots and clients are located at the leaves. GI uses a greedy approach that iteratively selects a number of server positions, one for each service, such that the identifiability obtained by the adopted client-to-server paths, is maximized at each iterative step. The authors prove that the number of paths placed by this heuristic is a

constant approximation of the optimal solution.

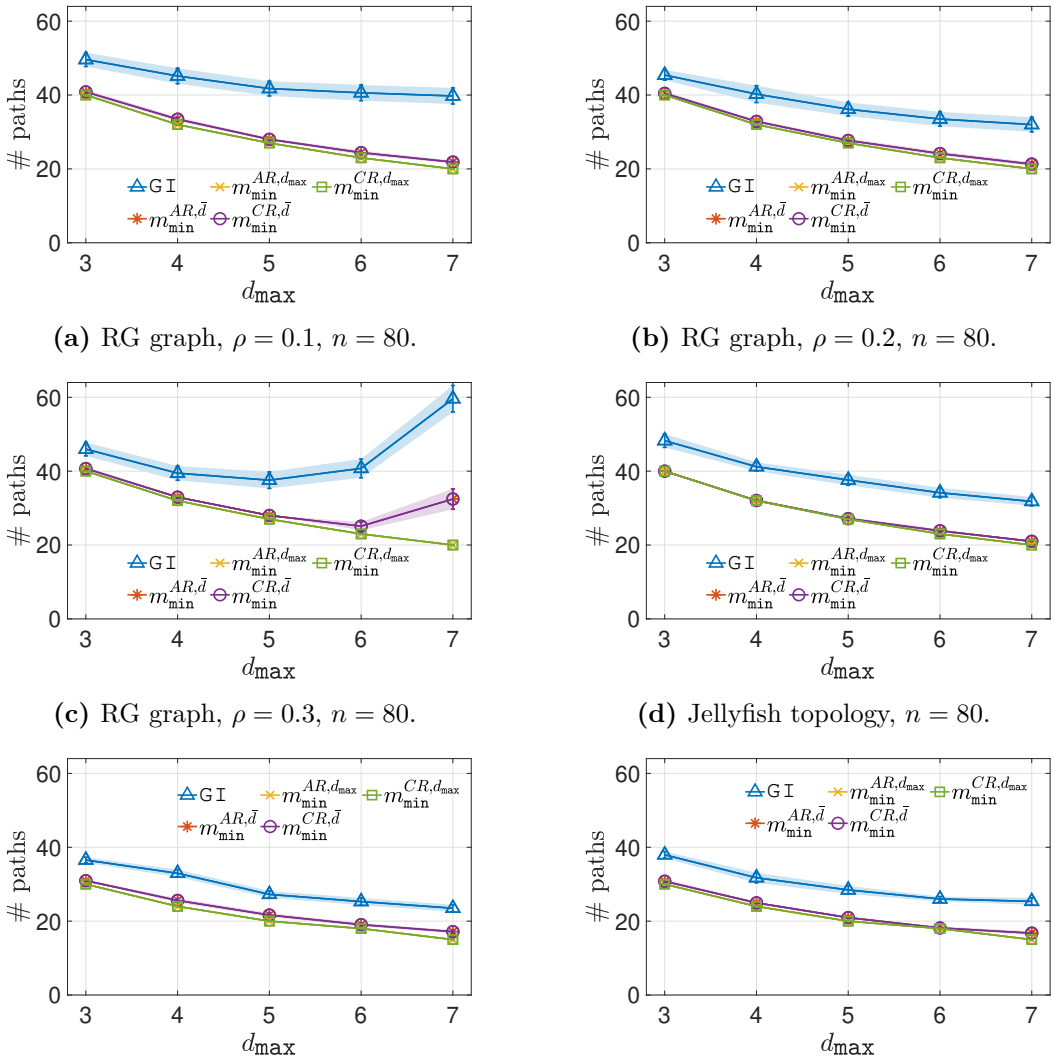
In our experiments we modify the GI approach to obtain an upper bound to the number of paths that are necessary to uniquely identify the state of a given number of nodes n. In particular, in order to ensure maximum flexibility to the choice of the set of monitoring paths we consider only one client for each server, and a number of paths that is equal to the number of deployed servers. Moreover, we relax the quality of service requirements to obtain all the server locations which are at a distance lower than or equal to $d_{\texttt{max}}$ from the client. The algorithm ends as soon as the selected paths are sufficient to identify the desired number of nodes n.

In our implementation of GI communication between any two endpoints is obtained through a shortest path routing algorithm. The adoption of a deterministic tie breaking rule ensures that the obtained routing scheme is deterministic. In order to prevent the use of degenerate paths (i.e., paths traversing only one node), servers are never located on the same node as the related client.

In order to boost identifiability of the greedy approach, we consider a preliminary phase where a set of paths is deployed according to a greedy for node coverage approach. This coverage phase is also common to [103], and is motivated by the observation that greedy for identifiability approaches select short paths in the early iterations, to obtain maximum identifiability, which prevents further identification in the later steps of the algorithm, due to insufficient node coverage, an issue that is easily solved by letting the algorithm use longer paths in the early execution steps.

3.4.3 Tests

The bounds of Theorems 3.3.1 and 3.3.2 are compared with the results obtained by GI, which provides an upper bound to the minimum number of paths that are necessary to uniquely identify the state of a given number of nodes n. We carry out two different sets of experiments. In the first set, the maximum path length is fixed, $d_{\texttt{max}} = 4$, whereas the number of nodes to be identified is variable. Figure 3.6 shows the curves of the number of paths necessary to identify an increasing number of nodes for GI with respect to the bounds. The bounds are also evaluated with the average path length resulting from the path choices of GI. Each figure represents the experiment on a different topology. For random topologies (random geometric graphs and jellyfish topologies), we generate graphs of 100 nodes. GI is then run on all such topologies with an increasing number of nodes to identify (from 10 to 80). For the fiber and the internet topologies instead, the number of nodes to identify goes from 10 to 60, being 63 and 69, respectively, the total number of nodes of the networks.
A similar setting was also implemented in the second set of experiments, depicted in Figure 3.7. For these experiments, curves represent how the number of paths necessary to identify a fixed number of nodes n changes depending on different values of $d_{\texttt{max}}$. For random topologies, $n = 80$, whereas for US Signal and Uninett, $n = 60$. Also in this case, we generated random topologies having 100 nodes.
Tests on random topologies have been executed by generating 20 different graphs of each type. Shades and bars in the curves of random topologies represent the standard deviation of the mean number of paths used by GI and of the bounds with variable average path length, $\bar{d}$ (Figures 3.6a-3.6d and Figures 3.7a-3.7d). In contrast, shades and bars in the curves related to real topologies (US Signal and Uninett, Figures 3.6e and 3.6f and Figures 3.7e and 3.7f) represent the standard deviation of the mean number of paths used by a randomized version of GI where routing consistency is still satisfied. We tested on random geometric graphs with different values of ρ (0.1, 0.2, 0.3) in order to analyse the goodness of our bounds on graphs with different properties. When generating random geometric graphs, there is no guarantee that the resulting graph is connected, unless $\rho = \sqrt{2}$ (in this case, the RG graph is a clique, being $\sqrt{2}$ the maximum distance between two nodes in an unit square). Experimentally speaking, we encountered non-connected graphs only for $\rho = 0.1$. When this



(a) RG graph, $\rho = 0.1$, $n = 80$.

(b) RG graph, $\rho = 0.2$, $n = 80$.

(c) RG graph, $\rho = 0.3$, $n = 80$.

(d) Jellyfish topology, $n = 80$.

(e) US Signal, $n = 60$.

(f) Uninett, $n = 60$.

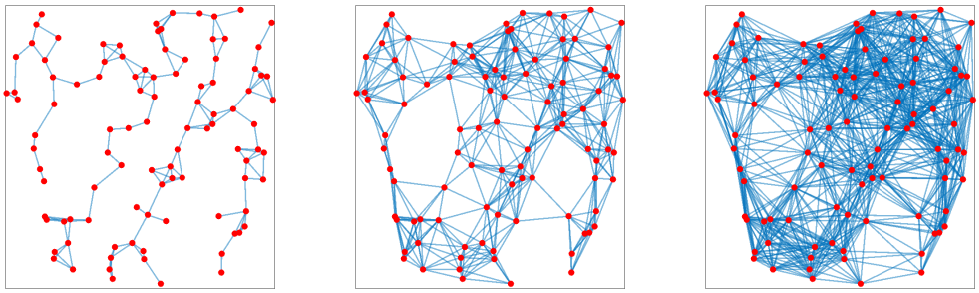
Figure 3.7. Number of paths to identify n nodes with variable values of $d_{\max}$ on different topologies.

event occurs, we link together the least number of nodes belonging to different connected components that are the closest (by means of the Euclidean distance), until the graph is connected. Variations of ρ have a great impact on the structure of the resulting graph. The characteristics of the set of 20 random geometric graphs generated with 100 nodes, with respect to different values of ρ are summed up in Table 3.2. Figure 3.8 shows an example of how different values of ρ change the structure of a random geometric graph with 100 nodes.

Notice that, when ρ increases, so does the average node degree, whereas the network diameter decreases correspondingly. When the maximum path length is fixed to 4 (Figures 3.6a-3.6c), different values of ρ do not imply sensible differences in the performance of GI, whose trend corresponds to the one of our bounds. This confirms the fact that our bounds do not depend on topology structures that can be extremely different. On the other hand, when the maximum path length is variable (Figures 3.7a-3.7c), we can observe the following facts: first of all, in order to test for $d_{\max} = 7$, it is necessary to set the condition that the

	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.3$	$\rho = 0.3^*$
∂_{min}	1	2.1	7.25	6.35
∂_{max}	7.75	18.8	34.2	33.2
∂_{avg}	3.28	10.31	20.99	20.65
δ	39.3	9.75	6.45	7

Table 3.2. Average properties of 20 random geometric graphs of 100 nodes for different values of ρ. Here $\partial_{min/max/avg}$ are the mean minimum, maximum and average degrees of the nodes. δ is the average diameter of the graph.



(a) $\rho = 0.1$ **(b)** $\rho = 0.2$ **(c)** $\rho = 0.3$

Figure 3.8. Random geometric graphs with 100 nodes having the same geometric coordinates, built with different values of ρ.

graph diameter δ is greater or equal to 7. In our experiments, this condition is always met for $\rho = 0.1$ and 0.2, whereas the same does not hold for $\rho = 0.3$ (see Table 3.2, column $\rho = 0.3$, where $\delta < 7$). In Table 3.2, the column $\rho = 0.3^*$ corresponds to random geometric graphs with $\rho = 0.3$ that satisfy the condition $\delta \geq 7$. We generated graphs satisfying this condition for the experiments in Figure 3.7c. Figures 3.7a-3.7c show that our bounds are closer to the curves representing the performance of GI when $\rho = 0.2$. As a matter of fact, when $\rho = 0.3$, the average distance between nodes (in number of hops) is smaller, and the graph diameter is never greater than 7, our maximum path length. As a consequence, nodes are highly connected on average. For this reason, despite the greater path length availability, only a very few paths reach the maximum path length. On the other hand, after the coverage phase, more shorter paths are needed in order to guarantee that $n = 80$ nodes are identifiable, implying that the average path length decreases, as we can observe by the growing trend of the curves $m_{\texttt{min}}^{AR,\bar{d}}$ and $m_{\texttt{min}}^{CR,\bar{d}}$. On the other hand, for $\rho = 0.1$, a few nodes have a high centrality, whereas most of the nodes have degree 2, meaning that the majority of the network nodes are distributed in chains. As node identifiability holds when nodes have unique encodings, e.g., different sets of paths crossing them, chains are structures that are hard to identify by means of monitoring paths, and large path length do norepresent an advantage for this specific structure. When $\rho = 0.2$, the resulting graphs are not star-shaped, but at the same time long paths are available, showing their stronger identification power. On jellyfish topologies (Figures 3.6d and 3.7d), our bounds are very close to the results obtained by GI, with negligible differences between the curves. One of the properties of jellyfish topologies is that servers can reach one another with shorter paths with respect to other topologies for data centres (e.g., fat trees), [129], for this reason the curves of our bounds are tighter for smaller values of $d_{\texttt{max}}$ in Figure 3.7d. Despite this, in both configurations (Figures 3.6d and 3.7d) the curves representing the results of GI scale analogously with our

bounds.
The results shown in this section highlight that our bounds are very close to the number of monitoring paths that a greedy algorithm would use in order to guarantee nodes identifiability. We stressed our experiments to evaluate our bounds on synthetic networks with very different structures and connectivity properties. Experiments show that the bounds presented in this work represent a very trustful estimation of the number of paths for node identifiability on Jellyfish topologies, as well as on real internet and fiber optical networks. In addition, we can also observe that knowledge of $\bar{d}$ can be used to provide tighter bounds, specially when there are few paths of length $d_{\texttt{max}}$ in the network.

3.5 Conclusions

In this chapter we provide theoretical lower bounds on the minimum number of monitoring end-to-end paths necessary to achieve the desired level of identifiability in a network in terms of identifiable nodes. We study how the routing scheme affects the bound values by giving two different formulations, for arbitrary and consistent routing, respectively. We also study how requirements on the maximum and average path length affect the bound formulation, highlighting the dependence of the minimal number of required paths on QoS constraints. We carried out an extensive set of experiments on synthetic and real topologies to evaluate the tightness of the proposed lower-bounds. The synthetic topologies that we used are commonly employed for modelling ad-hoc wireless networks and data centers, whereas the real networks are an internet and a optical fiber network located in Norway and in the USA. We used a state-of-the-art algorithm for network identifiability maximization via path deployment to obtain feasible solutions as upper bounds of the optimum and evaluate the tightness of the proposed lower bound. We show that the provided bounds provide a high approximation in all the performed experiments.

Appendix

3.A An analogy with separating systems

Separating systems were introduced by Renyi in [120] in the context of Information Theory. In this section, we analyse the analogy that holds between separating systems and sets of paths guaranteeing node identifiability under arbitrary routing, and between their cardinality.

Definition 3.A.1. *Let H be a finite set of elements. The family $\mathcal{A} \subseteq 2^H$ is a* separating system *if for every $x, y \in H$, $x \neq y$, there is a set $S \in \mathcal{A}$ such that $x \in S \wedge y \notin S$ or $x \notin S \wedge y \in S$.*

We can translate Definition 3.A.1 in the scenario of measurement paths by considering $H = V$, set of nodes in the network, and $\mathcal{A} = P \subseteq 2^V \setminus \emptyset$, set of paths, each being represented as the set of nodes it traverses. We have seen in Definition 3.2.1 that a set of nodes is identifiable if all such nodes have different binary encodings, and none of them is 0^m. In the scenario of separating systems, this means that every two nodes are identifiable if there exists at least one set S (i.e., one path p) that contains one and only one of such two nodes (i.e., if their binary encoding differs in at least one bit). Therefore, the property of a family of sets to be a separating system, corresponds to saying that such sets represent paths that ensure identifiability to the nodes they traverse. In contrast to Definition 3.A.1, from which it trivially follows that an empty set is a separating system, we exclude empty paths (i.e., paths not traversing any node) from our arguments. In [81], Katona addresses the problem of finding minimum size of a separating system, and in [68] a strengthening of the main theorem shown in [81] and reported in Theorem 3.A.1, is proved. We denote with $m(n,k)$ the smallest size of a separating system $\mathcal{A} \subseteq 2^{[n]}$ of sets of size k.

Theorem 3.A.1. *For $k < n/2$, $m(n,k)$ is equal to the smallest natural m, for which there exist natural numbers $j < m-1$ and $a < \binom{m}{j+1}$ such that:*

$$\begin{aligned} &\sum_{i=0}^{j} i\binom{m}{i} + a(j+1) \leq km, \\ &\sum_{i=0}^{j} \binom{m}{i} + a = n. \end{aligned} \tag{3.3}$$

This result is analogous to our result in Theorem 3.3.1, shifted by one, as a consequence of the fact that we exclude nodes of binary encoding 0^m. First of all, notice that the condition $k < n/2$ is justified by our result in Corollary 3.3.1, where we upper-bound $d_{\texttt{max}}$ with 2^{m-1} as $m_{\texttt{min}} = \lceil \log_2(n+1) \rceil$, and therefore $2^{m-1} = 2^{\log_2(n+1)-1} = (n+1)/2$. Then we observe that the role of j in Theorem 3.A.1 is the same of $i_{\texttt{max}}$ in Theorem 3.3.1, with the only difference being that $i_{\texttt{max}} \leq m-1$, whereas the term $\lfloor l_{(i_{\texttt{max}}+1)} \cdot m/(i_{\texttt{max}}+1) \rfloor$ in Theorem 3.3.1 represents the size of a proper subset of the $(i_{\texttt{max}}+1)$-uples out of m elements, and therefore it satisfies the condition on variable a in Theorem 3.A.1.

Chapter 4

Failure Localization through Progressive Network Tomography

In this chapter, we tackle the problem of localizing network failures by means of end-to-end monitoring paths. In contrast with the previous chapters, where we gave topology agnostic bounds on the number of identifiable nodes in the network, here we face the problem of detecting node faults under given failure scenarios. Boolean Network Tomography falls short of providing efficient failure identification in real scenarios, due to the large combinatorial size of the solution space, especially when multiple failures occur concurrently. First, we consider a *static failure scenario*. In this context, we aim at minimizing the number of probes to obtain failure identification. To face this problem we propose a progressive approach to failure localization based on stochastic optimization, whose solution is the optimal sequence of monitoring paths to probe. We address the complexity of the problem by proposing a greedy strategy in two variants: one considers exact calculation of posterior probabilities of node failures, given the observation, whereas the other approximates these values by means of a novel failure centrality metric. Secondly, we adapt these two strategies to a *dynamic failure scenario* where nodes states can change throughout a monitoring period. By means of numerical experiments conducted on real network topologies, we demonstrate the practical applicability of our approach. Our performance evaluation evidences the superiority of our algorithms with respect to state of the art solutions based on classic Boolean Network Tomography as well as approaches based on sequential group testing.
Part of the results shown in this Chapter were recently submitted and accepted by *IEEE International Conference on Computer Communications (INFOCOM)*, [6], drafted in collaboration with prof. Annalisa Massini, prof. Novella Bartolini and Sc. Federico Trombetti, and soon to be published.

4.1 Introduction

As discussed in Section 1.1, a major challenge of localizing failed nodes in a network by means of binary measurements comes from the fact that observations of the outcome of monitoring paths (working/failed) induce a system of Boolean equations that is commonly under-determined, hence allowing multiple solutions [106]. Moreover, exact assessment of the status of each network component is not always achievable if monitors can only be deployed on a given subset of nodes, and routing of probing paths is not controllable. In addition, as discussed in Chapters 2 and 3, when the number of potentially concurrent failures is

unbounded, maximum identification of failed components may require an enormous number of monitoring paths and related probes [16,17], which severely limits the applicability of the approach.

However, we notice that executing the probing activity in a progressive manner, according to which the next probing path is selected on the basis of the information obtained from the previous probes, is particularly helpful in reducing the number of required probes to assess the status of the network under a specific failure scenario. According to this approach, hereby referred to as *progressive BNT*, the outcome of any new network measurement is used to simplify the problem instance. In fact, we observe that if a monitoring path is traversed successfully, we can ascertain the status of all the traversed components as working. In contrast, if a monitoring path fails, it certainly contains at least a failed component. It follows that, depending on the current observation scenario, monitoring a path may contribute valuable knowledge to different degrees. To measure the incremental value of monitoring paths in a progressive probing activity, we introduce a new notion of path utility which takes account of the added failure localization information with respect to the previously obtained network assessment. By using the information obtained by monitoring a given subset of the available paths, we can calculate the posterior expectation of the utility of monitoring any of the paths which have not been probed yet. By applying a Bayesian approach we are able to design a stochastic optimization problem which maximizes the expected utility over a progressive monitoring activity. We formulate a dynamic programming approach to derive the optimal progressive policy to maximize failure identification. However, we point out that the aforementioned optimization is computationally intractable for two reasons. The first reason is the large size of the state space representation, where each path may contribute different pieces of information, depending on its outcome. The second reason is because the computation of the posterior probabilities of a path to work properly, is exponential in the number of paths composing the network. In order to cope with the described complexity we propose a simplified approach based on two fundamental pillars.

On the one hand, rather than resorting to dynamic programming which would require the exploration of exponentially-many intermediate states in the progressive execution of the probing activity, we propose a greedy approach, called Posterior Probability Greedy (PoPGreedy), that selects the path that more likely contributes disambiguation of the state of a large number of network components.

On the other hand we approximate the posterior probability of a path failure by means of a polynomially computable approximation metric, to which we refer with the name of *failure centrality*. Failure centrality of a node reflects the probability that a node is broken, based on the currently available observation. We call this approach *Failure Centrality Greedy (FaCeGreedy)*

In order to measure the failure localization capability of the proposed approaches and be able to provide a quantitative evaluation, we introduce four novel metrics to measure the accuracy in properly localizing working as well as failed nodes. We compare FaCeGreedy and PoPGreedy both in terms of failure detection performance and related execution time. The experiments show that FaCeGreedy provides an excellent approximation of the stochastic optimization approach, in a negligible time. Instances that require an execution time of a week for the exact optimization, are solved in a matter of minutes by FaCeGreedy.

We also compare the performance of FaCeGreedy with algorithms for failure assessment based on classic BNT approaches. Simulations show that, as expected, FaCeGreedy has superior performance as it localizes more failures with fewer probing paths than BNT approaches.

To complete our analysis, we compare FaCeGreedy with AdaptiveFinder [80], a state of the art solution based on sequential group testing, and with AdaptivePathConstruction (APC), [103], a routing-constraint algorithm, also based on sequential group testing, for link failure detection. We recall that, being a group testing solution and not a network tomography approach, AdaptiveFinder has much more freedom than FaCeGreedy in selecting

the composition of the probing sets in terms of network components. Despite the higher flexibility in selecting testing sets, AdaptiveFinder performs worse than FaCeGreedy when the number of paths is given, and during its progressive execution. APC instead investigates on the state of the network by means of end-to-end paths that are given as an input, similarly to our scenario. We translate the link failure problem into a problem of node failure localization. The experiments show that in all the experimented settings, setting the number of tests to the minimum required by FaCeGreedy to localize all the failed nodes, AdaptiveFinder only localizes about half of the failed components. AdaptiveFinder requires many more probing sets than FaCeGreedy to correctly localize all the failures. In addition, although APC works well on small networks where only few nodes fail, it performs worse than FaCeGreedy when large networks are involved and many multiple failures occur, as it requires up to three times more path probes than FaCeGreedy. Finally, we also show that the approach introduced in this chapter may be easily extended to deal with dynamic changes within the network. This evolution to our original approach comes with low computational cost. Our original contributions are the following:

- We formulate the problem of progressive network tomography in terms of stochastic optimization and Bayesian analysis.
- We give an exact solution approach and discuss its complexity, motivating the need to resort to polynomial heuristic approaches.
- We formulate a novel failure centrality metric to approximate the failure probability of a node, given the observation of the outcome of a given set of probing paths.
- We formulate four novel metrics to quantitatively measure the capability of a monitoring algorithm to properly localize network failure and reduce the localization uncertainty.
- We propose two greedy approaches, called PoPGreedy and FaCeGreedy, based on Bayesian utility maximization.
- We prove optimality approximation for PoPGreedy.
- By means of simulations conducted on real network topologies, we compare FaCeGreedy and PoPGreedy against classic Boolean Tomography approaches, as well as approaches based on sequential group testing, showing that the our solutions outperform the others in all the performance metrics, and in all the considered scenarios.
- We show computational inexpensive altered versions of PoPGreedy and FaceGreey (DPoPGreedy and DFaCeGreedy) to deal with dynamic changes.

4.2 Related Work

Network tomography employs path probing to localize network failures. Network tomography techniques are broadly categorized in two families depending on the metric of interest for the inspection, additive or non-additive. An additive metric establishes a linear relationship between the measurement of a path and measurements of individual links and nodes composing the path. Along this line of research, Tati et al. [135] proposed a path selection algorithm to improve link metric identifiability, by maximizing the rank of successful measurements subject to random link failures. The work of Ren et al. [119] proposed algorithms to determine which link metrics can be identified and where to place monitors to maximize the number of identifiable links, subject to a bounded number of link failures. Additive metric tomography was also studied in [66, 89], to identify of additive link metrics under topology changes.

In contrast, non-additive tomography refers to non linear relationships between the path and its component metrics. The most relevant examples are those related to congestion

or failure localization, where the dominant factor of a path state is the state of its worst performing component. In this thesis, we focus on the second of the aforementioned families, namely on the case of non-additive tomography, and more specifically Boolean Network Tomography.

The early works on this topic focused on best-effort inference. For example, Duffield et al. [41, 46] and Kompella et al. [83] aimed at finding the minimum set of failures that can explain the observed measurements, and Nguyen et al. [106] aimed at finding the most likely failure set that explains the observations. Later, the identifiability problem attracted attention. Ma et al. characterized in [92] the maximum number of simultaneous failures that can be uniquely localized, and then extended the results in [94] to characterize the maximum number of failures under which the states of specified nodes can be uniquely identified as well as the number of nodes whose states can be identified under a given number of failures. In contrast to [94], the work in [16, 17] provide fundamental bounds that are topology agnostic, i.e., only based on the number of monitoring paths and high level routing consistency properties. The related optimization problems have also been studied under different formulations. For instance, the work by Bejerano et al. in [18] formulates the problem of optimally placing monitors to detect failed nodes via round-trip probing and demonstrate its NP-hardness. The work by Cheraghchi et al. [31] formulates the identifiability problem for a graph-based group-testing framework, where the test sets are constrained by the topology. Nevertheless, in the addressed framework the test sets are not end-to-end paths, but just connected components determined by random walks on the monitored network graph.

Ma et al. [93] proposed polynomial time heuristics to deploy a minimum number of monitors to uniquely localize a given number of failures under various routing constraints. When monitoring is performed at the service layer, He et al. [65] proposed service placement algorithms to maximize the number of identifiable nodes by monitoring the paths connecting clients and servers. In [136], Tati et. al. tackle the problem of selecting paths under network failures by maximizing the robustness against failures under a budget constraint on probing cost, assuming a known failure distribution. They also propose a reinforcement learning algorithm to pick a set of robust paths while learning path robustness. Differently, in this chapter we perform path selection for failure identification rather than for robust paths discovery.

Boolean Network Tomography suffers from two problems which severely limit its practical applicability to real settings. A first limitation is in the usual assumption of knowing an upper bound on the number of congested or failed links. Such a limitation is mostly due to the size explosion of the candidate failure set scenarios. Most of the proposed works are not designed to work under an unbounded number of failing components. Second, the cited works aim at designing monitoring paths so as to ensure node failure identifiability, according to the definition given in [92], under any failure scenario that meet the above mentioned constraint on the number of failed elements.

Unlike these works we do not assume any bound on the number of failures and we focus on actual node state identification rather than on identifiability. In fact, we observe that by monitoring paths in a real failure scenario, it is possible to identify the state of network components that are not theoretically identifiable in all failure scenarios as prescribed in [92], but are so in the considered real case setting.

With a similar goal, the authors of [108] model the tomography problem as a Markov Decision Process, and solve it with a Q-learning technique. The actions of the decision process are related to the diagnosis of the congestion status of individual links. The work in [95] also utilizes machine learning techniques based on neural networks to infer a network topology from incrementally selected paths, with the purpose to predict the performance of paths that are not directly probed.

We adopt incremental path selection, with the purpose of identifying the state of individual network components on known topology networks. Based on the information

progressively gathered, the instance of the failure identification problem is updated and reduced.

One approach towards this the same goal is the algorithm AdaptiveFinder [80], which considers progressive monitoring of graph-based test groups. We consider this proposal as a benchmark for performance comparisons against our own approach. AdaptiveFinder considers a network graph, and creates arbitrary sets of connected network components to determine the next paths to test according to a progressive approach. Unlike this work we consider testing sets which are end-to-end monitoring paths, where pairs of monitors are connected by a series of nodes that strictly follows the routing protocol in use by the considered network. Similarly, Adaptive Path Constraction (APC), [103], is a group-testing routing-constraint algorithm for detecting link failures by means of BNT techniques, that aims at minimizing the number of path probes. Differently from our Bayesian approach, in APC the choice of the next path to probe follows a binary search based idea. As we will see in Section 4.7 thanks to the incremental knowledge constructed through our Bayesian decision support, our approach overcomes the limitation imposed by the routing algorithm and provides superior performance than the AdaptiveFinder and APC approaches.

In this chapter, we also take into account dynamic changes in the nodes of the network, and we show how our two proposed algorithms can be adapted to an online setting where a network outsider wants to infer the state of the nodes continuously. The problem of detecting failures occurring dynamically within a network attracted attention in recent years. A large portion of the available literature focuses on specific networks (e.g., data centres [10] and Wireless Sensor Networks (WSNs) [79, 102, 121, 134]). In [70], Huang et. al. highlight practical issues when tomography techniques are used to infer link degradation within a network. Their approach is divided into an initial offline phase (a set of paths covering the network is selected), followed by an online phase (where monitor nodes periodically probe measurements along the defined paths in order to track possible changes in the performance of the links). In [74], Johnsson et. al. propose a two-step algorithm to interpret and analyze the outcome of path probes in order to detect and localize failures. Differently from these works, we consider path selection to be the key part of the online phase: we not only provide a way to interpret data, but also we show how to obtain the most informative data.

4.3 Problem Formulation

We consider a network modeled as a graph $G = (V, E)$, and a set of monitor nodes $V_M \subseteq V$ (shortly called *monitors*). For each ordered pair (i, j) of monitors in V_M we consider a unique monitoring path whose sequence of nodes is only determined by the routing algorithm in use. We consider uncontrollable routing, (see [92]), i.e., monitoring paths are determined by the routing protocol used by the network, not controllable by the monitors. Routing between the monitors i and j is not necessarily symmetric, but is assumed to be deterministic, and known. We shortly denote with $\hat{m}$ the set of nodes traversed by the monitoring path m. We refer to $\mathcal{M}$ as to the set of monitoring paths available for probing the network.

By probing the paths of $\mathcal{M}$ it is possible to obtain indirect information on the state of the traversed nodes. Both working and failed paths provide helpful information for the state assessment of the network components. In particular, all the nodes of a working paths must be properly functioning, whereas a non working path must contain at least a broken component. By probing paths in a sequence, it is possible to determine the most suitable choice for the next path to probe, on the basis of the information gathered so far. We address the problem of designing a *Progressive Monitoring Policy (PMP)*, i.e., a sequence of paths to be probed one by one, such that we can identify the status of the largest number of nodes, in the minimum number of steps (number of paths). We refer to this problem as the *PMP problem*.

Table 4.3.1. Summary of notations

Notation	Description
$\hat{m}$	set of nodes traversed by path $m \in \mathcal{M}$
S_i	state of node v_i (failed if $S_i = 0$, working otherwise)
Z_j	state of path m_j (failed if $Z_j = 0$, working otherwise)
Z_j^o	observed (tested) state of path m_j
$\mathcal{A}$	$\mathcal{A} \triangleq \{a_1, a_2, \ldots, a_{\lvert\mathcal{M}\rvert}\}$: set of monitoring decisions
$\mathcal{T} \subseteq \mathcal{M}$	set of tested monitoring paths
$\mathcal{R} = \mathcal{M} \setminus \mathcal{T}$	set of not yet tested monitoring paths
$\mathcal{F} \subseteq \mathcal{T}$	set of failed tested monitoring paths
$O_{\mathcal{T}}$	set of observed outcomes of paths in $\mathcal{T}$

In the following, we denote with S_v the event 'node v works', and $\bar{S}_v$ the event 'node v is broken'. If no information is available concerning the distribution of failures in the network, it is reasonable to assume uniform probability of node failures, namely that all nodes have equal *prior probability* p to be damaged, that is $P(\bar{S}_i) = p$ for all $i \in V$, while we denote with $P(S_i) = (1-p)$ the prior probability that i is a working node. A path fails if at least one of its nodes fails, while a path works if all of its node work. Unlike classic tomography approaches, we do not assume any prior knowledge of the exact number of failed nodes.

Classic approaches to Boolean Network Tomography adopt the concept of k-identifiability [16, 17, 92], which refers to the capability of inferring the state of individual nodes from the state of the monitoring paths. A node v is k-identifiable in $\mathcal{M}$ if any two sets of failing nodes F_1 and F_2 of size at most k, which differ at least in v (i.e., one contains v and the other does not), cause the failures of different subsets of paths in $\mathcal{M}$. The concept of k-identifiability assumes knowledge of an upper bound k to the number of occurring failures, and characterizes nodes regardless of their status, failed or working, but only in terms of whether their status can be uniquely inferred by observing the outcome of the monitoring paths of $\mathcal{M}$. However, our setting is characterized by (1) absence of a bound on the number of simultaneous failures, (2) uncontrollable position of monitor nodes and (3) given routing algorithm. In such a setting, the PMP problem is particularly challenging as no node is guaranteed to be identifiable according to classic tomography. We note that, especially for large values of k, k-identifiability of a node is an unlikely condition that is hardly verified in real networks.

4.3.1 Bayesian utility of path probing

Let Z_j be the event that path $m_j \in \mathcal{M}$ properly works, and $\bar{Z}_j$ the event that path m_j fails. Under the assumption of uniform probability of node failures a *prior estimate* of the state probability of path $m_i \in \mathcal{M}$ is $P(Z_i) = (1-p)^{|\hat{m}_i|}$, and $P(\bar{Z}_i) = 1 - P(Z_i)$.

In our problem setting, the state of the network can only be observed by probing monitoring paths in a sequence of monitoring interventions. We denote by $\mathcal{A} \triangleq \{a_1, a_2, \ldots, a_{|\mathcal{M}|}\}$ the set of possible monitoring decisions, where decision a_i implies monitoring the network through path m_i.

We denote by $\mathcal{T} \subseteq \mathcal{M}$ the already monitored paths. We denote with Z_i^o the outcome of the probing activity along path m_i, for any $m_i \in \mathcal{T}$. Knowledge of the outcome of the paths in $\mathcal{T}$ constitutes a source of additional information $O_{\mathcal{T}} \triangleq \{Z_j^o | m_j \in \mathcal{T}\}$ that can be used to produce a better — a posteriori — estimate, of the network status. We denote with $\mathcal{R}$ the current residual set of paths, i.e., paths which have not been monitored yet, namely $\mathcal{R} \triangleq \mathcal{M} \setminus \mathcal{T}$. By knowing the values of the random variables in $O_{\mathcal{T}}$ we can update the *posterior estimate* of the state probability of any path $m_i \in \mathcal{R}$, which is $P(Z_i | O_{\mathcal{T}})$.

We note that the outcome of any monitoring path is as informative as it contributes

(a) Original topology **(b)** Pruned topology

Figure 4.3.1. Removal of a working node (in green)

identification of the status of individual network components, or decreases the size of the identification problem instance. More specifically, we observe that monitoring working or non-working paths contributes useful information for failure identification in different manners.

Information obtained by monitoring working paths: If a probed path works then all its traversed nodes work as well. After the observation of a working path, the current instance of the failure identification problem can be reduced by considering a logical representation of the network graph constructed by *pruning the working nodes* and short circuiting the incident edges, as in Figure 4.3.1. We call $m^{(\mathcal{T})}$ and $\hat{m}^{(\mathcal{T})}$ the path m in the pruned logical graph, after testing the paths of $\mathcal{T}$, and the set of nodes that the pruned path traverses, respectively. Similarly, the residual set of paths to be monitored $\mathcal{R}$ may be reduced as well when any two paths m_i and m_j traverse the same set of nodes after pruning working nodes, namely when $\hat{m}_i^{(\mathcal{T})} = \hat{m}_j^{(\mathcal{T})}$, after pruning, as in Figure 4.3.2. Likewise, paths consisting only of nodes which have been found to be working, will be removed as well, as a consequence of the pruning of all their nodes.

(a) Original topology **(b)** Pruned topology

Figure 4.3.2. Merge of paths after node pruning

For this reason, the utility deriving from probing a working path is set proportional to its number of nodes. More precisely, the utility of probing a path $m_i^{(\mathcal{T})}$ (obtained as a logical representation of m_i after pruning all the working nodes) is proportional to the amount of newly found working nodes $|\hat{m}_i^{(\mathcal{T})}|$. In addition to this we notice that the longer a working path is, the smaller the search area for locating failed nodes will be. In particular, by pruning certainly working nodes, it may also happen that some non-working nodes are identified *by exclusion*, namely because they belong to non-working paths, already monitored, which identify subsets of candidate non-working nodes, reduced to size one after pruning the newly found working nodes. We define as $\mathcal{F}_1^{(m,\mathcal{T})}$ the set of failed paths m_i which have already been monitored (i.e. $m_i \in \mathcal{T}$) and have length equal to 1 after node pruning resulting from path m being working. Hence, we consider another additive term, to the utility of monitoring working paths, i.e. $|\mathcal{F}_1^{(m,\mathcal{T})}|$. Consider the example of Figure 4.3.3. Assume that the monitoring activity starts by probing path m_2 first, which is found to be non-working, hence m_2 is inserted in $\mathcal{T}$. Then the monitoring activity proceeds by considering path m_1 which is properly working. Knowledge of the outcome of m_1 allows us to assess the status of the nodes v_i, with $i = 1, \ldots, 5$, as working. As a consequence these nodes are all pruned, and can be removed by all the non working paths included in $\mathcal{T}$. Due to the pruning of v_4 and v_3 the length of the already monitored path m_2 reduces to 1 in the logical representation of the network graph with pruned components, which implies $|\hat{m}_2^{(\mathcal{T})}| = 1$. Hence, $m_2^{(\mathcal{T})}$ turns to be a failing path of one only node, v_6, whose state must be failed, by exclusion.

Finally, we notice also that it never happens that working nodes are discovered by

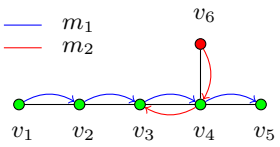


Figure 4.3.3. Identification by exclusion

exclusion as they must belong to working paths, in which case they would have already been pruned[1].

Information obtained by monitoring non-working paths: When probing paths fail, we also have relevant information on the network status. A failing path corresponds to a subset of nodes containing at least a failed node. When a path failure occurs, the nodes of the path must undergo additional monitoring, i.e. probing intersecting paths, to obtain precise failure identification. Indeed, short failed paths allow to localize node failures more precisely than long ones. Moreover, finding failed nodes, or set of nodes containing at least a failure, suggests not to probe paths that, containing at least a failed node, will certainly fail if probed. In the example of Figure 4.3.4 we consider a scenario in which, after some probing activity (not shown in the figure), v_5 is found to be properly functioning. Nevertheless, by observing the failure of monitoring along path m_1 we assess the failure of node v_4 This suggests the removal of path m_3 from the set of monitoring paths, as its failure can be deduced from the failure of the included nodes v_4 which is known to be broken. As a consequence, whenever a monitoring path m_i fails, the monitoring problem can be simplified by removing all the paths m including the entire set of nodes of the failed path m_i, i.e. the set $\{m_j | \hat{m}_j \supseteq \hat{m}_i\}$. Additionally, we note that after pruning nodes, we may end up with some degenerate paths with cardinality one. If this occurs, the probing of these paths gives direct information on the node states, both if the path works and if it does not. While this situation is already considered in the utility of working paths, to take it into account also for the case of a non working path m_i, we consider a further information utility component, in the form of an additive term $\lfloor 1/|\hat{m}_i^{(\mathcal{T})}| \rfloor$ which is equal to one only if path $m_i^{(\mathcal{T})}$ traverses one only node, and is zero otherwise.

(a) Original topology **(b)** Pruned paths

Figure 4.3.4. Pruning of monitoring paths

In conclusion, every time we have certainty of the state of a node (either working or broken), we prune the node (if it works) or the paths including it (if it does not work). When a certain path m is probed, if $m^{(\mathcal{T})}$ works, then all of its sub-paths certainly work as well (i.e., paths $m' \, s.t. \, \hat{m}'^{(\mathcal{T})} \subseteq \hat{m}^{(\mathcal{T})}$). If $m^{(\mathcal{T})}$ fails, then all of its super-paths are failing, too (i.e., paths $m' \, s.t. \, \hat{m}^{(\mathcal{T})} \subseteq \hat{m}'^{(\mathcal{T})}$). When the status of non tested paths can be assessed with certainty due to the described pruning actions on the logical graph, we do not consider them for successive probes, and the set of available actions $\mathcal{A}$ is updated consequently.

In summary, if we make decision a_i corresponding to monitoring path m_i, the information

[1]Discovery of working node by exclusion may instead happen if there is knowledge of the number of failed nodes, which is not considered here.

utility is proportional to $|\hat{m}_i^{(\mathcal{T})}|+|\mathcal{F}_1^{(m_i,\mathcal{T})}|$ if the path m_i works, and to $\lfloor 1/|\hat{m}_i^{(\mathcal{T})}|\rfloor$ otherwise. We can then formulate the information utility function, for each decision $a_i \in \mathcal{A}$ as follows:

$$\lambda(a_i|Z_i) = \begin{cases} |\hat{m}_i^{(\mathcal{T})}| + |\mathcal{F}_1^{(m_i,\mathcal{T})}| & \text{if } Z_i = 1 \\ \left\lfloor \frac{1}{|\hat{m}_i^{(\mathcal{T})}|} \right\rfloor & \text{if } Z_i = 0 \end{cases} \tag{4.1}$$

Correspondingly, we calculate the following *expectation of conditional utility given the observation*:

$$\mathcal{U}(a_i|O_{\mathcal{T}}) = \lambda(a_i|Z_i)P(Z_i|O_{\mathcal{T}}) + \lambda(a_i|\bar{Z}_i)P(\bar{Z}_i|O_{\mathcal{T}}) \tag{4.2}$$

As the available paths may give a different contribution to the identification task, some of them may become redundant, depending on the probing order, which brings our attention to determine an efficient progressive monitoring policy, i.e. to solve the PMP problem. Formally, a PMP policy is a sequence of monitoring actions of $\mathcal{A}$. In the following we aim at defining a PMP policy which maximizes the number of nodes whose state is identified, i.e. the utility defined above.

4.4 Stochastic optimization of PMP

We consider a decision process, in the discrete time, which may end when one of the following conditions occurs:

- Every node status is known
- There are no more paths to monitor (each of the remaining path cannot add any information on the node states)
- The maximum number of probing steps has been reached.

At each step, the process may make one of the decisions in $\mathcal{A}$, whose utility depends on the outcome of the related monitoring path. The number of steps before termination is uncertain. An upper bound is given by the number of monitoring paths N. We recall that we do not assume symmetric routing, i.e. the upper bound on the number of monitoring paths is given by the number of ordered pairs of monitoring nodes $N = V_M \cdot (V_M - 1)$.

Considering the discussion made in Section 4.3.1, we formulate the failure identification problem in terms of *stochastic optimization*. At the n-th step, the state of the decision process is given by the set $O_{\mathcal{T}}^{(n)}$, which reflects the observations made until step n, provides the current knowledge of the status of network components at step n and determines the future action utility values, according to Equation 4.2.

As actions cannot be repeated in consecutive monitoring steps, we denote the actions available at step n, $\mathcal{A}(O_{\mathcal{T}}^{(n)})$, shortly as follows: $\mathcal{A}^{(n)} \triangleq \mathcal{A}(O_{\mathcal{T}}^{(n)}) = \{a_i : m_i \in \mathcal{R}^{(n)}\}$, where $\mathcal{R}^{(n)}$ is the set of monitoring paths which have not been tested yet at the n-th step.

We seek a decision policy that maximizes the expected sum of the utilities incurred by its decisions. The optimal decision policy depends on the utilities, on the number of steps taken to assess the state of the network, and our confidence that obtaining such information through monitoring is actually possible.

Let $V(O_{\mathcal{T}}^{(n)}, n)$ denote the expected information (utility) that will be obtained by the optimal decision policy (e.g. nodes still to be assessed), starting from the observation $O_{\mathcal{T}}^{(n)}$ at step n. If we choose action a_i the expected gain at the following step is given by Equation 4.2. Now, let $E_U(O_{\mathcal{T}}^{(n)}, n)$ be the optimal remaining utility, after step $n+1$, given a monitoring decision in state $O_{\mathcal{T}}^{(n)}$. $E_U(O_{\mathcal{T}}^{(n)}, n)$ describes the optimal decision policy utility after step $n+1$ and so it is stated in terms of $V(O_{\mathcal{T}}^{(n+1)}, n+1)$. In particular, given that the monitoring

path selected at step n is the one pointed by action $a^{(n)} = a_i^*$, corresponding to path m_{i^*}, we have

$$E_U(O_{\mathcal{T}}^{(n)}, n|a_i^*) = P(Z_{i^*}) \cdot V(O_{\mathcal{T}}^{(n)} \cup \{Z_{i^*} = 1\}, n+1) + \\ + P(\bar{Z}_{i^*}) \cdot V(O_{\mathcal{T}}^{(n)} \cup \{Z_{i^*} = 0\}, n+1).$$

By the principle of optimality (Bellman equation) we have the following:

$$V(O_{\mathcal{T}}^{(n)}, n) = \\ = \max_{a_i \in \mathcal{A}^{(n)}} \left\{ P(Z_{i^*}) \cdot \left(|\hat{m}_i^{(\mathcal{T})}| + |\mathcal{F}_1^{(m_i, \mathcal{T})(n)}| + V(O_{\mathcal{T}}^{(n)} \cup \{Z_{i^*}\}, n+1)) \right) + \right. \\ \left. + P(\bar{Z}_{i^*}) \cdot \left(\left\lfloor \frac{1}{|\hat{m}_i^{(\mathcal{T})}|} \right\rfloor + V(O_{\mathcal{T}}^{(n)} \cup \{\bar{Z}_{i^*}\}, n+1)) \right) \right\}.$$

While the equation suggests the use of a dynamic programming approach, over a finite horizon, to solve the PMP problem, we underline the following challenges. (1) The computational complexity in the calculation of the posterior probability $P(Z_i|O_{\mathcal{T}})$ is exponential in the number of paths. (2) There is the well known curse of dimensionality in the representation of the state space of the process, which needs to take account of the outcome of each monitored path. In fact, we note that it is not sufficient to represent the state with the vector of the identified node status, because of the possibility to have delayed assessment of broken nodes, that must be considered to properly calculate the utility terms expressed by Equation 4.2 for the case of working paths. Therefore non working monitored paths must be part of the state representation. Working monitored paths must also be stored in the state representation, to determine the available decisions. Hence the state space of the process is exponential in the number of paths.

We will devote the next sections to polynomial approaches to the design of efficient PMP policies and to metrics to quantitatively measure such efficiency.

4.4.1 The PoPGreedy approach

A *Bayesian greedy strategy* to monitoring path selection and probing is one that progressively selects the next path based on the current *utility maximization rule* and updates the overall observation for the next step. Initially (at step 0) $O_{\mathcal{T}} = \emptyset$, therefore the calculation of the initial action utility is based on prior probabilities as follows:

$$\mathcal{U}^{(0)}(a_i) = \lambda(a_i)P(Z_i) + \lambda(a_i)P(\bar{Z}_i). \tag{4.3}$$

Hence, at step 0, the Bayesian strategy consists in selecting the action that maximizes the utility based on prior knowledge:

$$a^{(1)} = \arg\max_{a_i \in \mathcal{A}} \mathcal{U}^{(0)}(a_i).$$

Anytime a new path is monitored, it produces an outcome which requires the update of the current estimate of path failure probabilities.

At step $n+1$ the Bayesian strategy selects the action $a^{(n+1)}$ that maximizes the expectation of the utility given the current observation:

$$a^{(n+1)} = \arg\max_{a_i \in \mathcal{A}^{(n)}} \mathcal{U}^{(n)}(a_i|O_{\mathcal{T}}^{(n)})$$

The testing procedure is described in Algorithm 2. Given a graph G representing the network topology, a set of paths $\mathcal{M}$ and a prior probability of node failure p, the algorithm returns the posterior probability of failure of all nodes as is obtained after probing at most K paths in $\mathcal{M}$ and a related ranking.

At each iteration the algorithm selects the path m with maximum expected utility (ties are broken by considering a priority based on the path index, i.e. if m_i and m_j, with $i < j$ have the same utility, the algorithm selects m_i). Depending on the outcome of the test, either the set of failed paths (if m failed) or the set of working nodes (otherwise) is updated, together with actions corresponding to testing non visited super-paths and sub-paths of a failing/working paths, respectively (**lines** 17 and 21). We refer to this approach as to PoPGreedy (Posterior Probability Greedy), and detail it in Algorithm 2.

Algorithm 2 PoPGreedy

Input: $G = (V, E)$: graph representing a network topology.
$\mathcal{M}$: set of walkable paths in the network.
p: initial probability of node failures.
K: maximum number of path probes.
Output: sorted sequence of nodes depending on their probability of failure: $\boldsymbol{V_f}$.

1: $W = \emptyset$ (set of working nodes)
2: $\mathcal{T} = \emptyset$ (set of tested paths)
3: $\mathcal{F} = \emptyset, \mathcal{F} \subseteq \mathcal{T}$ (set of failed paths)
4: $\mathcal{R} = \mathcal{M} \setminus \mathcal{T}$ (set of non visited paths)
5: $\mathcal{A}^{(0)} = \{a_1, \ldots, a_{|\mathcal{M}|}\}$
6: $O_{\mathcal{T}} = \emptyset$
7: **for** $i = 1, \ldots,$ K **do**
8: $\quad a^{(i)} = \arg \max_{a \in \mathcal{A}^{(i-1)}} \mathcal{U}(a|O_{\mathcal{T}})$
9: $\quad$ **if** $\mathcal{U}(a^{(i)}|O_{\mathcal{T}}) = 0$ **then**
10: $\quad\quad$ return list of nodes sorted by $P(\bar{S}_v|O_{\mathcal{T}})$, $\boldsymbol{V_f}$
11: $\quad \mathcal{T} \leftarrow \mathcal{T} \cup \{m_i\}$
12: $\quad \mathcal{R} \leftarrow \mathcal{R} \setminus \{m_i\}$
13: $\quad$ ***test*** $\boldsymbol{m_i}$
14: $\quad$ **if** m_i fails **then**
15: $\quad\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{m_i\}$
16: $\quad\quad O_{\mathcal{T}} = O_{\mathcal{T}} \cup (\bar{Z}_i)$
17: $\quad\quad \mathcal{A}^{(i)} \leftarrow \mathcal{A}^{(i-1)} \setminus (\{a_i\} \cup \{a_j \ : \ \hat{m}_i^{(\mathcal{T})} \subseteq \hat{m}_j^{(\mathcal{T})}\})$
18: $\quad$ **else**
19: $\quad\quad W \leftarrow W \cup \{\hat{m}_i\}$
20: $\quad\quad O_{\mathcal{T}} = O_{\mathcal{T}} \cup (Z_i)$
21: $\quad\quad \mathcal{A}^{(i)} \leftarrow \mathcal{A}^{(i-1)} \setminus (\{a_i\} \cup \{a_j \ : \ \hat{m}_j^{(\mathcal{T})} \subseteq \hat{m}_i^{(\mathcal{T})}\})$

An example of execution of PoPGreedy: We show an example of execution on the network represented in Figure 4.4.1. We assume priori probability $p = 0.1$ and let node v_9 be the only failed node in the network. Nodes v_1, v_2, v_3 and v_4 are monitors, and consider undirected paths. We consider the 6 monitoring paths shown in the figure.

- *Step 1:* The paths that maximize utility at the first iteration are m_3, m_5 and m_6. For the tie breaking rule we choose path m_3. The path works. Hence $\mathcal{R} = \mathcal{M} \setminus \{m_3\}$, $O^{\mathcal{T}} = \{Z_3\}$ and the set of working nodes $W = \{1, 4, 7, 8\}$. It results that $a_4 = \arg\max_{a \in \mathcal{A}} u(a|O_{\mathcal{T}})$, $u(a_4|O_{\mathcal{T}}) = 2.187$.
- *Step 2:* Test m_4, $\mathcal{R} = \mathcal{R} \setminus \{m_4\}$. The path fails, therefore $\mathcal{F} = \{m_4\}$ and $O_{\mathcal{T}} = O_{\mathcal{T}} \cup \bar{Z}_4$. It holds that $a_1 = \arg\max_{a \in \mathcal{A}}$, with $u(a_1|O_{\mathcal{T}}) = 1.1358$.

- *Step 3:* Test m_1, $\mathcal{R} = \mathcal{R} \setminus \{m_1\}$. The path works, hence: $W = W \cup \hat{m}_1^{(\mathcal{T})}$ and $O_{\mathcal{T}} = O_{\mathcal{T}} \cup Z_1$. At this point, it results that $u(a_2|O_{\mathcal{T}}) = 1.278$, while $u(a_6|O_{\mathcal{T}}) = 0$. By knowing that m_4 failed while node v_2 works, we can claim with certainty that m_6 will also fail, as it steps onto all the remaining nodes of m_4.
- *Step 4:* Test m_2, $\mathcal{R} = \mathcal{R} \setminus \{m_2\}$. The path works, hence: $W = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $O_{\mathcal{T}} = O_{\mathcal{T}} \cup Z_2$. The utility of the two non visited paths, m_5 and m_6 is zero, therefore the execution is over. The algorithm returns the failure probabilities: $P(\bar{S}_9|O_{\mathcal{T}}) = 1$, $P(\bar{S}_{10}|O_{\mathcal{T}}) = 0.1$ and $P(\bar{S}_v|O_{\mathcal{T}}) = 0$ for all the other nodes.

It must be noted that, although the algorithm leaves some uncertainty on the state assessment of node v_{10} this is due to the impossibility of obtaining certain status identification for v_{10} with the available paths. None of the existing paths can disambiguate the status of such a node as the only paths traversing it fail because of the failure of node v_9.

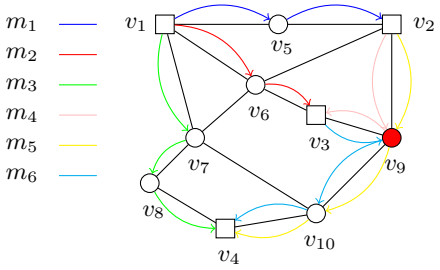


Figure 4.4.1. Example topology with a failure in v_9

4.4.2 Optimality approximation

The definition of utility of an action given prior observations (Equation 4.1) can be extended in order to characterize the utility of a set of actions (corresponding to the utility of probing a set of distinct paths, $\mathcal{T}$) as follows:

$$\lambda(\{a^{(1)}, \ldots, a^{(|\mathcal{T}|)}\}, O_{\mathcal{T}}) = \left| \bigcup_{m_i \in \mathcal{T} \setminus \mathcal{F}} \hat{m}_i \right| + \left| \bigcup_{m_i \in \mathcal{T} \setminus \mathcal{F}} \mathcal{F}_1^{(m,\mathcal{T})} \right| + F_{\mathcal{T}},$$

where $F_{\mathcal{T}} := \{w \in V : P(\bar{S}_w|O_{\mathcal{T}}) = 1\}$ is the set of detected failed nodes by probing paths $\mathcal{T}$. This definition allows us to formulate the problem of assessing the maximum number of node states with K path probes as a formal maximization problem:

$$\begin{aligned} &\max_{\mathcal{A}^{(\mathcal{T})} \subseteq \mathcal{A}} \lambda(\mathcal{T}|O_{\mathcal{T}}) \\ &s.t. \quad |\mathcal{T}| \leq K \end{aligned} \tag{4.4}$$

where $\mathcal{A}^{(\mathcal{T})} = \{a^{(1)}, \ldots, a^{(|\mathcal{T}|)}\}$. Constant approximations for deterministic optimization problems where the objective function has properties of monotonicity and submodularity were proved in [104]. More recently, the concept of *adaptive* monotonicity and submodularity, originally introduced in [60] and lately revised in [53], extended such properties to the context of stochastic optimization problems, that is where our scenario belongs. In a stochastic maximization problem, the function to be maximized depends on a set of observations $O_{\mathcal{T}}$ on the state of the elements of the ground set, in our case, the state of paths in $\mathcal{M}$. In this context, greedy policies choose at each step the action that maximizes the expected value of the utility, that is known with certainty only after tests take place. Notice that PoPGreedy

in Algorithm 2 follows the Adaptive Greedy Algorithm structure shown in [60]. Before reporting the definitions of adaptive monotonocity and submodularity, we give definition of conditional expected marginal benefit ([60]). In the following definitions, X is a finite set of elements.

Definition 4.4.1. *Let $Y \subset X$ and let $x \in X \setminus Y$. The* conditional expected marginal benefit *of x with respect of a function f having observed O_Y is:*

$$\Delta(x|O_Y) := \mathbb{E}[f(Y \cup \{x\}, O_Y) - f(Y, O_Y)], \tag{4.5}$$

where by O_Y we mean the restriction of the observations O_X to the subset Y.

Definition 4.4.2. *A function $f : 2^X \times O_X \to \mathbb{R}$ is* adaptive monotone *if $\forall x \in X$ and $\forall Y \subseteq X$ it holds that $\Delta(x|O_Y) \geq 0$.*

Definition 4.4.3. *A function $f : 2^X \times O_X \to \mathbb{R}$ is* adaptive submodular *if $\forall Z \subseteq Y \subset X$ and $\forall x \in X \setminus Y$ we have $\Delta(x|O_Z) \geq \Delta(x|O_Y)$.*

In our scenario, the ground set X is the set of all possible actions $\mathcal{A}$ on paths $\mathcal{M}$, and the state of a path is either normal or defective (or equivalently, 1 or 0). Paths' states can be assessed only through observations on path probes. Observe that the definition of conditional expected marginal benefit corresponds to the definition of expected utility given in Equation 4.2:

$$\begin{aligned}
&\Delta(a|O_{\mathcal{T}}) = \\
&\left(\left|\hat{m} \cup \bigcup_{m_i \in \mathcal{T}\setminus\mathcal{F}} \hat{m}_i\right| + \left|\mathcal{F}_1^{(m,\mathcal{T})} \cup \bigcup_{m_i \in \mathcal{T}\setminus\mathcal{F}} \mathcal{F}_1^{(m_i,\mathcal{T})}\right| + |F_{\mathcal{T}}|\right) P(Z|O_{\mathcal{T}}) \\
&+ \left(\left|\bigcup_{m_i \in \mathcal{T}\setminus\mathcal{F}} \hat{m}_i\right| + \left|\bigcup_{m_i \in \mathcal{T}\setminus\mathcal{F}} \mathcal{F}_1^{(m,\mathcal{T})}\right| + |F_{\mathcal{T}}| + \left\lfloor \frac{1}{|\hat{m}^{(\mathcal{T})}|} \right\rfloor\right) P(\bar{Z}|O_{\mathcal{T}}) \\
&- \left|\bigcup_{m_i \in \mathcal{T}\setminus\mathcal{F}} \hat{m}_i\right| - \left|\bigcup_{m_i \in \mathcal{T}\setminus\mathcal{F}} \mathcal{F}_1^{(m,\mathcal{T})}\right| - |F_{\mathcal{T}}| = \\
&= (|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}_1^{(m,\mathcal{T})}|) P(Z|O_{\mathcal{T}}) + \left\lfloor \frac{1}{|\hat{m}^{(\mathcal{T})}|} \right\rfloor P(\bar{Z}|O_{\mathcal{T}}) = \\
&= \mathcal{U}(a_i|O_{\mathcal{T}}).
\end{aligned}$$

For adaptive monotone and submodular functions, solutions achieved by greedy policies are constant approximations to the optimal solutions ([60], [53]). While it is trivial to prove adaptive monotonicity for our utility function, we can exhibit an example (depicted in Figure 4.4.2) showing that λ is not adaptive submdular. Assume $p = 0.1$ is the a priori node failure probability and let $\mathcal{T} = \{m_1\}$ and $\mathcal{T}' = \{m_1, m_2\}$, with $\mathcal{F} = \{m_1, m_2\}$. Then $P(Z_3|O_{\mathcal{T}}) = 1 - \frac{1-(1-p)^2-(1-p)^4+(1-p)^5}{1-(1-p)^4} = 0.638$, which leads to $\mathcal{U}(a_3|O_{\mathcal{T}}) = |\hat{m}_3^{(\mathcal{T})}|P(Z_3|O_{\mathcal{T}}) = 1.2766$, whereas $P(Z_3|O_{\mathcal{T}'}) = 1 - \frac{1-(1-p)^2-2(1-p)^4+2(1-p)^5}{1-2(1-p)^4+(1-p)^5} = 0.789$, and hence $\mathcal{U}(a_3|O_{\mathcal{T}'}) = |\hat{m}_3^{(\mathcal{T}')}|P(Z_3|O_{\mathcal{T}'}) = 1.578$, which is greater than $\mathcal{U}(a_3|O_{\mathcal{T}})$.
When the objective function of a maximization problem is not adaptive submodular, as for our definition of utility, it is still possible to study an approximation of the solution obtained by a greedy policy with respect to the optimal one by bounding the *adaptive submodularity ratio* $\gamma_{O_{\mathcal{T}},k}(\lambda, p)$, with a scalar $\alpha \in (0, 1]$. The resulting approximation is:

$$\lambda_{avg}(\pi^G) \geq \left(1 - \exp\left(-\frac{\alpha K}{h}\right)\right) \lambda_{avg}(\pi^*) \tag{4.6}$$

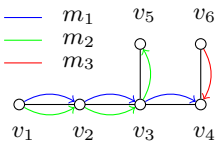


Figure 4.4.2. Example showing non adaptive submodularity of f.

where $\lambda_{avg}(\pi^G/\pi^*)$ are the average quantity of information gained by the greedy and the optimal policies π^G and π^*, respectively. Parameters K and h are the constraint to the maximum number of tests and the height of the decision tree of policy π^*, respectively. This result, together with the definition of adaptive submodular ratio, was recently proposed in [57].

Upper-bound to adaptive submodularity ratio

The goal of this section is to exhibit a scalar $\alpha > 0$ such that

$$\frac{\sum_{m\in\mathcal{M}} P(m \in \mathcal{T}^\pi)\Delta(a|O_\mathcal{T})}{\Delta(\pi|O_\mathcal{T})} \geq \alpha. \tag{4.7}$$

The adaptive submodularity ratio is upperbounded by 1 and it is equal to 1 if and only if λ is adaptive submodular. Here $\mathcal{T}^\pi$ is the set of paths chosen by a policy π, whereas $O_\mathcal{T}$ is a set of partial observations over a set of path $\mathcal{T}$. It holds that:

$$\frac{\sum_{m\in\mathcal{M}} P(m \in \mathcal{T}^\pi)\Delta(a|O_\mathcal{T})}{\Delta(\pi|O_\mathcal{T})} \geq \alpha \Rightarrow \tag{4.8}$$

$$\frac{\sum_{m\in\mathcal{M}} P(m \in \mathcal{T}^\pi)\Delta(a|O_\mathcal{T})}{\sum_{m\in\mathcal{M}} P(m \in \mathcal{T}^\pi)\Delta(a|O_{\mathcal{T}'})} \geq \alpha \tag{4.9}$$

where $O_{\mathcal{T}'}$ is the set of observations such that the next path chosen by policy π is m, and $\mathcal{T} \subset \mathcal{T}'$. From the discussion in [57], it holds that the inequality 4.8 can be equivalently expressed as follows:

$$\sum_{m\in\mathcal{M}} P(m \in \mathcal{T}^\pi)\Delta(a|O_\mathcal{T}) \geq \alpha \sum_{m\in\mathcal{M}} P(m \in \mathcal{T}^\pi)\Delta(a|O_{\mathcal{T}'})$$

$$\sum_{m\in\mathcal{M}} P(m \in \mathcal{T}^\pi) d_a(\alpha) \geq 0, \tag{4.10}$$

where $d_a(\alpha) = \Delta(a|O_\mathcal{T}) - \alpha\Delta(a|O_{\mathcal{T}'}) \equiv \mathcal{U}(a|O_\mathcal{T}) - \alpha\mathcal{U}(a|O_{\mathcal{T}'})$. To prove the previous relation, we want to show that for every $a \in \mathcal{A}$ (action corresponding to probing path m such that $m \notin \mathcal{T}'$) it holds that $d_a(\alpha) \geq 0$. Among all path choices, we just need to study the contribution of those such that $d_a(1) < 0$. Therefore, we exclude all actions corresponding to the following sets of paths from our analysis: *i)* all paths $m \in \mathcal{T}'$ that were already tested; *ii)* all paths for which it holds that $P(Z|O_{\mathcal{T}'}) = 0$ or 1 (as in such case, $d_a(\alpha) = \mathcal{U}(a|O_\mathcal{T}) \geq 0$); *iii)* all paths such that $P(Z|O_\mathcal{T}) = 0$ or 1, as this implies $P(Z|O_{\mathcal{T}'}) = 0$ or 1 respectively, and therefore $d_a(1) = 0$; *iv)* paths m such that $\mathcal{U}(a|O_\mathcal{T}) \geq \mathcal{U}(a|O_{\mathcal{T}'})$. For all the listed cases, $d_a(1) \geq 0$. We study the maximum difference (i.e., the maximum value of $|d_a(\alpha)|$ with $d_a(\alpha) < 0$) that may occur between $\mathcal{U}(a|O_\mathcal{T})$ and $\mathcal{U}(a|O'_\mathcal{T})$ for all other paths.

In order to accomplish to this task, we need to study the smallest non-zero value of $\mathcal{U}(a|O_\mathcal{T})$, Δ_{min}, and the greatest value of $\mathcal{U}(a|O_{\mathcal{T}'})$, Δ'_{max}. By choosing $\alpha = \frac{\Delta_{min}}{\Delta'_{max}}$, the relation in Equation 4.10 holds always.

Smallest value of $\mathcal{U}(a|O_\mathcal{T})$ $P(Z|O_\mathcal{T})$ is positive and minimum when every node v of m is traversed by failing paths of length 2. This is because the probability of failure of a node v is directly proportional to the number of failing paths traversing it and inversely proportional to the number of nodes traversed by such paths. Nevertheless, observe that if even just one of such paths had length 1 (i.e., it would only pass through a node of m), then the probability of failure of m would be 1 and its utility would be 0; this situation would fall into the set *iii.* of paths that we exclude from this analysis. Hence, when a node is traversed only by failing paths of length 2, its failure probability is maximal and therefore its working probability is minimal, excluding the case where $P(Z|O_\mathcal{T}) = 0$. We call P_{min} such value of $P(Z|O_\mathcal{T})$. In this case, $\mathcal{U}(a|O_\mathcal{T}) = (|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}_1^{(m,\mathcal{T})}|)P_{min}$. Notice that this expression exhibits explicit growing dependency of $\mathcal{U}(a|O_\mathcal{T})$ on $|\mathcal{F}_1^{(m,\mathcal{T})}|$. In Appendix 4.A we show that $(|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}_1^{(m,\mathcal{T})}|)P_{min}$ is indeed the smallest value of $\mathcal{U}(a|O_\mathcal{T})$ subject to $P(Z|O_\mathcal{T}) \in (0,1)$ despite the presence of the term $|\mathcal{F}_1^{(m_i,\mathcal{T})}|$. Let us study now how $\mathcal{U}(a|O_\mathcal{T})$ changes with respect to $|\hat{m}|$. Note that $\mathcal{U}(a|O_\mathcal{T})$ grows linearly with $|\hat{m}|$ if $P(Z|O_\mathcal{T})$ is fixed. Nevertheless, if we consider the case where every node of $m^{(\mathcal{T})}$ is traversed by some failing paths of length 2, then the contribution given by each node to the decrease of $P(Z|O_\mathcal{T})$ is greater than adding 1 to $|\hat{m}|$, i.e., one such node would contribute to the exponential decrease of $P(Z|O_\mathcal{T})$, while it would make $|\hat{m}|$ increase just by 1. This consideration emerges explicitly in the equations that follow. The path working probability $P(Z|O_\mathcal{T}) = P_{min}$ is equal to:

$$
\begin{aligned}
P_{min} &= \prod_{v\in\hat{m}} P(S_v|O_\mathcal{T}) = \\
&= \prod_{v\in\hat{m}} 1 - \frac{p}{1 - \sum_{i=2}^{\partial_v+1} (-1)^i (1-p)^i \binom{\partial_v}{i-1}} \geq \\
&\geq \left[1 - \frac{p}{1 - \sum_{i=2}^{\partial_{max}+1} (-1)^i (1-p)^i \binom{\partial_{max}}{i-1}}\right]^{|\hat{m}|} \geq \\
&\geq \left[1 - \frac{p}{1 - \sum_{i=2}^{\partial_{max}+1} (-1)^i (1-p)^i \binom{\partial_{max}}{i-1}}\right]^{|\hat{m}_{max}|}
\end{aligned}
$$

where ∂_v is the number of failing paths traversing v, ∂_{max} is the biggest among them and $|\hat{m}_{max}|$ is the length of the longest path. Notice that $\partial_{max} \leq |\mathcal{F}_1^{(m,\mathcal{T})}|$. The denominator appearing in the last expression can be written as follows:

$$
\begin{aligned}
&1 - \sum_{i=2}^{\partial_{max}+1} (-1)^i (1-p)^i \binom{\partial_{max}}{i-1} = \\
&1 + (1-p) \cdot \sum_{i=1}^{\partial_{max}} (-1)^i (1-p)^i \binom{\partial_{max}}{i} = \\
&1 + (1-p) \cdot \sum_{i=1}^{\partial_{max}} (p-1)^i \binom{\partial_{max}}{i} = \quad \text{[Newton's binomial]} \\
&1 + (1+p) \cdot [(p-1+1)^{\partial_{max}} - 1] = \\
&1 + (1-p)(p^{\partial_{max}} - 1).
\end{aligned}
\tag{4.11}
$$

Therefore the minimum value of $\mathcal{U}(a|O_\mathcal{T})$, Δ_{min} is:

$$\begin{aligned}\Delta_{min} &= (|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}_1^{(m,\mathcal{T})}|) \prod_{v \in \hat{m}} 1 - \frac{p}{1 + (1-p)(p^{\partial_v} - 1)} \\ &\geq (|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}_1^{(m,\mathcal{T})}|) \left[1 - \frac{p}{1 + (1-p)(p^{\partial_{max}} - 1)}\right]^{|\hat{m}_{max}|}.\end{aligned} \tag{4.12}$$

Notice that we are excluding from our analysis the case $|\hat{m}^{(\mathcal{T})}| = 1$, that is the only situation in which the second term of the utility $\left\lfloor \frac{1}{|\hat{m}^{(\mathcal{T})}|} \right\rfloor$ is non zero: as a matter of fact, Equation 4.12 proves that the larger the path length, appearing as the exponent of P_{min}, the smaller is Δ_{min}. To recap, the smallest value of $\mathcal{U}(a|O_{\mathcal{T}})$ is achieved when path $m^{(\mathcal{T})}$ is long and all of its nodes are traversed by failing paths of length 2.

Greatest value of $\mathcal{U}(a|O_{\mathcal{T}'})$ From the situation described above, we can analyse what is the greatest value of $\mathcal{U}(a|O_{\mathcal{T}'})$. First of all, observe that $\forall \mathcal{T}, \mathcal{T}'$ such that $\mathcal{T} \subset \mathcal{T}'$ it holds that $|\hat{m}^{(\mathcal{T})}| \geq |\hat{m}^{(\mathcal{T}')}|$. As a consequence, it holds that $\mathcal{U}(a|O_{\mathcal{T}'}) > \mathcal{U}(a|O_{\mathcal{T}})$ if and only if $P(Z|O_{\mathcal{T}'})$ is sufficiently larger than $P(Z|O_{\mathcal{T}})$. In general, $P(Z|O_{\mathcal{T}'}) > P(Z|O_{\mathcal{T}})$ with $O_{\mathcal{T}} \subset O_{\mathcal{T}'}$ in two occasions: either because of the presence of some functioning paths in $\mathcal{T}' \setminus \mathcal{T}$ traversing m, or because it was possible to localize failures of paths traversing nodes of m on some other nodes. In the first case it results that $|\hat{m}^{(O_{\mathcal{T}'})}| < |\hat{m}^{(O_{\mathcal{T}})}|$, therefore the increase of $P(Z|O_{\mathcal{T}'})$ could be contrasted by the decrease of $|\hat{m}^{(O_{\mathcal{T}'})}|$, possibly resulting in $\mathcal{U}(a|O_{\mathcal{T}}) > \mathcal{U}(a|O_{\mathcal{T}'})$. In the second case instead, it holds that $|\hat{m}^{(\mathcal{T})}| = |\hat{m}^{(\mathcal{T}')}|$. Hybrid situations may occur, too. We shall first consider the second case: assume it was possible to assess as "failed" all nodes not in $\hat{m}$ appearing in the two-length paths traversing m. Therefore, $\mathcal{U}(a|O_{\mathcal{T}'})$ becomes equal to $|\hat{m}^{(\mathcal{T}')}|(1-p)^{|\hat{m}^{(\mathcal{T}')}|}$. Notice that this then this the initial expected value of the utility function of a, when no observations were made. As a matter of fact, the working probability of a node only grows when a working path traverses it (and in such case it becomes 1). Now we can also consider the case in which $|m^{(\mathcal{T}')}|$ is reduced if some working path m' partially covering nodes of m was tested. Assuming $m^{(\mathcal{T})}$ is long enough, we want to analyse the growing trend of $|\hat{m}^{(\mathcal{T}')}|(1-p)^{|\hat{m}^{(\mathcal{T}')}|}$. Indeed, the first term of this expression $|\hat{m}^{(\mathcal{T}')}|$ trivially grows linearly, whereas $(1-p)^{|\hat{m}^{(\mathcal{T}')}|}$ decreases with $|\hat{m}^{(\mathcal{T}')}|$. The trend of their products depends on the value of p. Excluding the trivial cases where $p = 0$ or 1, it is easy to prove analytically that the maximum value of $|\hat{m}^{(\mathcal{T}')}|(1-p)^{|\hat{m}^{(\mathcal{T}')}|}$ is for $|\hat{m}^{(\mathcal{T}')}| = -\left[\frac{1}{\ln(1-p)}\right]$. Here $n = [x]$ is the rounded natural value of x. Therefore the maximum value of $\mathcal{U}(a|O_{\mathcal{T}'})$ is $\Delta'_{max} = -\left[\frac{1}{\ln(1-p)}\right](1-p)^{-\left[\frac{1}{\ln(1-p)}\right]}$. Notice that the maximum value of $\mathcal{U}(a|O_{\mathcal{T}'})$ is reached when $\mathcal{F}_1^{(m,\mathcal{T})} = \emptyset$. Indeed, in case $\mathcal{F}_1^{(m,\mathcal{T})} \neq \emptyset$, the working probability of m would decrease exponentially, at the face of a linear growth of the deterministic multiplier $(|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}_1^{(m,\mathcal{T})}|)$, as explained in Appendix 4.A.

Solution approximation By choosing $\alpha = \frac{\Delta_{min}}{\Delta'_{max}}$ as discussed in the previous sections, it holds that for all paths m and observations $O_{\mathcal{T}}$ and $O_{\mathcal{T}'}$, $d_m^{\alpha} \geq 0$, implying soundness of Equation 4.7. Given a lower bound to the adaptive submodularity ratio, we may use the result shown in [57] to claim the following:

Proposition 4.4.1. *If π^G is the policy representing the adaptive greedy algorithm using K steps, and $\lambda : 2^{\mathcal{M}} \times O_{\mathcal{M}} \to \mathbb{R}_{\geq 0}$ is the utility function defined in Equation 4.4, then:*

$$\lambda_{avg}(\pi^G) \geq \left(1 - \exp\left(-\frac{\alpha K}{k}\right)\right) \lambda_{avg}(\pi^*)$$

where π^* is the optimal policy, k is the number of steps that π^* takes to reach convergence and $\alpha = \frac{\Delta_{min}}{\Delta'_{max}}$.

Proof. The statement is a direct consequence of the following facts: *i.* λ is adaptive monotone. *ii.* Theorem 1 in [57], reported in Equation 4.6. *iii.* PoPGreedy is an Adaptive Greedy Algorithm. □

Notice that α is dependent on controllable parameters ∂_{max} and $|\hat{m}_{max}|$ that do not depend on the network topology but only on the routing paths choice.
When PoPGreedy is run on the simple example shown in Figure 4.4.1, only in two occasions it happens that $\mathcal{U}(a|O_{\mathcal{T}}) < \mathcal{U}(a|O_{\mathcal{T}'})$. The one marking maximum difference holds for $a = a_5$, $\mathcal{T} = \{m_3, m_4\}$ and $\mathcal{T}' = \{m_1, m_3, m_4\}$, where $\mathcal{U}(a_5|O_{\mathcal{T}}) = 1.076$ and $\mathcal{U}(a^{(5)}|O_{\mathcal{T}'}) = 1.278$. For this example, $\alpha = 0.842$.

4.4.3 Computational Complexity

Theorem 4.4.1. *The computational complexity of PoPGreedy (Algorithm 2) is* $O(K \cdot |\mathcal{M}| \cdot 2^{|\mathcal{F}|})$*, where K is the maximum number of path probes.*

Proof. At each of the $O(\mathrm{K})$ steps of the algorithm, expected utilities are updated (**line** 8). This operations requires computing $P(\bar{Z}|O_{\mathcal{T}})$ for all paths that are not tested yet:

$$P(\bar{Z}|O_{\mathcal{T}}) = P(\bar{Z} \wedge O_{\mathcal{T}})/P(O_{\mathcal{T}}). \tag{4.13}$$

Observe that, when computing the joint probability of the outcomes of previously tested paths, the contribution given by working paths simply results in pruning working nodes from non working paths. Therefore, the joint probabilities in equation (4.13) may be computed as follows:

$$P(O_{\mathcal{T}}) = P\left(\bigwedge\nolimits_{m \in \mathcal{F}} \bar{Z}^{o}_{m^{(\mathcal{T})}}\right). \tag{4.14}$$

The expression in the previous equation requires a number of addends that is exponential in the number of failed paths ($2^{|\mathcal{F}|}$). Computing node failure probabilities (**line** 10) requires the same number of operations. The final cost is $O(\mathrm{K} \cdot |\mathcal{M}| \cdot 2^{|\mathcal{F}|})$. □

Even considering sporadic failures, it is hard to predict how much the exponential factor may grow. Even within the same topology, the time required for computing the joint probability $P(O_{\mathcal{T}})$ is highly dependent on where failures occur: if highly connected nodes fail, the number of failed paths may be big, which makes the computation of the failure probabilities extremely time consuming.
The above reasoning motivates the use of polynomially computable metrics to approximate the nodes' conditioned failure probabilities. In the next section, we define a polynomially computable centrality metric that captures the trend of how node failure probabilities are influenced when conditioned by iterative observations on test outcomes.

4.5 Failure centrality

We hereby define the *failure centrality* of a node v given the observation $O_{\mathcal{T}}$.

Definition 4.5.1. *The* failure centrality *of a node v given the observation* $O_{\mathcal{T}}$ *is* $c(v|O_{\mathcal{T}}) = 0$ *if v is traversed by some working paths in* $\mathcal{T}$*, it is equal to the prior probability of failure if v is not traversed by any path in* $\mathcal{T}$*, otherwise* $c(v|O_{\mathcal{T}}) = \max\{T_1; T_2\}$*, where:*

$$T_1 = \left\lceil \frac{\sum_{m \in \mathcal{M}_v^{(\mathcal{T})} \cap \mathcal{F}} \left\lfloor \frac{1}{|\hat{m}^{(\mathcal{T})}|} \right\rfloor}{|\mathcal{M}_v^{(\mathcal{T})} \cap \mathcal{F}|} \right\rceil, \tag{4.15}$$

$$T_2 = \mathcal{P}_v + H(\lfloor \mathcal{P}_v \rfloor - 1) \cdot \left(1 - \frac{\epsilon}{\mathcal{P}_v} - \mathcal{P}_v\right), \tag{4.16}$$

$$\mathcal{P}_v = \frac{|\mathcal{M}_v^{(\mathcal{T})} \cap \mathcal{F}|}{|\bigcup_{m \in \mathcal{M}_v^{(\mathcal{T})} \cap \mathcal{F}} \hat{m}^{(\mathcal{T})}|}, \tag{4.17}$$

where $\mathcal{M}_v^{(\mathcal{T})}$ is the set of monitoring paths $m^{(\mathcal{T})}$ crossing node v. $H(x)$ is the Heaviside function ($H(x) = 0$, if $x < 0$, and $H(x) = 1$ when $x \geq 0$). $\epsilon > 0$ is a small constant.

Node centrality is used to approximate the value of $P(\bar{S}_v|O_{\mathcal{T}})$ in the calculation of the posterior estimate of the state probability of any path $m_i \in \mathcal{R}$, which is $P(Z_i|O_{\mathcal{T}})$, that may be time consuming. The possible values of $c(v|O_{\mathcal{T}})$ span in the interval $[0, 1]$ and, in analogy with probabilities, $c(v|O_{\mathcal{T}}) = 0$ means that the failure probability of node v is 0, that is, $v \in W$, whereas if $c(v|O_{\mathcal{T}}) = 1$ implies that the node is broken.

In the following we give some observations and proposition to characterize the values of the node failure centrality given the observation.

Observation 4.5.1. *Firstly, observe that $T_1 \in \{0, 1\}$. Indeed, for any failed path it holds that $|\hat{m}^{(\mathcal{T})}| \geq 1$, therefore the maximum value of the numerator in equation (4.15) is $|\mathcal{M}_v \cap \mathcal{F}|$, proving that T_1 can not be greater than 1. When there is at least one path m s.t. $|\hat{m}^{(\mathcal{T})}| = 1$, $T_1 = 1$, otherwise $T_1 = 0$.*

Proposition 4.5.1. *For all nodes v and observations $O_{\mathcal{T}}$ it holds that $0 \leq T_2 < 1$.*

Proof. While it is trivially true that $T_2 \geq 0$, we prove that T_2 cannot be greater than or equal to 1. We observe that if $\mathcal{P}_v < 1$, then $T_2 = \mathcal{P}_v$. When $\mathcal{P}_v = 1$, T_2 becomes $1 - \epsilon$, while if $\mathcal{P}_v > 1$, $T_2 = 1 - \frac{\epsilon}{\mathcal{P}_v}$, that is a monotonically growing function with a horizontal asymptote in 1. □

Proposition 4.5.2. *Let $v \in V$ be a node and $O_{\mathcal{T}}$ the outcome of some path probes. If $c(v|O_{\mathcal{T}}) = 1 \implies v$ is broken.*

Proof. In Proposition 4.5.1 we prove that $T_2 < 1$, hence $c(v) = 1 \iff T_1 = 1$. When there is at least a failed path m traversing v such that $|\hat{m}^{(\mathcal{T})}| = 1$, the numerator *num* of T_1 is $0 < num \leq |\mathcal{M}_v \cap \mathcal{F}|$ and therefore $T_1 = c(v|O_{\mathcal{T}}) = 1$. When this situation occurs, the probability of failure of node v is indeed 1, as this means that the failure of path m is only due to the failure of node v. □

Proposition 4.5.3. *Let $v \in V$ be a k-identifiable node with respect to the set of paths $\mathcal{T}$, where k is the number of failures in the network, and let $O_{\mathcal{T}}$ be the outcomes of path probes on $\mathcal{T}$. If v is broken $\implies c(v|O_{\mathcal{T}}) = 1$.*

Proof. Since v is k-identifiable, this means that the set of paths crossing v is different from the sets of paths crossing any other set of nodes of size at most k. In particular, it is different from the set of paths crossing the other $k - 1$ broken nodes. Hence there must be at least one path m that passes through v and not through any other failed node, and therefore $\hat{m}^{(\mathcal{T})} = \{v\}$. What is left to prove is that there is some set of observations $O_{\mathcal{T}}$ that allows to disambiguate v by finding out that indeed $\hat{m}^{(\mathcal{T})} = \{v\}$. If $|\hat{m}^{(\mathcal{T})}| = 1$, then this is trivially true. Also if $k = 1$, by definition, node v must be traversed by a set of paths different than the set of paths traversing any other node laying in m; therefore there exist some working path that passes through nodes in $\hat{m} \setminus \{v\}$. Otherwise, again from the definition of k-identifiability, the failure of node v must produce different sets of failed paths than the ones resulting from simultaneous failures of v and any other node in $\hat{m} \setminus \{v\}$. As a consequence, there must be some working path passing through the nodes in $\hat{m} \setminus \{v\}$ and not through v, making it possible to verify through end-to-end monitoring measurements that $\hat{m}^{(\mathcal{T})} = \{v\}$, which results in $T_1 = c(v|O_{\mathcal{T}}) = 1$. □

To conclude the discussion on the formulation of the centrality, we comment on the choice of term T_2 in equation (4.16). This formulation is motivated by the observation that node failure probabilities are directly proportional to the number of failed paths traversing a node, and inversely proportional to the number of nodes $w \notin W$ being traversed by such paths. This property is satisfied by both $\mathcal{P}_v$ and $1 - \frac{\epsilon}{\mathcal{P}_v}$. Furthermore, by experimental observations, we noticed that $P(\bar{S}_v|O_\mathcal{T})$ grows steeply with the number of terms $\bar{Z}_i$ (where $v \in \hat{m}_i^{(\mathcal{T})}$) in $O_\mathcal{T}$ when $P(\bar{S}_i|O_\mathcal{T}) \ll 1$, while it slowly converges to 1 for $P(\bar{S}_v|O_\mathcal{T}) \lesssim 1$ for increasing numbers of negative tests on paths passing through v. Similarly, $T_2 = \mathcal{P}_v$ when $\mathcal{P}_v < 1$, while $T_2 = 1 - \frac{\epsilon}{\mathcal{P}_v}$ for $\mathcal{P}_v \geq 1$.
In order to tune the value of ϵ we observe that if $q^* = \max \mathcal{P}_v \ s.t. \mathcal{P}_v < 1$, then $q^* \leq \frac{d-1}{d}$, where $d = |\bigcup_{m \in \mathcal{M}_v} \hat{m}^{(\mathcal{T})}|$. Therefore, for $\epsilon < 1 - q^*$, the growing trend of T_2 would be still satisfied when $\mathcal{P}_v$ exceeds 1.

4.5.1 Centrality-based Utility

Because of the dependencies among path failures, computing the joint probability $P(O_\mathcal{T})$ requires teh computation of $2^{|\mathcal{F}|}$ addends. In order to reduce computational costs, we approximate the probability that a path works, conditioned on the observation, as follows:

$$\tilde{P}_c(Z_i|O_\mathcal{T}) = \prod_{v \in \hat{m}_i^{(\mathcal{T})}} (1 - c(v|O_\mathcal{T})). \tag{4.18}$$

Definition 4.5.2. *The expected conditional utility based on failure node centrality is given by the formula:*

$$\mathcal{U}_c(a_i|O_\mathcal{T}) = \lambda(a_i|Z_i)\tilde{P}_c(Z_i|O_\mathcal{T}) + \lambda(a_i|\bar{Z}_i)\tilde{P}_c(\bar{Z}_i|O_\mathcal{T}) \tag{4.19}$$

if $\nexists\ m' \in \mathcal{F} : \hat{m}'^{(\mathcal{T})} \subseteq \hat{m}_i^{(\mathcal{T})}$. *Otherwise* $\mathcal{U}_c(a_i|O_\mathcal{T}) = 0$. *Here,* $\lambda(a_i|Z_i)$ *is defined as in equation (4.1) and* $\tilde{P}_c(\bar{Z}|O_\mathcal{T}) = 1 - \tilde{P}_c(Z|O_\mathcal{T})$.

The condition that equation (4.19) is valid if $\nexists\ m' \in \mathcal{F} : \hat{m}'^{(\mathcal{T})} \subseteq \hat{m}_i^{(\mathcal{T})}$ serves to recognize situations as the one described in Figure 4.3.4, where we observed that if a path fails, every of its super-path is going to be failing, too. Thanks to prior observation we can assess the state of such paths and therefore there is no need to probe them.

4.5.2 Probing Algorithm with Centrality: FaCeGreedy

Algorithm 2 may be adapted to use this metric instead of the exact conditional probability by applying the following modifications:

- Input: change p for c as initial node centrality.
- Lines 8 and 9: substitute $\mathcal{U}(a|O_\mathcal{T})$ with $\mathcal{U}_c(a|O_\mathcal{T})$.
- Line 10: replace $P(\bar{S}_v|O_\mathcal{T})$ with $c(v|O_\mathcal{T})$.

We hereby call FaCeGreedy (Failure Centrality Greedy algorithm) the Algorithm 2 with the modifications described above.

An example of execution of FaCeGreedy: By running FaCeGreey on the example in Figure 4.4.1, with initial node centrality $c = 0.1$ and $\epsilon = 0.05$, the path probe sequence is the same as the one resulting by PoPGreedy, and final node centralities are $c(v_9|O_\mathcal{T}) = 1$, $c(v_{10}|O_\mathcal{T}) = 0.1$ while $c(v|O_\mathcal{T}) = 0\ \forall v \in W$.

Computational Complexity

The computational complexity of Algorithm 2 changes when centrality and centrality-based utility (Definitions 4.5.1 and 4.5.2) are implemented instead of probability and utility (equation 4.2).

Theorem 4.5.1. *The computational complexity of FaCeGreedy (Algorithm 2 with the changes described above) is* $O(K \cdot (|V| \cdot |\mathcal{F}|^2 + |\hat{m}_{max}|))$*, where* K *is the maximum number of path probes,* V *is the set of nodes of the network,* $\mathcal{F}$ *is the set of failing tested paths and* $|\hat{m}_{max}|$ *is the maximum path length.*

Proof. The total number of tests is $O(\mathrm{K})$. computing the centrality of a node v requires scrolling the failed paths and searching for possible sub-paths in order to compute $\mathcal{P}_v$ (equation (4.17)). This is comprehensive of computing $|\mathcal{M}_v \cap \mathcal{F}|$ in equation (4.15) and requires $O(|\mathcal{F}|^2)$ operations. This is done for all nodes v at each iteration. Computing the centrality-based utility of a path requires a number of operations that is linear in the number of nodes paths pass through. The overall cost of the algorithm is $O(\mathrm{K} \cdot (|V| \cdot |\mathcal{F}|^2 + |\hat{m}_{max}|))$, where $|\hat{m}_{max}|$ is the maximum path length. □

4.6 Dynamic Failures

In this section we show how the proposed algorithms can be used in dynamic scenarios, where nodes' states may change throughout the monitoring activity, working nodes may fail, and broken nodes may be repaired. When we consider this scenario, past observations do not guarantee certain information, in contrast with the static model that we adopted in the previous sections. Furthermore, while in PoPGreedy and FaCeGreedy the path probe activity would naturally stop when the expected utility function of non-tested path results to be 0, the dynamic failure scenario that we are introducing can rather be classified as an infinite horizon problem. We adapt PoPGreedy and FaCeGreedy to take into account the newly introduced dependency on time by considering the following facts: *i.* we do not suppose to have any knowledge about prior node failure probabilities nor on the maximum number of failures; *ii.* we do not assume knowledge on the time required for a node to be fixed, nor on a node's life time. We call these dynamic-aware algorithms *Dynamic PoPGreedy (DPoPGreedy)* and *Dynamic FaCeGreedy (DFaCeGreedy)*. To model this scenario, we discretize time into the intervals between path probes, and we assign to each node v a probability to transition from working to failed ($p_{W \to F}$) and a probability to transition from failed to working (($p_{F \to W}$) at each time step. We assume that it is more likely for a node to be fixed, rather than for a node to fail (i.e., $p_{W \to F} < p_{F \to W}$). We base our procedure on the observation that information gained in the past progressively expires by the passing of the time. Because of the computational complexity that would result in a Bayesian analysis where probabilities are explicitly time-dependent, we consider the following simplified and easily computable approach: we define a *window* that is the set of the last probed paths. We assume that the width of the window ℓ_w is big enough to ensure at least network coverage. The window slides progressively: at each time step the least recently probed path in the window is removed from it, and a new path is probed and brought inside the window. We consider valid the information obtained by the last ℓ_w path probes, whereas we consider previous observations expired. DPoPGreedy and DFaCeGreedy work as their static counterparts inside the window, unless a contradiction is detected. A contradiction inside a window occurs when the joint probability of the last ℓ_w path probe outcomes is 0. This could happen either because a path traversing working nodes fails, or because a super-path of a failed path works. When this occurs, we locate the most recent path that causes a contradiction, and we remove it together with all the older paths from the window, as the information they provided is corrupted.

4.7 Experimental Results

In the following we provide a performance evaluation of both the variants PoPGreedy and FaCeGreedy of our approach, against state of the art solutions for classic Boolean Network Tomography and sequential graph-based group testing. In the experiments we assume cycle-free routing between monitor nodes. Our evaluation considers the following metrics: Section 4.7.1. If not explicitly stated otherwise, initial failure probability and centrality are set to 0.1.

4.7.1 Metrics

We consider the output of any of the probing algorithms in terms of the probability associated to each node failure. We compare the performance of the heuristics with respect to the results that would be obtained by using all the monitoring paths.

In the following, we call $W_{\mathcal{M}}$ and $B_{\mathcal{M}}$ the set of nodes correctly classified as working (failure probability 0) and broken (failure probability 1), respectively when all paths of $\mathcal{M}$ are probed. Bisognerebbe spiegare meglio che "tutti i path" sono tutti i path che ad un metodo è concesso provare. Per adaptive finder, si tratta di tutti i possibili path del grafo. Similarly, we denote with W_h and B_h the same sets according to the classification made by any of the heuristics h, which selects progressive monitoring policy, probing only a subset of the paths in $\mathcal{M}$.

We denote with $a_W \triangleq |W_h|/|W_{\mathcal{M}}|$ the *accuracy of detection of working nodes*, namely the fraction of nodes classified as working by the heuristics, over the number of nodes recognized as working when all available paths are probed.

Similarly we denote with $a_B \triangleq |B_h|/|B_{\mathcal{M}}|$ the *accuracy of detection of broken nodes*.

The next two metrics measure the correctness of the ranking $\boldsymbol{V}_f$ produced by the heuristics to sort the nodes in terms of failure probability: $R_1 \triangleq \frac{\left|\boldsymbol{V}_f[1:k] \cap F\right|}{k}$, where F is the set of failed nodes, $k = |F|$, and with $\boldsymbol{V}_f[1:k]$ we denote the nodes in the first k positions in the rank $\boldsymbol{V}_f$, i.e. with highest failure probability; $R_2 \triangleq \frac{k}{n-i+1}$, where i is the index of a failed node appearing in $\boldsymbol{V}_f$. If k is the number of truly failed nodes, R_1 counts how many of those appear in the highest k positions in the ranking, while R_2 says how many nodes' state we should verify before finding all the failed nodes.

It holds that $R_1 \leq 1$ and $R_2 \leq 1$, and $R_1 = 1$ when the top k positions are indeed occupied by the truly failed nodes F; $R_2 = 1$ when the nodes whose failing probability is 1 appear in the top k positions of the rank $\boldsymbol{V}_f$. Therefore, $R_1 = 1 \iff R_2 = 1$. R_1 metric is similar to the recall metric of ML [114], but it evaluates probabilistic outcomes instead of binary classifications. In addition to the metrics a_W, a_B, R_1 and R_2, we also consider the number of probes required to reach convergence and the execution time, when comparing our approaches to the previous solutions.

For evaluating DPoPGreedy and DFaCeGreedy, we use metrics that capture the ability to detect node state changes, and metrics that measure the reliability of the classification results step by step. For the first category, we compute the percentage of detected node state changes in both ways ($W \to F$ and $F \to W$), and the time for detection in terms of time stamps. For assessing the classification reliability in each sliding window, we use the classical definitions of precision and recall:

$$P = \frac{tp}{tp + fp}, \qquad R = \frac{tp}{tp + fn}$$

where tp (true positive) is the number of correctly classified nodes; fp (false positive) is the number of nodes erroneously classified either as working or as failed; fn (false negative) is the sum of the number of real working nodes that are not classified as working, and of the

number of real failed nodes that are not classified as failed. Notice that the recall is similar to R_1, except that R_1 evaluates probabilistic outcomes instead of binary classifications.

4.7.2 Benchmark solutions

To validate our approach we compare it with previous solutions based on classical Boolean Network Tomography as well as an approach based on progressive graph-constrained group testing. For the first set of benchmarks we consider the *greedy for coverage*, *greedy for identifiability* and *greedy for distinguishability* (GC, GI, GD) heuristics defined in [65]. At each iteration, the next path to probe among the available input paths is chosen as the one that maximizes network coverage/identifiability/distinguishability, respectively. When the greedy procedures meet some stopping criteria, node failure probabilities $P(\bar{S}_v|O_{\mathcal{T}})$ are computed and the outcome is evaluated in terms of the metrics described in Section 4.7.1.

Together with this, we compare our method to the adaptive, graph-constrained group testing algorithm introduced in [80], to which we refer to as AdaptiveFinder, (AF). The goal of AdaptiveFinder is to detect the set of defective items (nodes) in a graph with the least number of probes. The main differences between our setting and the one adopted by AdaptiveFinder are that, although graph-constrained, AdaptiveFinder is not routing-constrained, meaning that monitoring probes are not limited to move along end-to-end paths that are determined by the routing scheme implemented in the network, and given as input (i.e., they can be trees or contain cycles); in addition, direct node inspection is allowed through degenerate paths composed of only one node, meaning that all nodes are monitoring nodes. These two facts result in a major flexibility of AdaptiveFinder, i.e. an advantageous degree of freedom that is not available to our approach. Nevertheless, we note that this constitute an unrealistic capability in a general network tomography scenario and it is more expensive to implement on an actual network because it requires all nodes of the network to be provided with a monitoring system software and also assumes fully controllable routing. The set of paths available to our algorithm is limited to a small subset of the possible paths that AdaptiveFinder is allowed to walk across. For these reasons, accuracy metrics for AdaptiveFinder are taken with respect to the ground truth.

Notice that, because of the possibility of direct node inspection, there is no uncertainty in the sets of nodes classified as failed by AdaptiveFinder, hence this algorithm is not susceptible to lack of identifiability, that instead is an ascertained issue in network tomography. As a consequence, when the algorithm is run until convergence and a maximum number of recursive steps is not fixed, it manages to assess with certainty the state of all nodes, even when they are not $k-$identifiable, being k the number of broken nodes. Finally, we also compare our results with the Adaptive Path Construction (APC) algorithm, [103]. Similarly to PoPGreedy and FaCeGreedy, APC investigates on the state of the network by means of end-to-end monitoring paths that are given and determined by uncontrollable routing. APC may be divided in two phases. In the first phase and differently from us, a greedy for coverage is applied. The outcomes of the path probes used in this phase are then analysed, and if they are not sufficient for identifying the status of all nodes, the adaptive group testing phase is executed: the decision on the action to take at a certain step (i.e., the next path to probe) follows the binary search idea: the path whose number of nodes is closer to the half of the number of still unclassified nodes in the network is tested. The original output of APC is the set of the failed nodes and of the candidate nodes (nodes that are not classified as working and that might be failed). In PoPGreedy and FaceCeGreedy, these sets correspond to the set of nodes whose failing probability/centrality is 1, and to the set of nodes whose failing probability/centrality lies in $(0, 1)$, respectively. In order to compare APC in terms of the metrics introduced in Section 4.7.1, we compute the failure probability of the nodes in the candidate set, we set to 1 the failure probability of the nodes in the identified set, and as 0 the failure probability of all remaining nodes.

4.7.3 Tests

We perform experiments by considering different settings. In particular, we show experiments conducted on two different networks, an internet network in Europe, BICS [82], and a fiber network topology of Minnesota [12]. We use the first network for understanding thoroughly the behaviour of our algorithms and benchmark methods, and we see that such considerations hold on the bigger network.
In Table 4.7.1 features of the two topologies are detailed (left) as well as networks' features taking into account monitor-to-monitor path choices. Table 4.7.1 details features of the two topologies as well as networks' features taking into account monitor-to-monitor path choices. We use the smaller network, BICS, for running a thorough study of the behaviour of our algorithms and benchmarks, before extending our conclusions to the case of the larger Minnesota network. In the experiments, the set of candidate monitors is chosen randomly, with several paths between the same monitor pairs, to ensure broad network coverage.

	BICS	MN
$\|V\|$	33	681
$\|E\|$	48	921
δ_{min}	1	1
δ_{max}	8	13
δ_{avg}	3	2.7
diameter	9	29
$n_{\delta=1}$	5	134

	BICS	MN
$\|V^C\|$	33	450
$\|E^C\|$	43	610
$\delta^{\mathcal{M}}_{min}$	1	1
$\delta^{\mathcal{M}}_{max}$	29	631
$\delta^{\mathcal{M}}_{avg}$	9.9	60.7
longest path	9	27
$\|V_M\|$	10	62
$\|\mathcal{M}\|$	55	1996

Table 4.7.1. On the left: experimental settings. δ = node degree, $n_{\delta=1}$ = number of dangling nodes (degree 1). On the right: path characteristics. V^C is the set of covered nodes; E^C is the set of covered links; $\delta_i^{\mathcal{M}}$ = number of paths in $\mathcal{M}$ traversing node v_i.

Experiments on BICS network

Figures 4.7.1 to 4.7.4 are related to the BICS network. All curves are averaged on 20 experiments and show the value of the metrics defined in Section 4.7.1 on PoPGreedy, FaCeGreedy and all benchmarks. Shades/bars depict standard deviation. In the experimental configurations shown in Figures 4.7.1 and 4.7.2 all the approaches stop either when they reach convergence or when they reach a maximum number of path probes. Such bound is the number of path probes needed by PoPGreedy to converge (i.e., expected utility is 0) for each experiment. In particular, in Figure 4.7.1, we show the evolution of the metrics iterative-wise when four failures occur in the network. In Figure 4.7.2 instead, we show how the aforementioned metrics, as well as the elapsed times and the average number of tested paths, change for a growing number of failed nodes (from 1 to 5 failures). Notice that FaCeGreedy and GC always reach convergence before PoPGreedy (Figure 4.7.2f), but, while GC has poor, non-improvable performance in terms of node classifications, FaCeGreedy, together with PoPGreedy, always reach the same performances achieved by probing all paths (see Figures 4.7.2a to 4.7.2d), that is the upper-bound to the ability of node states assessment by means of end-to-end monitoring paths. Greedy identifiability and greedy distinguishability instead stop before convergence for all tests. AdaptiveFinder manages to converge with a very small number of paths only when a single failure

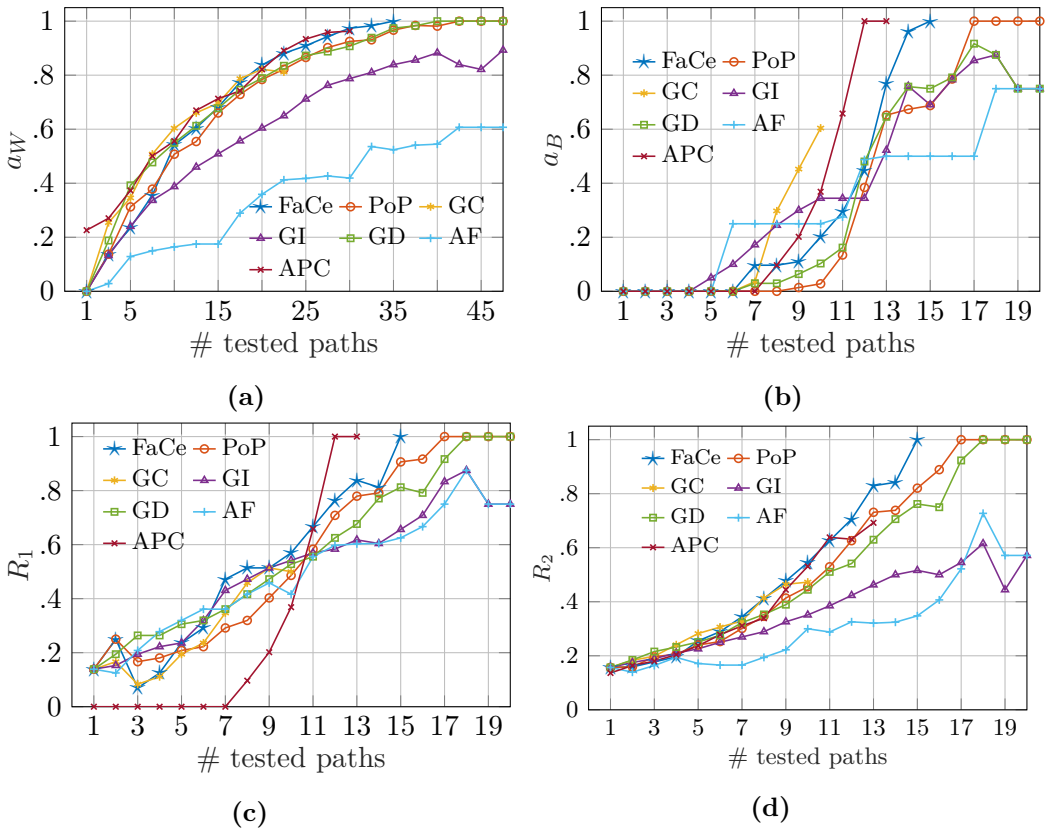


Figure 4.7.1. Metrics evolution, iteration-wise (BICS network with 4 failed nodes). Bounded.

occurs in the network. In contrast, APC converges with a a similar number of paths as FaCeGreedy. This is because the number of failures considered in this experimental scenario is small, and the initial coverage phase implemented by APC helps with the detection of many working nodes. Observing Figure 4.7.1 we can notice that since the number of tests changes depending on where the 4 failed nodes are located in the network, curves may be subject to oscillations at the end, as fewer tests reach the highest numbers of tested paths. Within one single test, a_W and a_B have a monotone growing trend, while R_1 and R_2 may oscillate: as a matter of facts, during intermediate probes working nodes may gain a high failure probability (hence moving to the top positions of the sorted node failure probability chart) and then their failure probability goes abruptly to 0 when a working path traverse them. An observable phenomenon is that a_W curves are concave and they grow steeply with the very first experiments, and become less steep when they approach the maximum value (i.e., $a_W = 1$). This is because of the sporadic failures scenario that we are considering: failed nodes are a small percentage of the set of all nodes, and therefore working paths are more likely to exist with respect to failing ones. Consequently, correct working node classification is easier and faster to achieve within the first tests. On the contrary, a_B curves follow a convex function trend and in the first tests they may be 0. This is because it takes a number of tests before a node can be classified as failed (i.e., failure probability equal to 1).

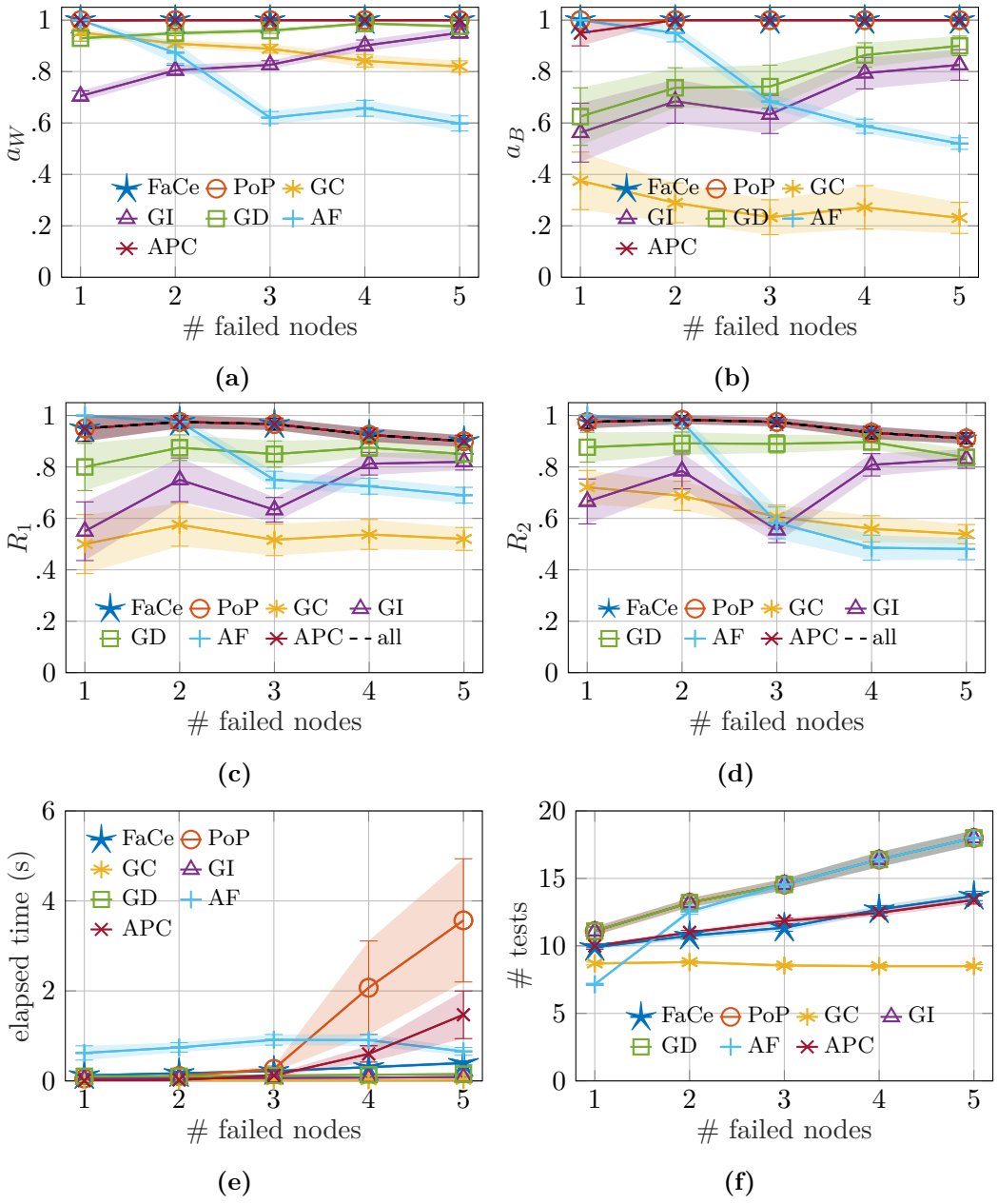


Figure 4.7.2. Tests on BICS network. Bounded

Similar considerations on the evolutionary curves hold for the experiments shown in Figure 4.7.3. Figures 4.7.3 and 4.7.4 are again related to the BICS network. In this experimental configuration, all methods stop either because the reach convergence, or because they probe all available monitoring paths. The latter condition does not hold for AdaptiveFinder, since it is not limited to move along given paths between monitors. Except for greedy coverage, consistently with what we observed for the previous set of experiments, AdaptiveFinder requires a greater number of tests than those used by PoPGreedy and FaCeGreedy to converge (see Figure 4.7.4f), while greedy identifiability and greedy distinguishability always probe all available

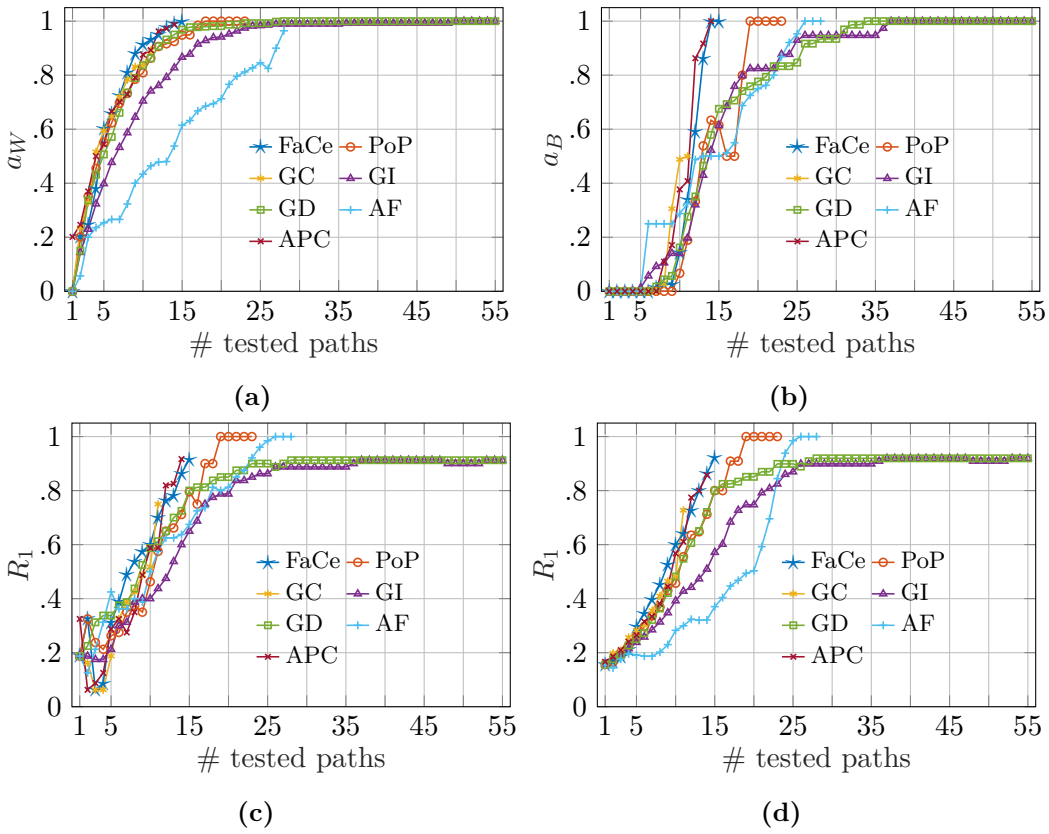


Figure 4.7.3. Metrics evolution iteration-wise (BICS network with 4 failed nodes). Unbounded.

paths. When we do not give constraints on the maximum number of paths to probe, AdaptiveFinder converges to the ground truth: it correctly classifies all nodes. We stress that this is due to its possibility to monitor single nodes directly and to its freedom to walk on the network without the restriction of moving along given paths. Once again, PoPGreedy and FaCeGreedy achieve the same performance as probing all paths would do, but testing with little portions of available monitoring paths. This holds also for APC in this failure scenario. As expected, the average elapsed time required by PoPGreedy considerably increases with the number of failed nodes, even on a small network (see Figures 4.7.2e and 4.7.4e). High variance is due to its exponentially dependence on the number of failed paths, amplifying the discrepancy of when central or non central nodes fail. For this reason, in the next set of experiments, we are not going to consider such method.

Minnesota

Figures 4.7.5 and 4.7.6 show our experiments on the Minnesota network. In Figure 4.7.5, tests are run until convergence or until a maximum number of tests K has been reached, whichever occurs earlier. In this case, the bound K is given by the number of path probes needed by FaCeGreedy to converge. In contrast, experiments in Figure 4.7.6 are run until convergence or until all available paths are

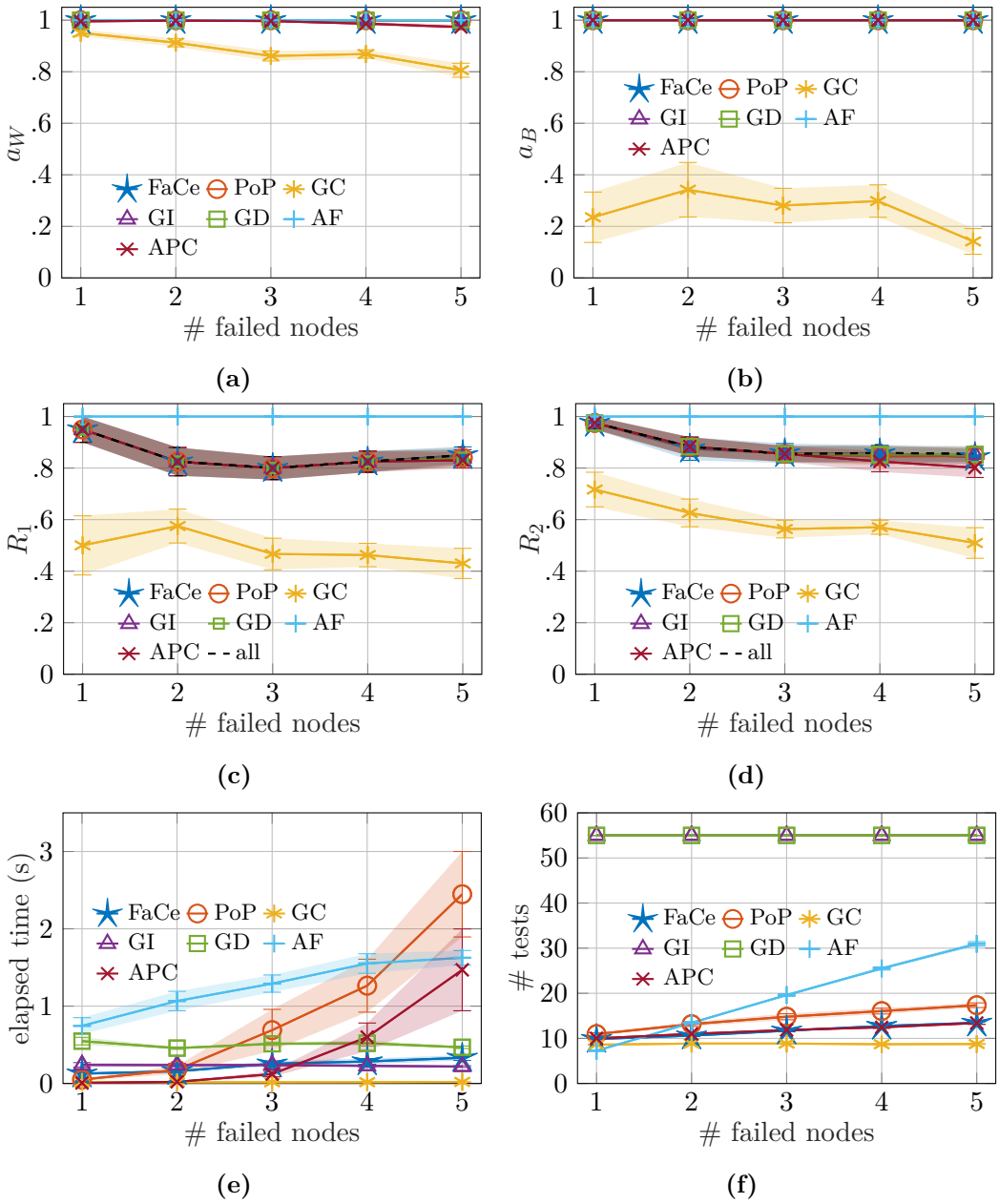


Figure 4.7.4. Tests on BICS network. Unbounded

probed. Again, we observe that GI and GD need to test all available paths and are still unable to converge because of their inability to take account of the progressively available information which can be obtained by probing the paths in a sequence. In fact, in Figure 4.7.5, GI and GD use the same number of paths as FaCeGreedy but with much inferior classification performance, whereas for the unbounded tests in Figure 4.7.5, they reach the same performances of FaCeGreedy by probing all available paths. On the other hand, FaCeGreedy is able to obtain full network information by converging with less than 9% of all the available paths. As in the previous experiments, CG is faster in covering the network, but performs poorly in

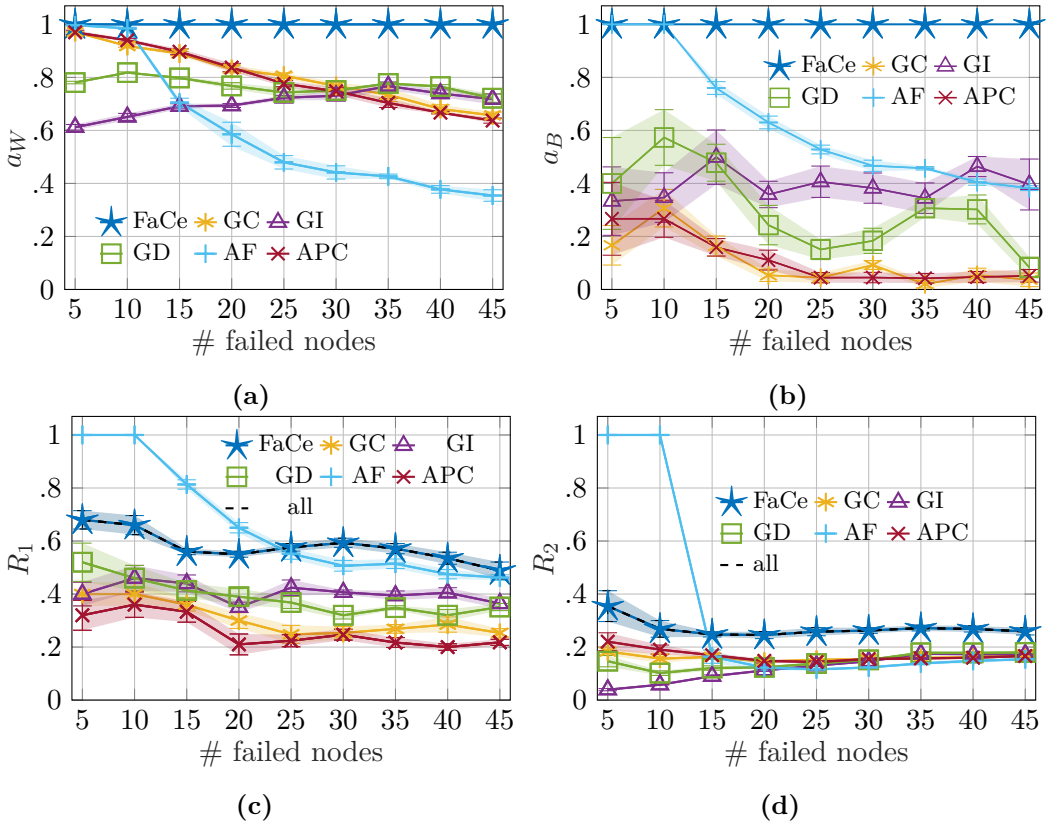


Figure 4.7.5. Tests on the Minnesota topology. Bounded.

terms of failure detection. Once again, in this configuration, in the unbounded case of Figure 4.7.5, AF is able to correctly detect all the failures within the maximum number of tests K only when the failure set is very small. In contrast, in the unbounded scenario, AdaptiveFinder reaches convergence with a higher number of tests than the ones required by FaCeGreedy (Figure 4.7.6e). Despite the good performance of APC in the previous network, when APC is run on Minnesota and when many failures occur, it reaches convergence with many more paths than the ones used by FaCeGreedy (Figure 4.7.6e), and performs poorly in the bounded tests (Figure 4.7.5). This is due to two factors: ensuring network coverage may be convenient for small networks with a little number of failed nodes, but it is not as effective in large networks with many failed nodes. Similarly, using a binary search approach is not as convenient when many failures occur.

Together with the aforementioned metrics, we also study how different choices of prior centrality values (c) may affect the performance of FaCeGreedy in terms of a_B. Figure 4.7.6f depicts how the accuracy of detection of broken nodes changes at each iteration of FaCeGreedy for $c = 0.05, 0.08, 0.1$. For each experiment, 35 failed nodes (8% of the total number of covered nodes) are generated. Despite curves vary throughout intermediate iterations, and despite small differences in the final number of tested paths FaCeGreedy is able to reach maximal accuracy (i.e., $a_B = 1$) also for under and over estimated choices of c (that is, $c = 0.05$ and $c = 0.1$) for all choices of c, proving its consistency, and robustness against potentially wrong settings of

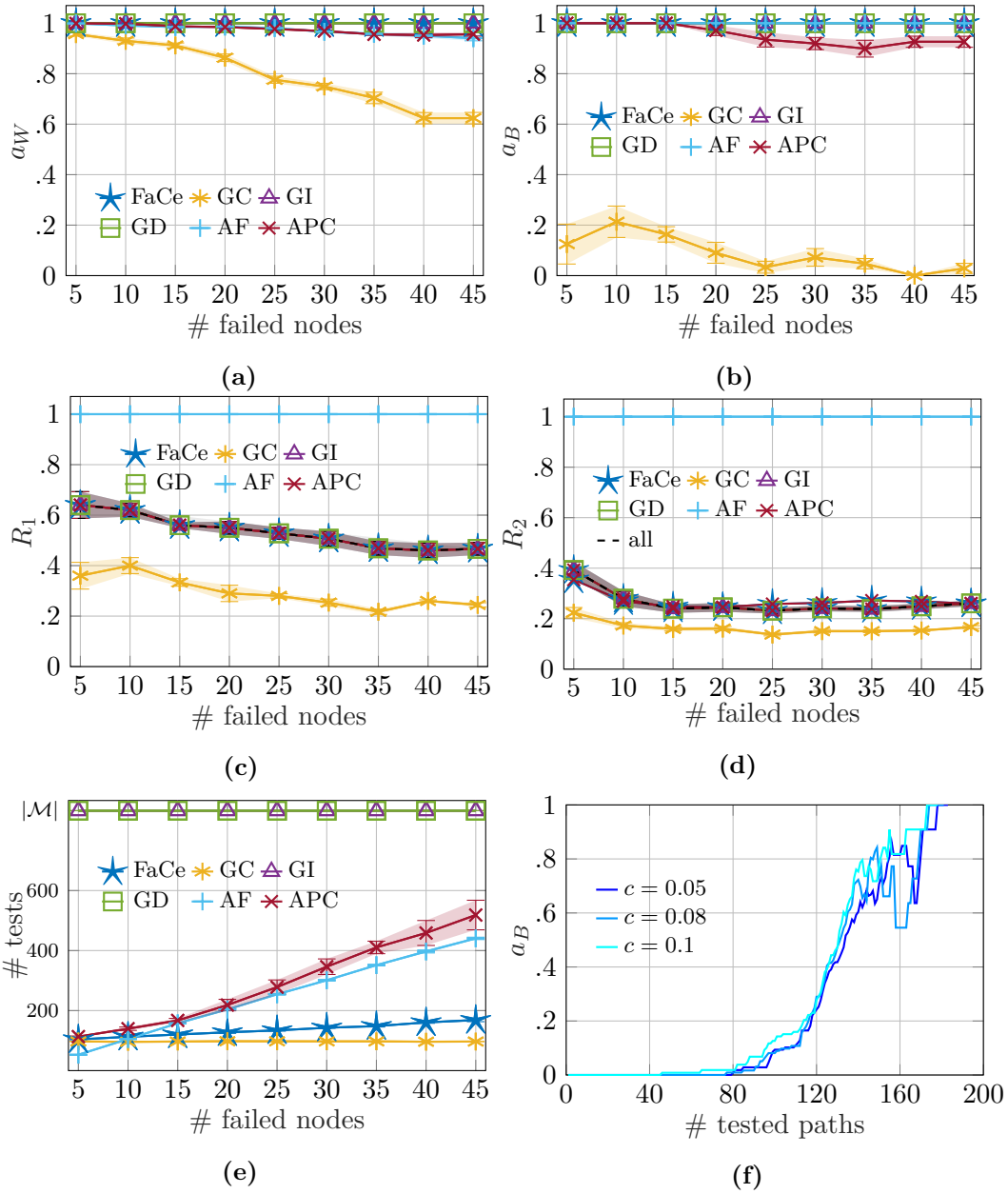
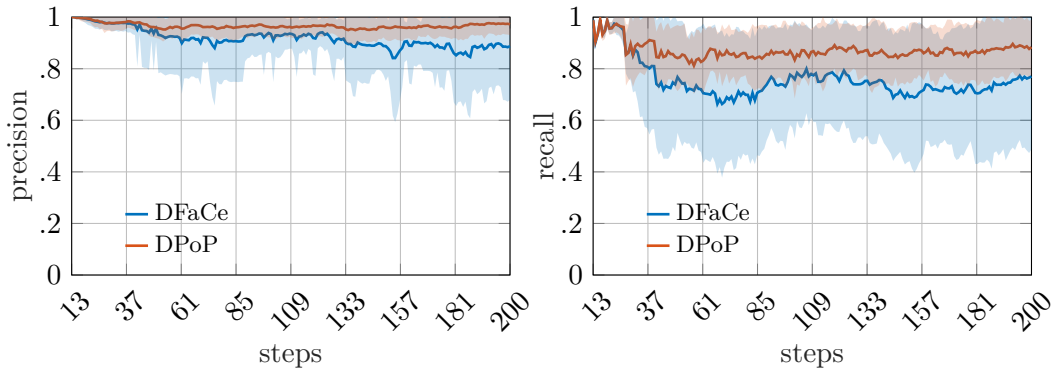


Figure 4.7.6. Tests on the Minnesota topology. Unbounded

the prior probability or centrality of a node.

Experiments on Dynamic Failures

Figure 4.7.7 shows the average precision and recall of DPoPGreedy and DFaCe-Greedy on the BICS network. We run the algorithms for 200 steps, and we consider the window size ℓ_w to be 12. We compute the evaluation metrics at each time step, from 13 to 200. In Table ..., we show the percentage of detected node state changes and the average time for change detection. The experiments are leveraged on 200



(a) (b)

Figure 4.7.7. Precision and Recall of DFaCeGreedy and DPoPGreedy.

	$\%_{F\to W}$	$\%_{W\to F}$	$t_{F\to W}$	$t_{W\to F}$
PoP	82.3 (±15.6)	89.7 (±9.1)	6.6 (±2.4)	12.7 (±6.3)
FaCe	56.7 (±21.5)	89.7 (±13.6)	14.0 (±8.7)	14.5 (±7.6)

Table 4.7.2. Average percentages of detected state changes and average time for change detection, and their standard deviation.

experiments, and standard deviations are provided between parenthesis.

4.8 Conclusions

Boolean Network Tomography (BNT) provides the design of end-to-end monitoring paths to ensure network failure localization. However, when the number of concurrent failures is unknown, BNT techniques hit the snag of the huge dimension and intractability of the solution space. With this work we propose a progressive approach to failure localization in the challenging scenario where failures may occur in an unknown and unbounded number. A set of monitoring paths is probed in a progressive manner, and decisions on which path to probe are made on the basis of a Bayesian approach which optimizes the expected value of the failure related information that can be obtained by incrementally monitoring new paths. To face the complexity of calculating posterior failure probabilities at each monitoring step, we propose a *failure centrality metric*, computable in polynomial time, which reflects the likelihood of a node to be the site of a failure. We use such a metric to guide decision making and provide a conclusive assessment of the state of network components. By means of numerical experiments conducted on synthetic as well as real network topologies, we demonstrate the practical applicability of our approach. The experiments show that our approach outperforms state of the art solutions based on classic Boolean Network Tomography as well as approaches based on progressive group testing.

Appendix

4.A Derivation of the minimum value of $\mathcal{U}(a|O_{\mathcal{T}})$ subject to $P(Z|O_{\mathcal{T}}) \in (0,1)$

In Section 4.4.2 we discuss what is the minimum value of $\mathcal{U}(a|O_{\mathcal{T}})$ subject to $P(Z|O_{\mathcal{T}}) \in (0,1)$. First of all, notice that when $|\hat{m}^{(\mathcal{T})}| > 1$, the second component of $\mathcal{U}(a|O_{\mathcal{T}})$, that is $\lfloor \frac{1}{|\hat{m}^{(\mathcal{T})}|} \rfloor P(\bar{Z}|O_{\mathcal{T}})$, is equal to 0. We can observe from Equation 4.12 that Δ_{min} decreases exponentially with $|\hat{m}|$, and therefore, $\exists n_0 \in \mathbb{N}$ such that $\forall\, |\hat{m}^{(\mathcal{T})}| \geq n_0$, $\mathcal{U}(a|O_{\mathcal{T}})$ decreases with $|\hat{m}^{(\mathcal{T})}|$. Such n_0 is $-\left\lfloor \frac{1}{\ln(P_{min})} \right\rfloor$, that is 0 for all $p \in (0,1)$. Therefore the decreasing trend of Δ_{min} holds for all possible values of $|\hat{m}^{(\mathcal{T})}|$. For this reason, for our analysis we legitimately consider the second component of $\mathcal{U}(a|O_{\mathcal{T}})$ to be zero. In Section 4.4.2, we claim that the minimum non-zero value of $P(Z|O_{\mathcal{T}})$ is $P_{min} = \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1+(1-p)(p^{\partial_v}-1)}$. As a matter of fact, the probability of failure of a path is the product of the conditional probability of failure of its nodes, $P(\bar{S}_v|O_{\mathcal{T}})$. Such probability depends on the number of failing paths traversing each node v, on their lengths and on their intersections. In particular, it is easy to see that the shorter the failing paths traversing the node and the least the cardinality of their intersections, the more $P(\bar{S}_v|O_{\mathcal{T}})$ grows. Therefore, the limit situation that we are seeking for occurs indeed when each node of $m^{(\mathcal{T})}$ is traversed by a large number of failing paths of length 2, i.e., paths passing through v and another node that is not in $\hat{m}^{(\mathcal{T})}$. When such condition holds though, then the set of all such 2-length paths is the set $\mathcal{F}_1^{(m,\mathcal{T})}$, as if m works, it would not only be possible to classify all of its nodes as working, but also all of the other nodes of such 2-length paths as failed. Therefore, in this situation it holds that $\mathcal{U}(a|O_{\mathcal{T}}) = (|\hat{m}^{(\mathcal{T})} + |\mathcal{F}_1^{(m,\mathcal{T})}|)P_{min} := \Delta_{min}$. We wonder if it is possible to get a smaller value of such $\mathcal{U}(a|O_{\mathcal{T}})$ in a situation where P_{min} is sacrificed for a slightly higher value of $P(Z|O_{\mathcal{T}})$, but where at the same time $|\mathcal{F}_1^{(m,\mathcal{T})}| = 0$. The next-most smaller value of $P(a|O_{\mathcal{T}})$ such that $\mathcal{F}_1^{m,(\mathcal{T})} = \emptyset$ results when all nodes v of $\hat{m}$ are traversed by paths of length 3, that only intersect in v and such that the two remaining nodes are not in $\hat{m}^{(\mathcal{T})}$. In this situation, if m works, all nodes it traverses would be identified as working, but none of the nodes of the original 3-length paths that are not in $\hat{m}^{(\mathcal{T})}$ would be uniquely identified, hence $\mathcal{F}_1^{(m,\mathcal{T})} = \emptyset$. We call P_3 such probability. It holds that:

$$P_3 = \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1 - \sum_{i=1}^{\partial_v} (-1)^{i+1} (1-p)^{2i+1} \binom{\partial_v}{i}}$$

$$
\begin{aligned}
&= \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1-(1-p)\sum_{i=1}^{\partial_v}(-1)^{i+1}(1-p)^{2i}\binom{\partial_v}{i}} \\
&= \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1+(1-p)\sum_{i=1}^{\partial_v}(-1)^{i}(1-p)^{2i}\binom{\partial_v}{i}} \\
&= \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1+(1-p)\sum_{i=1}^{\partial_v}(-1)^{i}[(1-p)^2]^{i}\binom{\partial_v}{i}} \\
&= \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1+(1-p)\sum_{i=1}^{\partial_v}[-(1-p)^2]^{i}\binom{\partial_v}{i}} \\
\text{[Newton's binomial]} \quad &= \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1+(1-p)[(1-(1-p)^2)^{\partial_v}-1]} \\
&= \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1+(1-p)[(1-1+2p-p^2)^{\partial_v}-1]} \\
&= \prod_{v \in \hat{m}^{(\mathcal{T})}} 1 - \frac{p}{1+(1-p)[(2p-p^2)^{\partial_v}-1]}.
\end{aligned}
$$

Hence the resulting conditional expected marginal benefit is $\mathcal{U}(a|O_{\mathcal{T}}) = |\hat{m}^{(\mathcal{T})}|P_3 =: \Delta_3$.

We see now that $\Delta_{min} < \Delta_3$ for growing values of $|\mathcal{F}^{(m,\mathcal{T})}|$. Asymptotically speaking, we can assume without loss of generality that $\partial_v = \partial_w \ \forall v, w \in \hat{m}^{(\mathcal{T})}$. Since $\sum_v \partial_v = |\mathcal{F}^{(m,\mathcal{T})}|$, it results that $\partial_v = \frac{|\mathcal{F}^{(m,\mathcal{T})}|}{|\hat{m}^{(\mathcal{T})}|}$. We show that $\frac{\Delta_{min}}{\Delta_3} < 1$ for growing values of $|\mathcal{F}^{(m,\mathcal{T})}|$.

$$
\begin{aligned}
\frac{\Delta_{min}}{\Delta_3} =& \frac{|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}^{(m,\mathcal{T})}| \left[1 - \frac{p}{1+(1-p)(p^{\partial_v}-1)}\right]^{|\hat{m}^{(\mathcal{T})}|}}{|\hat{m}^{(\mathcal{T})}| \left[1 - \frac{p}{1+(1-p)[(2p-p^2)^{\partial_v}-1]}\right]^{|\hat{m}^{(\mathcal{T})}|}} \\
=& \frac{|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}^{(m,\mathcal{T})}|}{|\hat{m}^{(\mathcal{T})}|} \left[\frac{1+(1-p)(p^{\partial_v}-1)-p}{1+(1-p)(p^{\partial_v}-1)} \cdot \frac{1+(1-p)[(2p-p^2)^{\partial_v}-1]}{1+(1-p)[(2p-p^2)^{\partial_v}-1]-p}\right]^{|\hat{m}^{(\mathcal{T})}|} \\
=& \frac{|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}^{(m,\mathcal{T})}|}{|\hat{m}^{(\mathcal{T})}|} \left[\frac{p^{\partial_v}(1-p)(2-p)^{\partial_v}+p}{(2-p)^{\partial_v}[p^{\partial_v}(1-p)+p]}\right]^{|\hat{m}^{(\mathcal{T})}|} \leq 1
\end{aligned}
$$

this is true $\forall |\hat{m}^{(\mathcal{T})}|$ if and only if $|\mathcal{F}^{(m,\mathcal{T})}|$ is sufficiently large and

$$\frac{p^{\partial_v}(1-p)(2-p)^{\partial_v}+p}{(2-p)^{\partial_v}[p^{\partial_v}(1-p)+p]} \leq 1$$

which is always true for $p \in (0,1)$, as it holds if and only if

$$
\begin{aligned}
& p^{\partial_v}(1-p)(2-p)^{\partial_v} + p \leq (2-p)^{\partial_v}[p^{\partial_v}(1-p)+p] \\
\iff & p \leq (2-p)^{\partial_v} p \\
\iff & (2-p)^{\partial_v} \geq 1 \qquad \text{since } p \in (0,1)
\end{aligned}
$$

$$\iff \partial_v = \frac{|\mathcal{F}^{(m,\mathcal{T})}|}{|\hat{m}^{(\mathcal{T})}|} \geq 0.$$

Therefore $\frac{\Delta_{min}}{\Delta_3} \to 0$ for growing values of $|\mathcal{F}_1^{(m,\mathcal{T})}|$ and it is < 1 for $|\mathcal{F}_1^{(m,\mathcal{T})}|$ sufficiently large and $\forall |\hat{m}^{\mathcal{T}}|$ and $\forall p \in (0,1)$. This proves that the minimum value of $\mathcal{U}(a|O_\mathcal{T})$ subject to $P(Z|O_\mathcal{T}) \neq 0,1$ is indeed $\Delta min = (|\hat{m}^{(\mathcal{T})}| + |\mathcal{F}_1^{(m,\mathcal{T})}|) \prod_{v \in \hat{m}} 1 - \frac{p}{1+(1-p)(p^{\partial_v}-1)}$.

Part II

Parallelization of Fundamental Operations in Numerical Linear Algebra

Chapter 5

Introduction to Part II

Having robust and reliable tools for fast computing is a very common and relevant concern in the most disparate applications in Computer Science, scientific and engineering modelling. Two main factors contribute to this purpose: algorithms having low computational and communication costs, and the availability of architectures for high performance computing. A wide set of problems relies more or less explicitly on numerical Linear Algebra in order to simplify problems whose symbolic solution would be prohibitively expensive in terms of computational cost. For this reason numerical Linear Algebra is continuously evoked in numerical Calculus in order to solve scientific and engineering problems as fast as possible. Pioneered by celebrated mathematicians such as Alan Turing, [139], John Von Neumann, [105], James H. Wilkinson, [143] and Alston S. Householder, [69], Numerical Linear Algebra was born with the purpose of implementing solutions to problems in continuous mathematics on the earliest computers. In recent years, with the favour of the growth of computational power of super computers, the research in numerical Linear Algebra has greatly evolved. This fact has allowed to widen knowledge and experimental possibilities in a great number of fields where Mathematical Analysis cannot be used in practice because of the size of the data and of the hardness of finding an analytic solution. Some peculiar examples of fields of science whose advances are partially made possible by research in Numerical Linear Algebra and High Performance Computing (HPC), are Meteorology (including weather forecasting, seismic phenomena simulations, wind and ocean waves predictions), Astrophysics (e.g., protoplanetary systems and galaxies modelling, simulating evolution of stars), Biology (e.g., modeling large bio-molecular systems, simulating bio-molecular reactions, understanding organ functions), just to cite a few.
Two pillars of numerical Linear Algebra are matrix multiplications and linear solvers, as they recur in several contexts. One example is Geometry processing for Computer Graphics, where 3D surfaces are discretized and represented as meshes on which differential operators (e.g., gradient, divergence and Laplacian) are computed on the simplexes of the graph resulting from the mesh. These operators are approximated by adjacency matrices that are usually very large and sparse, and that are applied to functions by means of classical matrix multiplications. Again in Geometry processing, Ovsjanikov et. al. in [107] were the first to phrase functional maps between manifolds in terms of systems of linear equations to search for correspondences between

non-Euclidean domains, allowing efficient point-to-point recognition of remeshed versions of 3D surfaces. In Computer Vision, and in particular in Convolutional Neural Networks (CNN), a great number of matrix convolutions are computed in order to extract features from images.
As already mentioned, another vast field of application of numerical Linear Algebra and HPC is system simulation. Observations on scientific phenomena (spanning from meteorology and astrophysics, to biological and chemical processes) and on engineering systems (such as vehicles, medical engines and building structures) are modelled by means of systems of either ordinary (ODEs) or partial (PDEs) differential equations. Finding an analytic solution to large systems of ODEs and PDEs is unfeasible, and instead numerical approximation methods are adopted. Some of these, as Domain Decomposition Method, Finite Element Method (FEM), Finite Difference Method (FDM), consist in transforming the systems of differential equations into linear systems, that can be readily solved by linear solvers coming from Linear Algebra.

Together with the formalization of algorithmic techniques for solving analytical problems, efficient parallel implementations that are frequently adapted to the growing industry of computers and clusters architectures, represent a major field of research for accelerating the solution of the aforementioned problems.

In this thesis, we focus on parallel implementations of basic matrix operations and linear solvers, because, as we have seen, they appear as intermediate steps integrated within complex frameworks in systems modelling and simulations and in several areas of Computer Science. In addition, we describe what we believe is the most appropriate parallel paradigm for implementing each operation.

In the following Section, we report some preliminary concepts in Numerical Linear Algebra that will be used throughout Part II of this thesis.

5.1 Preliminaries in Numerical Linear Algebra and HPC

Firstly, we shall recall that the multiplication between two matrices $\mathbf{A} \in \mathbf{R}^{m \times n}$ and $\mathbf{B} \in \mathbf{R}^{n \times p}$ results in a matrix $\mathbf{C} \in \mathbf{R}^{m \times p}$ such that $c_{i,j} = \langle \mathbf{a}_{i,:}, \mathbf{b}_{:,j} \rangle$, with $i = 1, \ldots, m$ and $j = 1, \ldots, p$, being $c_{i,j}$ the entry of $\mathbf{C}$ with coordinates (i, j), whereas $\mathbf{a}_{i,:}$ and $\mathbf{b}_{:,j}$ are the i-th row and the j-th column of matrices $\mathbf{A}$ and $\mathbf{B}$, respectively. The operation $\langle ., . \rangle$ is the scalar product, that is well defined as $\mathbf{a}_{i,:}, \mathbf{b}_{:,j} \in \mathbf{R}^n$. It is easy to see that the computational cost of the standard matrix multiplication is cubic in the matrices' dimensions, $O(m \cdot n \cdot p)$. Trivially, if the matrices are square, i.e., $\mathbf{A}, \mathbf{B} \in \mathbf{R}^{n \times n}$, the computational cost becomes $O(n^3)$.
The first to break the $O(n^3)$ operation count for matrix multiplication was Strassen in 1969, [133]. While the naive matrix multiplication between two 2×2 matrices $\mathbf{A}$ and $\mathbf{B}$ requires the following 8 products:

$$
\begin{aligned}
c_{1,1} &= a_{1,1}b_{1,1} + a_{1,2}b_{2,1} \\
c_{1,2} &= a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\
c_{2,1} &= a_{2,1}b_{1,1} + a_{2,2}b_{2,1} \\
c_{2,2} &= a_{2,1}b_{1,2} + a_{2,2}b_{2,2}
\end{aligned}
$$

where $\mathbf{C} = \mathbf{AB}$, Strassen's algorithm only uses the following 7 multiplications:

$$
\begin{aligned}
m_1 &= (a_{1,1} + a_{2,2})(b_{1,1} + b_{2,2}) \\
m_2 &= (a_{2,1} + a_{2,2})b_{1,1} \\
m_3 &= a_{1,1}(b_{1,2} - b_{2,2}) \\
m_4 &= a_{2,2}(b_{2,1} - b_{1,1}) \\
m_5 &= (a_{1,1} - a_{1,2})b_{2,2} \\
m_6 &= (a_{2,1} + a_{1,1})(b_{1,1} + b_{1,2}) \\
m_7 &= (a_{1,2} - a_{2,2})(b_{2,1} + b_{2,2})
\end{aligned}
$$

that are eventually summed up to recover the product matrix $\mathbf{C}$:

$$
\begin{aligned}
c_{1,1} &= m_1 + m_4 - m_5 + m_7 \\
c_{1,2} &= m_3 + m_5 \\
c_{2,1} &= m_2 + m_4 \\
c_{2,2} &= m_1 - m_2 + m_3 - m_6.
\end{aligned}
$$

By generalization of 2×2 matrices to matrices of any size, divided in 2×2 blocks, Strassen's algorithm applies the 7 product computation recursively. The computational cost of Strassen's algorithm on matrices of size n therefore is $O(n^{\log_2 7}) \sim O(n^{2.81})$. Strassen's algorithm is celebrated also for being cache oblivious, as it can be implemented efficiently without requiring knowledge of specific memory and block size. After Strassen's results, others have provided algorithms with progressively lower computational costs. In 1987, Coppersmith and Winograd proposed an algorithm reducing the computational cost of generic matrix multiplications to $O(2^{2.38})$, [33]. This bound has been further improved in the last ten years by Stothers in 2010, [131], Williams in 2012 [144] and Le Gall in 2014 [58]. Currently the best bound belongs to Josh Alman and Virginia Vassilevska Williams, and is 2^{ω}, with $\omega = 2.3728596$, [3]. Despite the great efforts put for pushing this bound, these results are of purely theoretical interest and not usable in practice, as they are valid only for inputs beyond any practical size and because they exhibit serious numerical instability problems, [19, 125]. In this thesis, we show an algorithm that uses Strassen's procedure for the $\mathbf{A}^T\mathbf{A}$ matrix multiplication. The algorithm can be easily converted for shared and distributed architectures, and exhibits advantageous performance also on small sized matrices.

Another topic of interest in this thesis is the solution of systems of linear equations, $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. As we have seen in this chapter, they are a pillar to many applications in science and engineering. Recall that a system has one and only one solution if and only if the coefficient matrix $\mathbf{A}$ is non-singular, meaning that it is square and its determinant is non-zero. Depending on the properties of the matrix $\mathbf{A}$, that can be either structural (the matrix is *sparse* or *dense*) or mathematical ($\mathbf{A}$ is symmetric, positive-definite, orthogonal, etc.), different algorithms can be applied to solve the system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. Despite the vast literature, solving a linear system can still be challenging for a number of reasons: if the system is very sensitive to small perturbations in the input that may be caused by round-off errors (inherently

introduced in the systems when it is solved by a computer) it might be necessary to study preconditioning techniques to avoid error explosion and decrease sensitiveness; when a linear system is very large, providing a proper, scalable implementation that uses parallel architectures optimizing memory usage and limiting communication is a strategic task; because of the size of the input and of possible lack of desirable properties of the coefficient matrix, some problems require ad-hoc solutions in terms of algorithmic procedures and data structures.

Algorithms for solving linear systems can be divided into two categories: *direct* and *iterative* solvers. Direct solvers produce exact solutions to linear systems (excluding round-off), whereas iterative solvers generate a sequence of improving approximate solutions: given a linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, the idea is to define a matrix $\mathbf{M}$, such that the system $\boldsymbol{x} = \boldsymbol{x}(I - \mathbf{M}^{-1}\mathbf{A}) + \mathbf{M}^{-1}\boldsymbol{b}$ is equivalent to $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, being I the identity matrix. The matrix $\mathbf{B} := I - \mathbf{M}^{-1}\mathbf{A}$ is known as *iteration matrix*. Solutions are progressively updated as $\boldsymbol{x}^{(k+1)} = \mathbf{B}\boldsymbol{x}^{(k)} + \mathbf{M}^{-1}\boldsymbol{b}$ for any initial guess solution $\boldsymbol{x}^{(0)}$. An iterative method converges if and only if the spectral ray of $\mathbf{B}$ is less than 1. Formally, $\lim_{k\to\infty} \boldsymbol{x}^{(k)} = \boldsymbol{x}, \forall\, \boldsymbol{x}^{(0)} \iff \rho(\mathbf{B}) < 1$, where $\boldsymbol{x}$ is the exact solution to $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, [61]. Depending on the problem that is to be solved, one may choose to implement a direct or an iterative solver. As we shall detail in Chapter 7, direct solvers are preferable over iterative on dense systems, whereas direct solvers are commonly applied for solving sparse systems. A system is sparse when its coefficient matrix is a sparse matrix, i.e., if for each of its row and column the number of entries that are non-zero is $O(1)$.

We hereby list some of the most well-known linear solvers.

Gaussian Elimination/LU decomposition Given a linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, this direct method consists in repeatedly updating the rows of the augmented matrix $(\mathbf{A}|\boldsymbol{b})$ by summing them to the non-zero scalar multiple of other rows. The scalars are called *multipliers*. This procedure provides an equivalent system $\mathbf{A}'\boldsymbol{x} = \boldsymbol{b}'$, where the coefficient matrix is upper triangular. In the LU-decomposition, the Gaussian Elimination is applied only to the coefficient matrix $\mathbf{A}$, that is decomposed as the product of a lower-triangular and a upper-triangular matrix ($\mathbf{A} = \mathbf{LU}$), where $\mathbf{L}$ is the matrix of the multipliers. This technique can be applied to any coefficient matrix, even to non-square ones.

Cholesky Decomposition It is a direct method for solving systems of linear equations $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where $\mathbf{A}$ is a positive-definite matrix. It consists of decomposing the coefficient matrix as the produce of a lower triangular matrix and its transpose, $\mathbf{A} = \mathbf{L}\mathbf{L}^T$.

Jacobi algorithm This procedure falls in the class of iterative methods. The linear system is written in a form where each $i-$th equation is solved by its $i-$th variable, x_i. Starting from an initial solution guess, $\boldsymbol{x}^{(0)}$, the method consists in using the progressively updated solution $\boldsymbol{x}^{(k)}$ to derive a more accurate solution $\boldsymbol{x}^{(k+1)}$ by plugging the values of $\boldsymbol{x}^{(k)}$ in to the right-hand side of the equations written as described above. It can be expressed in matrix form by defining the iteration matrix $\mathbf{B} := I - \mathbf{D}^{-1}\mathbf{A}$, where $\mathbf{D}$ is the diagonal of $\mathbf{A}$.

A sufficient condition for convergence is that the coefficient matrix is strictly diagonally dominant. A detailed version of this method is in Chapter 7.

Gauss-Seidel algorithm This iterative method is very similar to Jacobi, except that in this algorithm, newly computed entries of the updated solution $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ are immediately plugged into the variables appearing in the $i-$th equation, $\forall i$. In matrix form, it can be described by the equation $\boldsymbol{x}^{(k+1)} = \mathbf{L}^{-1}(b - \mathbf{U}\boldsymbol{x}^{(k)})$, where $\mathbf{L}$ is the lower triangular part of $\mathbf{A}$, and $\mathbf{U}$ is its strictly upper triangular part.

Krylov subspace methods In [84], Krylov proved that the inverse of a matrix $\mathbf{A}$ can be expressed as a linear combination of its powers. This result has inspired a new way of designing iterative algorithms, that aims at searching for the solution of linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ in the Krylov subspace of size r, $\mathcal{K}_r(\mathbf{A}, \boldsymbol{b}) = \{I, \mathbf{A}\boldsymbol{b}, \mathbf{A}^2\boldsymbol{b}, \ldots, \mathbf{A}^{r-1}\boldsymbol{b}\}$. Chronologically, the first such method was the Conjugate Gradient (CG) algorithm, that is applicable to positive definite matrices, and that is generalized by the biconjugate gradient (BiCG) and more recently by the biconjugate gradient stabilized (BiCGSTAB) methods, to work stably also on non self-adjoint matrices. Other methods, such as GMRES and MINRES, search for a solution in $\mathcal{K}_r(\mathbf{A}, \boldsymbol{b})$ by iteratively minimizing the norm of the residual $||\mathbf{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}||$, [123].

Incomplete factorizations Matrix factorizations can be used for solving linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ by decomposing the coefficient matrix into the product of matrices that have mathematical or structural properties that ease the solution of the system. The LU factorization and the Cholesky decomposition described above are two examples. Nevertheless, decomposing a sparse matrix rarely results in the product of two sparse matrices. In order to prevent this, incomplete factorizations may be employed. The Incomplete LU factorization (ILU) factorizes a sparse matrix $\mathbf{A}$ into the product of a sparse lower triangular matrix $\mathbf{L}$ and a sparse upper triangular matrix $\mathbf{U}$ such that $\mathbf{R} = \mathbf{LU} - \mathbf{A}$ satisfies certain constraints, such as having zero entries in some locations, [123]. Similarly, the Incomplete Cholesky Decomposition (ICHOL) consists in computing a sparse, positive definite matrix $\mathbf{A}$ as the product of a sparse lower triangular matrix $\mathbf{L}$ and its transpose such that $\mathbf{A} \approx \mathbf{L}^T\mathbf{L}$, where $\mathbf{L}$ has a 0 wherever $\mathbf{A}$ has one, [61].

In this thesis, we focus on a class of linear systems that we call Quasi-Block Diagonal (QBD), and we show how these systems can be conveniently implemented on distributed architectures with negligible communication cost.

To complete our introduction to the second part of this thesis, we briefly describe the parallel programming models and architectures that we exploit to implement our numerical algorithms.

High performance computing (HPC) is the ability to process data and perform complex calculations at high speeds. This is achieved by running programs in parallel architectures where possibly numerous nodes work together to complete one or more tasks simultaneously. In the second part of this thesis, we implement well-engineered

numerical algorithms on clusters of multi-cored nodes. A simplified view of this architecture is depicted in figure 5.1.1. Compute nodes are connected with one another via a high speed interconnect network. Nodes do not share a common memory, and therefore they need to exchange messages through the network lines in order to cooperate, synchronize and share data. The Message Passage Interface (MPI), [137], is a standardized and portable message-passing standard that allows distributed processors to communicate with one another, and provides user friendly and portable libraries supporting SPMD/MIMD parallelism. In this thesis, we rely on MPI for distributing jobs among different nodes. Each distributed node can in turn contain multiple cores, i.e, be a multiprocessor. In a multiprocessor system all processes on the various CPUs share a unique logical address space, which is mapped on a physical memory that is shared among the processors. To exploit the computational power of multiprocessors, the software needs to be implemented in a multi-threading fashion, i.e., a parallel compute model whereby a primary thread forks a specified number of sub-threads and the system divides a task among them. Threads then run concurrently. To handle multi-threading, in this thesis we rely on the Open Multi-Processing API (OpenMP, [34]). We also exploit the Intel Math Kernel Libraries (MKL, [55]) multi-thread optimization of standard software libraries for numerical Linear Algebra, as BLAS and LAPACK. In MKL, cluster versions of numerical Linear Algebra routines are also available to take advantage of MPI parallelism in addition to single node parallelism from multi-threading.

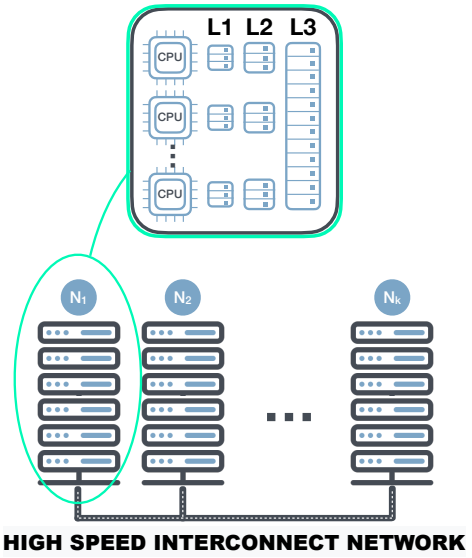


Figure 5.1.1. Cluster of multi-cored nodes.

Chapter 6

Efficiently Parallelizable Strassen-Based Multiplication of a Matrix by its Transpose

The multiplication of a matrix by its transpose, $\mathbf{A}^T\mathbf{A}$, appears as an intermediate operation in the solution of a wide set of problems. In this chapter, we propose a new cache-oblivious algorithm (AtA) for computing this product, based upon the classical Strassen algorithm as a sub-routine. In particular, we decrease the computational cost to $2/3$ the time required by Strassen's algorithm, amounting to $\frac{14}{3}n^{\log_2 7}$ floating point operations. AtA works for generic rectangular matrices, and exploits the peculiar symmetry of the resulting product matrix for saving memory. In addition, we provide an extensive implementation study of AtA in a shared memory system, and extend its applicability to a distributed environment. To support our findings, we compare our algorithm with state-of-the-art solutions specialized in the computation of $\mathbf{A}^T\mathbf{A}$, as well as with solutions for the computation of the general $\mathbf{A}^T\mathbf{B}$ product applied to this problem. Our experiments highlight good scalability with respect to both the matrix size and the number of involved processes, as well as favorable performance for both the parallel paradigms and the sequential implementation, when compared with other methods in the literature. The work presented in this chapter has been made in collaboration with Filippo Maggioli (M.Sc.), to whom goes the merit for code optimization, shared-memory implementations and with whom I worked on the algorithmic analysis, prof. Annalisa Massini and prof. Emanuele Rodolà. The results shown in this chapter has been recently accepted as a class A conference paper, [7].

6.1 Introduction

Matrix multiplication is a fundamental operation in Linear Algebra and high-performance computing, as it appears as an intermediate step in a wide set of problems. Many researchers have devoted their efforts to the algorithmic aspects of matrix multiplication, with the aim of improving the computational cost of existing algorithms and to devise and implement new solutions for parallel architectures. Designing a distributed algorithm for matrix multiplication is a challenging task,

due to the inherent dependence of the data scattered in the system's distributed memory, and due to the overhead due to the communication cost of assembling the resulting product matrix.

The product of a matrix by its transpose, $\mathbf{A}^T\mathbf{A}$ (as well as $\mathbf{A}\mathbf{A}^T$), is a particular matrix multiplication involved in several applications. For example, computing $\mathbf{A}\mathbf{A}^T$ is a straightforward, yet effective, method to check for orthogonality or to project vectors onto the space spanned by the columns of $\mathbf{A}$. This product, in fact, is repeatedly computed in the Gram-Schmidt algorithm for vector basis orthogonalization, where $\mathbf{A}$ is the progressively built projection matrix. One way to solve the least squares problem of under and over determined linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, is to solve the associated system of normal equations, obtained by left-hand multiplying the original system by $\mathbf{A}^T$, thus obtaining a square linear system $\mathbf{A}^T\mathbf{A}\boldsymbol{x} = \mathbf{A}^T\boldsymbol{b}$. Also, the Singular Value Decomposition (SVD) of a matrix $\mathbf{A}$ can be computed by studying the eigenproblem for $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$. Furthermore, the product of a matrix by its transpose commonly arises in discrete exterior calculus and discrete differential geometry. One example is given by the discrete heat kernel $\mathbf{K}(t) = \mathbf{\Phi}\mathbf{E}(t)\mathbf{\Phi}^T$, with $\mathbf{E}(t) = \exp(-\mathbf{\Lambda}t)$ being a diagonal matrix, so that $\mathbf{K}(t) = (\mathbf{\Phi}\mathbf{E}(t)^{1/2})(\mathbf{\Phi}\mathbf{E}(t)^{1/2})^T$ can be efficiently computed [146]. Many other applications of the product $\mathbf{A}^T\mathbf{A}$ are described in [132], together with its properties such as positive semi-definiteness.

In this work, we consider the multiplication between $\mathbf{A}^T$ and $\mathbf{A}$, where $\mathbf{A}$ may have any size and shape. We rely on a recursive approach that, as described in [109], is virtually tuning free and avoids the significant tuning efforts that are required by iterative blocked algorithms to achieve near-optimal performance. Our contribution is threefold.

- First, we introduce AtA (Section 6.3), a cache-oblivious algorithm for computing $\mathbf{A}^T\mathbf{A}$ that requires $2/3n^{(\log_2 7)} + 1/3n^2$ multiplications. We exploit the self-similarity of the $\mathbf{A}^T\mathbf{A}$ product with its sub-problems and the Strassen's algorithm, that is recursively applied to possibly rectangular matrices, without introducing additional computational and space cost, deriving from dynamic peeling and padding, as in [72, 138]. As an ideal extension, and differently from the approach presented in [47], our algorithm is not restricted to work on a specific algebraic field (i.e., $\mathbb{C}$ or any other finite field). We prove that AtA exhibits high efficiency for both memory and time, and show that the computational complexity of our algorithm does not hide large constant factors, making it efficiently implementable. Besides, we provide the overall asymptotic computational complexity. We also describe our implementation of Strassen's algorithm, and compare its performance with that of the Intel MKL BLAS `gemm` routine for matrix multiplication.

- Second, we describe AtA-S, our multi-threaded implementation of AtA for a shared memory system, relying on OpenMP (Section 6.4.2). Our implementation is dual-phase: in the first part of AtA-S, a scheduler simulating the recursion of AtA assigns different tasks to each thread in such a way that the compute phase can be carried out in perfect parallelism by preventing memory collisions. Performance evaluation shows that our implementation outperforms the multi-threaded Intel MKL BLAS routines (e.g. `syrk` for symmetric rank-K update) on large matrices, even when using Intel processors.

- Finally, we extend our approach for shared memory systems to distributed systems, leveraging the standard message-passing paradigm MPI. Our distributed algorithm ATA-D allows the distribution of the computational effort among a larger number of processes (Section 6.4.3). This is particularly convenient on very large matrices that require a prohibitive computation time.

To validate the effectiveness of our algorithms, we study their performance by running a set of tests on dense matrices of variable size (Section 6.5). We analyse different metrics for evaluating the scalability of our parallel implementation, and compare our results with benchmark solutions for distributed systems. We run tests on a cluster of multi-core nodes endowed with 2×8 core Intel Xeon E5-2630v3 processors, 2.4 Ghz, 4 GB RAM/core.

6.2 Related work

Nowadays, matrix multiplication is still a hot topic in HPC and numerical algorithmics. In 1969, Strassen [133] was the first to reduce the computational complexity of the standard matrix multiplication from $O(n^3)$ to $O(n^{\log_2 7}) \sim O(n^{2.81})$. More recently, Coppersmith and Winograd [33] devised an algorithm for matrix multiplication running in $\sim O(n^{2.38})$ time. In the last decade, this limit has been improved first by Stothers [131], then by Williams in [144], and finally by Le Gall in [58]. The latter works make use of algebraic tensors that, despite the elegance of the resulting method, are still hardly used in practice as they come at the cost of very large hidden constants and frequent memory access.

Several authors have designed hybrid algorithms, deploying Strassen's multiplication in conjunction with conventional matrix multiplication, to overcome the overhead of Strassen's algorithm on small matrices, see, e.g., [20, 24, 25, 67, 72]. Huss-Lederman *et al.* [72] propose two techniques, known as dynamic peeling and static padding, in order to apply Strassen's algorithm to odd-sized matrices. Thottethodi *et al.* [138] propose two strategies to optimize memory efficiency in Strassen, using a quad-tree decomposition of the matrices, and select the recursion truncation point to minimize padding and peeling without affecting performance.

Many researchers have proposed a parallel implementation of Strassen's algorithm. In [91], Luo and Drake explored Strassen-based parallel algorithms that use the communication patterns known for classical matrix multiplication. They considered using a classical 2D parallel algorithm and using Strassen locally. They also considered using Strassen at the highest level and performing a classical parallel algorithm for each sub-problem generated. In [63], the above approach is improved using a more efficient parallel matrix multiplication algorithm running on a more communication-efficient machine. They obtained better performance results compared to a purely classical algorithm for up to three levels of Strassen's recursion. In [85], Strassen's algorithm is implemented on a shared-memory machine. The trade-off between available parallelism and total memory footprint is found by distinguishing between *partial* and *complete* evaluation of the algorithm, i.e., breadth-first and depth-first traversal of the recursion tree. In [35], Strassen's algorithm is extended to deal with rectangular and arbitrary-size matrices. Their approach leverages on a suitable combination of Strassen's with ATLAS and GotoBLAS, and achieves up to 30%/22%

speed-up versus ATLAS/GotoBLAS alone on high-performance single processors. Other parallel approaches [38, 71, 130] have used more complex parallel schemes and communication patterns, and consider at most two steps of Strassen. In [13], a parallel algorithm based on Strassen's fast matrix multiplication, Communication - Avoiding Parallel Strassen (CAPS), is described. The authors present both a computational and a communication cost analysis of the algorithm, and show that it matches the communication lower bounds described in [14]. This work is extended in [37] to handle rectangular matrices (CARMA). More recently, Kwasniewski *et al.* [86] proposed a near optimal algorithm for matrix multiplication that models the matrix multiplication dependencies by the red-blue pebble game [73] to derive an I/O optimal schedule, improving the performance of previous works.

Both Strassen's algorithm and AtA fall into the class of recursive blocked algorithms. The work in [51] proves the effectiveness of this kind of algorithms for dense Linear Algebra. The work in [49] introduces FRPA, an interface for implementing recursive problems in parallel that gets as an input the recursive problem, and handles parallelization and auto-tuning automatically. Similarly to our approach, Charara *et al.* [28] propose block recursive matrix multiplication and linear solver algorithms. They show how recursion enhances data reuse and concurrency in GPUs. Differently from the work presented in this chapter, they specialize on triangular matrices. In [27], the authors also adapt this blocking strategy to handle batched operations on small matrix sizes (up to 256) to stress the register usage and maintain data locality. In [109], Elmar and Bientinesi introduce ReLAPACK, a collection of recursive algorithms for dense Linear Algebra. While this work corroborates the recursive approach that we implement in our algorithms, it does not provide a routine specialized in the $\mathbf{A}^T\mathbf{A}$ product for general matrices. Instead, they propose a routine for the same multiplication only on triangular matrices. We highlight that the solutions proposed for the multiplication of a matrix by its transpose on triangular matrices (TRSYRK) is useful for many applications but cannot be applied on general matrices.

Although much research has been devoted to optimizing the implementation of parallel matrix multiplication, very few solutions have been proposed for the $\mathbf{A}^T\mathbf{A}$ multiplication. In [47], Dumas *et al.* propose an algorithm for the product $\mathbf{A}\mathbf{A}^T$ whose computational complexity is improved by a constant factor with respect to previously known reductions. Unfortunately, this approach is applicable only to matrices lying in fields where skew-orthogonal matrices exist (e.g., $\mathbb{C}$ and finite fields of prime characteristics), which is not the case for $\mathbb{R}$ and $\mathbb{Q}$, that instead are important in many applications, such as the study of embedded systems, computational geometry and system simulations.

Except for some sporadic attempts to implement a method for distributing in a balanced way the workload for matrix multiplication among processes with the MapReduce programming model [76, 116], the approach that we implement here for the distributed parallel model has barely been investigated.

6.3 AtA

In this section, we describe our sequential recursive algorithm for the matrix multiplication $\mathbf{A}^T\mathbf{A}$, dubbed AtA, and we provide implementation details. We remark that our solution also works for the product $\mathbf{A}\mathbf{A}^T$ product. Yet, when row-major order is the default layout for array storage, the $\mathbf{A}^T\mathbf{A}$ multiplication is in practice harder to perform, as memory access is inherently column-wise, hence not cache friendly. Since AtA includes calls to Strassen for generic matrix multiplications, we also outline a time and space efficient implementation for this algorithm.

6.3.1 AtA in detail

Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ be a rectangular matrix. The idea behind AtA is the following: at each recursive step, matrix $\mathbf{A}$ is divided into four sub-matrices as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix} \quad \begin{aligned} \mathbf{A}_{1,1} &= \mathbf{A}_{0:m_1,0:n_1} \in \mathbb{R}^{m_1\times n_1} \\ \mathbf{A}_{1,2} &= \mathbf{A}_{0:m_1,n_1:n} \in \mathbb{R}^{m_1\times n_2} \\ \mathbf{A}_{2,1} &= \mathbf{A}_{m_1:m,0:n_1} \in \mathbb{R}^{m_2\times n_1} \\ \mathbf{A}_{2,1} &= \mathbf{A}_{m_1:m,n_1:n} \in \mathbb{R}^{m_2\times n_2} \end{aligned} \tag{6.1}$$

being $m_1 \overset{def}{=} \lfloor \frac{m}{2} \rfloor$, $m_2 \overset{def}{=} \lceil \frac{m}{2} \rceil$, $n_1 \overset{def}{=} \lfloor \frac{n}{2} \rfloor$, $n_2 \overset{def}{=} \lceil \frac{n}{2} \rceil$. We address to sub-matrices of a matrix $\mathbf{A}$ as to indexed sub-blocks $(\mathbf{A}_{i,j})$ or with line and column intervals $(\mathbf{A}_{r_1:r_2,c_1:c_2})$.

The product matrix $\mathbf{C} = \mathbf{A}^T\mathbf{A}$ is also split into four sub-matrices, resulting in the following:

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}^T\mathbf{A}_{1,1} + \mathbf{A}_{2,1}^T\mathbf{A}_{2,1} \in \mathbb{R}^{n_1\times n_1}, \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}^T\mathbf{A}_{1,2} + \mathbf{A}_{2,1}^T\mathbf{A}_{2,2} \in \mathbb{R}^{n_1\times n_2}, \\ \mathbf{C}_{2,1} &= \mathbf{A}_{1,2}^T\mathbf{A}_{1,1} + \mathbf{A}_{2,2}^T\mathbf{A}_{2,1} \in \mathbb{R}^{n_2\times n_1}, \\ \mathbf{C}_{2,2} &= \mathbf{A}_{1,2}^T\mathbf{A}_{1,2} + \mathbf{A}_{2,2}^T\mathbf{A}_{2,2} \in \mathbb{R}^{n_2\times n_2}. \end{aligned} \tag{6.2}$$

Both $\mathbf{C}_{1,1}$ and $\mathbf{C}_{2,2}$ consist of two addends that are, in turn, the left hand product of a matrix by its transpose. Hence, four recursive calls are employed to compute the sub-products $\mathbf{A}_{1,1}^T\mathbf{A}_{1,1}$ and $\mathbf{A}_{2,1}^T\mathbf{A}_{2,1}$ to obtain $\mathbf{C}_{1,1}$, and $\mathbf{A}_{1,2}^T\mathbf{A}_{1,2}$ and $\mathbf{A}_{2,2}^T\mathbf{A}_{2,2}$ to obtain $\mathbf{C}_{2,2}$. Since for any matrix $\mathbf{A}$ the product $\mathbf{A}^T\mathbf{A}$ is symmetric, at each recursive step only the lower triangular part of the product matrix is computed, $\text{low}(\mathbf{C}_{i,i})$, $i = 1, 2$. As for component $\mathbf{C}_{2,1}$, in order to compute its two terms in the sum, we implement the generalized Strassen's algorithm for non-square matrices. The sub-matrix $\mathbf{C}_{1,2}$ is equal to $\mathbf{C}_{2,1}^T$, and therefore must not be explicitly computed. In Algorithm 3 we provide the pseudo-code of AtA. The base case occurs when the number of entries of the sub-matrix fits in the cache. In that case, the multiplication is performed by the BLAS function for $\mathbf{A}^T\mathbf{A}$, `?syrk`, where the character `?` represents a generic data type in accordance with standard notation used in manuals, [55]. In Algorithm 3, we also sketch our implementation of Strassen: before the actual recursive Strassen algorithm is called (Strassen), in FastStrassen we conveniently prepare an environment for memory efficiency by pre-allocating the memory for Strassen's algorithm, as explained in Section 6.3.3. The reduced number of multiplications in Strassen's algorithm is achieved by computing more matrix additions. In our implementation of

Strassen, matrix additions are performed by calling the BLAS routine `?axpy` (which performs the vector addition $\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$). The base-case condition in Strassen is analogous to the one of AtA. When the base-case condition holds, we call the BLAS routine `?gemm` for the generic $\mathbf{A}^T\mathbf{B}$ multiplication. To handle odd-sized matrices, we do not implement notorious strategies such as peeling or padding, since these are known for introducing computational and memory overhead. Instead, we manage sums between matrices of discordant size by conveniently applying the BLAS routine `?axpy` for array sums, so that it simulates padding of an extra 0 column or row, by excluding the last row and/or column of a sub-matrix from the sum.

AtA and FastStrassen are designed to be efficient alternatives to the BLAS routines `?gemm` and `?syrk`. Thus, they perform the same operations, respectively $\mathbf{C} = \alpha\mathbf{A}^T\mathbf{B} + \beta\mathbf{C}$ and $\mathbf{C} = \alpha\mathbf{A}^T\mathbf{A} + \beta\mathbf{C}$. However, we avoid introducing the scaling factor β from our algorithms for clarity of exposure, since $\mathbf{C}$ can be simply scaled before applying the algorithms.

Algorithm 3 AtA - Serial

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{C} \in \mathbb{R}^{n\times n}, \alpha \in \mathbb{R}$
Output: Lower triangular part of $\mathbf{C} = \alpha\mathbf{A}^T \cdot \mathbf{A} + \mathbf{C}$

1: **procedure** AtA($\mathbf{A}$, $\mathbf{C}$, α)
2: **if** $m \times n \leq$ cache size **then**
3: $\mathbf{C} \leftarrow \mathbf{C}+$ `blas_?syrk`($\mathbf{A}$, α);
4: return;
5: **else**
6: Initialize pointers to $\mathbf{A}_{i,j}$ and $\mathbf{C}_{i,j}$, $i, j = 1, 2$;
7: AtA ($\mathbf{A}_{1,1}, \mathbf{C}_{1,1}$, α);
8: AtA ($\mathbf{A}_{2,1}, \mathbf{C}_{1,1}$, α);
9: AtA ($\mathbf{A}_{1,2}, \mathbf{C}_{2,2}$, α);
10: AtA ($\mathbf{A}_{2,2}, \mathbf{C}_{2,2}$, α);
11: FastStrassen ($\mathbf{A}_{1,2}$, $\mathbf{A}_{1,1}$, $\mathbf{C}_{2,1}$, α);
12: FastStrassen ($\mathbf{A}_{2,2}$, $\mathbf{A}_{2,1}$, $\mathbf{C}_{2,1}$, α);

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{m\times k}, \mathbf{C} \in \mathbb{R}^{n\times k}, \alpha \in \mathbb{R}$
Output: $\mathbf{C} = \alpha\mathbf{A}^T \cdot \mathbf{B} + \mathbf{C}$

1: **procedure** FastStrassen($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, α)
2: Allocate $\mathbf{M} = \mathbf{0}^{n\times k/2}$;
3: Allocate $\mathbf{P} = \mathbf{0}^{m\times n/2}$;
4: Allocate $\mathbf{Q} = \mathbf{0}^{m\times k/2}$;
5: Strassen($\mathbf{M}$, $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, α);

6.3.2 Computational Complexity

The idea behind Strassen's algorithm is to perform a 2×2 matrix multiplication using 7 multiplications instead of 8, as required by naive matrix multiplication [133]. Nevertheless, Strassen's algorithm involves 18 sums between sub-matrices. The following holds:

Proposition 6.3.1. *The computational complexity of Strassen's algorithm, $T_S(n)$, is the following:*

$$T_S(n) = 7T_S\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \approx 7n^{\log_2 7}. \tag{6.3}$$

Proof. At each recursive step, Strassen's algorithm makes 7 recursive calls and 18 sums. Therefore:

$$\begin{aligned} T_S(n) &= 7T_S\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 = \\ &= 7^k T_S\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 18 \cdot 7^i \left(\frac{n}{2^{i+1}}\right)^2 = \end{aligned}$$

[for $k = \log_2(n)$ and by geometric sums]

$$= n^{\log_2 7} + 6n^2\left(1 - n^{\log_2 \frac{7}{4}}\right) = 7n^{\log_2 7} - 6n^2.$$

□

In Algorithm 3, there are four recursive calls to AtA on basically halved dimensions, two calls to FastStrassen and 3 sums. Thus, we can derive the recurrence function for AtA runtime depending on the input size n:

Proposition 6.3.2. *The computational complexity of* AtA *(Algorithm 3), $T(n)$, is the following (Equation 6.4):*

$$T(n) = 4T\left(\frac{n}{2}\right) + 2T_S\left(\frac{n}{2}\right) + 3\left(\frac{n}{2}\right)^2 \approx \frac{2}{3}T_S(n). \tag{6.4}$$

Proof. AtA makes 4 recursive calls to itself on halved matrix size, 2 calls to Strassen, and 3 sub-matrix sums, hence:

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + 2T_S\left(\frac{n}{2}\right) + 3\left(\frac{n}{2}\right)^2 = \\ &= 4^k T\left(\frac{n}{2^k}\right) + 2\sum_{i=0}^{k-1} 4^i T_S\left(\frac{n}{2^{i+1}}\right) + 3\sum_{i=0}^{k-1} 4^i \left(\frac{n}{2^{i+1}}\right)^2 \end{aligned}$$

[for $k = \log_2 n$, by Prop. 6.3.1 and geometric sums]

$$= \frac{14}{3}n^{\log_2 7} + \frac{3}{4}n^2 \log_2 n - \frac{11}{3}n^2 \approx \frac{2}{3}T_S(n).$$

In particular, by considering only the first two terms of the recurrence relation, the number of multiplications performed by AtA is $2/3n^{\log_2 7} + 1/3n^2$. □

The overall computational complexity of AtA reduces the one of the general matrix multiplication $\mathbf{A}^T\mathbf{A}$, amounting to $n^2(n+1)$, and of Strassen's algorithm naively applied for computing $\mathbf{A}^T\mathbf{A}$, that would require the same number of products as for the general matrix multiplication, and only 16 sums instead of the 18 matrix additions in the original Strassen's formulation.

6.3.3 Space complexity

Algorithm 3 can be implemented with only constant additional memory by replacing lines 11-12 with calls to `?gemm`, but at the cost of losing performance improvement. The implementation we propose here, instead, relies on the FastStrassen function, which allows for the searched reduction of computational complexity.

At each recursive step, pointers to the current portions of **A** and **C** are initialized so that, when the condition for base-case occurs, the matrix multiplications are carried out on the correct sub-matrices of **A**, and stored in the corresponding locations in **C**.

In AtA, Strassen's algorithm for general matrix multiplication is called twice. One drawback of the naive Strassen implementation is the great amount of memory allocated at each recursive step to store the results of the intermediate matrix additions required by the algorithm. In order to avoid frequent memory allocations and releases, we call recursive Strassen (Strassen) on pre-allocated matrices, **M**, **P** and **Q** (FastStrassen). The size of such matrices is sufficiently large to store all intermediate matrix operation results throughout the recursive calls. In fact, given a $n \times n$ matrix, at each recursive step we halve both the dimensions, considering the ceiling of the result when matrices have odd sizes. By doing so, the amount of memory used by the algorithm when the base case is reached is

$$\sum_{i=1}^{\log_2 n} \frac{(n+\log_2 n)^2}{4^i} = (n+\log_2 n)^2 \left(\frac{1}{3} - \frac{4}{3n^2}\right) \leq \frac{n^2}{2} \tag{6.5}$$

which, multiplied by the three supporting matrices **M**, **P** and **Q**, results in a total of $\frac{3}{2}n^2$. Although the overall space complexity of Strassen does not change, we are able to save time for memory allocation at each recursive step. Consequently, the space complexity of AtA is $S(n) = \frac{3}{2}n^2$.

In Section 6.5, we show that Strassen's algorithm benefits from the described strategy for memory allocation.

6.3.4 Cache Complexity

In this section, we show the cache complexity of AtA. We assume the ideal cache model and we denote with M the cache size, and with b the size of the cache line.

Proposition 6.3.3. *The cache complexity of* AtA, $C_{\text{AtA}}(n; M, b)$, *is the same as the cache complexity of Strassen,* $C_S(n; M, b) = \Theta(1 + n^2/b + n^{\log_2(7)}/b\sqrt{M})$, *[56].*

Proof. We prove the thesis by induction. First, we observe that $C_{\text{AtA}}(2; M, b) = 6C_S(1; M, b) \leq 7C_S(1; M, b) = C_S(2; M, b)$. Assuming as inductive hypothesis that $C_{\text{AtA}}(n/2; M, b) \leq C_S(n/2; M, b)$, it holds that:

$$\begin{aligned} C_{\text{AtA}}(n; M, b) &= 4C_{\text{AtA}}(n/2; M, b) + 2C_S(n/2; M, b) \\ &\leq 6C_S(n/2; M, b) \leq 7C_S(n/2; M, b) = C_S(n; M, b). \end{aligned}$$

Furthermore, notice that: $C_S(n/2; M, b) \leq C_{\text{AtA}}(n; M, b) \leq C_S(n; M, b)$. Hence, the thesis holds. □

6.4 Parallel AtA

Our algorithm for the $\mathbf{A}^T\mathbf{A}$ product, AtA, can be conveniently parallelized to work on both shared and distributed-memory systems. We will refer to our shared and distributed-memory algorithms for $\mathbf{A}^T\mathbf{A}$ as AtA-S and AtA-D, respectively. Our parallel implementations of AtA take advantage of the recursive nature of AtA to distribute tasks (and possibly data) to different processes in an efficient way. To do so, an initial phase that implements a scheduler covering the recursion tree of AtA is integrated in both parallel algorithms. In this way, we assign a task to each different parallel process, as we explain in Section 6.4.1. After this preliminary phase, each process knows which sub-problem it has to solve.

Algorithm 4 RecursiveGEMM

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{m\times k}, \mathbf{C} = \mathbf{0}^{n\times k}$
Output: $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$

1: **procedure** RecursiveGEMM($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$)
2: **if** $m \times n + m \times k \leq$ cache size **then**
3: $\mathbf{C} +=$ `blas_?gemm`($\mathbf{A}^T$, $\mathbf{B}$);
4: return;
5: Initialize pointers to $\mathbf{A}_{i,j}$, $\mathbf{B}_{i,j}$ and $\mathbf{C}_{i,j}$, $i, j = 1, 2$;
6: RecursiveGEMM($\mathbf{A}_{1,1}$, $\mathbf{B}_{1,1}$, $\mathbf{C}_{1,1}$);
7: RecursiveGEMM($\mathbf{A}_{2,1}$, $\mathbf{B}_{2,1}$, $\mathbf{C}_{1,1}$);
8: RecursiveGEMM($\mathbf{A}_{1,1}$, $\mathbf{B}_{1,2}$, $\mathbf{C}_{1,2}$);
9: RecursiveGEMM($\mathbf{A}_{2,1}$, $\mathbf{B}_{2,2}$, $\mathbf{C}_{1,2}$);
10: RecursiveGEMM($\mathbf{A}_{1,2}$, $\mathbf{B}_{1,1}$, $\mathbf{C}_{2,1}$);
11: RecursiveGEMM($\mathbf{A}_{2,2}$, $\mathbf{B}_{2,1}$, $\mathbf{C}_{2,1}$);
12: RecursiveGEMM($\mathbf{A}_{1,2}$, $\mathbf{B}_{1,2}$, $\mathbf{C}_{2,2}$);
13: RecursiveGEMM($\mathbf{A}_{2,2}$, $\mathbf{B}_{2,2}$, $\mathbf{C}_{2,2}$);

6.4.1 Preliminary phase: task assignment

Usually, recursive algorithm are parallelized with a fork-join paradigm, according to their natural behaviour: at each recursive call, a new thread is created to accomplish that call. However, repeatedly creating and killing threads introduces a non-negligible overhead, specially when it happens as a nested procedure. A parallelized `for` loop approach can usually improve this thread start-up overhead. For this reason, rather than addressing the problem by distributing recursive calls between newly created threads, we simulate the behaviour of a fork-join algorithm to determine, for each thread, on which sub-matrices it must work. This is particularly useful to generalize our approach to both shared memory and distributed settings.

Building the task tree

To conveniently distribute tasks among P parallel processes collaborating to compute $\mathbf{A}^T\mathbf{A}$, in the first phase of our algorithms, each process builds the recursion

tree of a modified version of AtA, that we shall call AtANaive, and explores a part of it with a breadth-first search (BFS). AtANaive considers classic recursive general matrix multiplication instead of Strassen, and can be easily implemented by modifying lines 11 and 12 in Algorithm 3 to call RecursiveGEMM instead of Strassen. RecursiveGEMM, summarized in Algorithm 4, is a recursive algorithm for the naive general matrix multiplication. The reasons behind this choice will be explained later in this section. We define the *task tree*, denoted with $\mathcal{T}$, to be the sub-tree of the recursion tree of AtA, obtained by spanning the latter with a BFS, that is interrupted as soon as $\mathcal{T}$ counts P leaves, labeled from 0 to $P-1$. Both AtA-S and AtA-D implement the task tree, but with two main differences. The first one concerns the way matrix $\mathbf{A}$ is divided among processes: in the shared-memory approach, matrix $\mathbf{A}$ is tiled so that each thread writes on a specific memory portion, thus avoiding memory collisions and enforcing embarrassing parallelism. In the distributed-memory environment, there is no need to handle data collisions, hence we can assign tasks only focusing on workload balance. The second difference concerns the role of the leaves and of the inner nodes of $\mathcal{T}$: in AtA-D each node of $\mathcal{T}$ corresponds to a task, each assigned to only one process, whereas in AtA-S only leaf nodes correspond to actual computation. The parallel computation in AtA-D begins with performing tasks laying in the leaves of $\mathcal{T}$. In particular, the p-th leaf corresponds to the task that process p has to fulfil. In AtA-D, each leaf task contains directives on both the computational and communication activity that is due to the corresponding process. Specifically, a leaf task t provides the following information:

1. $t.computationType$: Which type of computation process p has to carry out. It can be either a $\mathbf{A}^T\mathbf{A}$ or a $\mathbf{A}^T\mathbf{B}$ multiplication;

2. $t.\mathbf{X}.offset$ and $t.\mathbf{X}.q$, with $\mathbf{X} \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, $q \in \{m, n\}$: The row and column offsets as well as the size of the sub-matrices of $\mathbf{A}$ and $\mathbf{C}$ process p has to work on;

3. $t.father$: The father process that sends sub-matrices of $\mathbf{A}$ to its children (during the distribution phase), and to which process p has to send the result of the task that was assigned to it or, if p is the father, the information on its children' tasks (during result retrieval).

Inner nodes of $\mathcal{T}$ instead, represent tasks concerning data distribution and retrieval, possibly involving sums of sub-matrices of $\mathbf{C} = \mathbf{A}^T\mathbf{A}$, and consequent communication (point 3 of the previous list), and are executed by a subset of processes. In contrast, in AtA-S only leaf nodes of $\mathcal{T}$ correspond to a task, whereas inner nodes are ignored, as no communication is involved. For the same reason, leaf tasks only include information about what kind of computation the corresponding threads have to carry out and on the sub-matrices they have to work on (points 1 and 2 of the previous list).

Load Balancing

The recursion tree of AtA-D is created so that, at each level, given P available processes, $\alpha \cdot P$ processes compute a general $\mathbf{A}^T\mathbf{B}$ matrix multiplication; for the

remaining $(1-\alpha)\cdot P$ processes, a task for a $\mathbf{A}^T\mathbf{A}$ multiplication is assigned to them. Here, $\alpha \in (0,1)$ is a parameter for balancing the workload among distributed processes, as the computational complexity of a $\mathbf{A}^T\mathbf{A}$ product is lower than the one of $\mathbf{A}^T\mathbf{B}$. Since the computational cost of AtA is $2/3$ the one of FastStrassen, and because in AtA there are four recursive calls to AtA itself, and two calls to FastStrassen, the optimal value for the balancing factor should be set to $\alpha = 3/7$. However, since the task tree $\mathcal{T}$ is built by calling RecursiveGEMM instead of FastStrassen, the number of multiplications carried out in $\mathcal{T}$ to perform $\mathbf{A}^T\mathbf{B}$ is twice the one needed to compute $\mathbf{A}^T\mathbf{A}$. In accordance, we set $\alpha = 1/2$. This task division is repeated recursively at each level, by progressively decreasing the number of available processes, P. As anticipated, the number of recursive parallel steps depends on P, and also by the scalar α. In particular, for $\alpha = 0.5$, the number of parallel levels in the task tree, ℓ is given by the following expression:

$$\ell(P) = \begin{cases} 0 & \text{if } P = 1 \\ 1 & \text{if } 2 \leq P \leq 6 \\ 1 + k + \operatorname{sign}\left(\frac{P}{4} \mod 8^{\max\{k;1\}}\right) & \text{if } P > 6 \end{cases} \tag{6.6}$$

where $k = \max\left\{k \in \mathbb{N} : \frac{P/4}{8^k} \geq 1\right\}$ and $\operatorname{sign}(x)$ is the sign function, returning 0 for $x = 0$ and 1 for $x > 0$. Indeed, when AtA-D is called on P processes, $P/2$ of them are going to compute $\mathbf{C}_{2,1}$; out of them, $P/4$ processes compute $\mathbf{A}_{1,2}^T\mathbf{A}_{1,1}$, whereas the remaining $P/4$ are in charge for $\mathbf{A}_{2,2}^T\mathbf{A}_{2,1}$ (see Equation 6.2). These tasks are in turn distributed among 8 processes each, recursively (corresponding to the eight

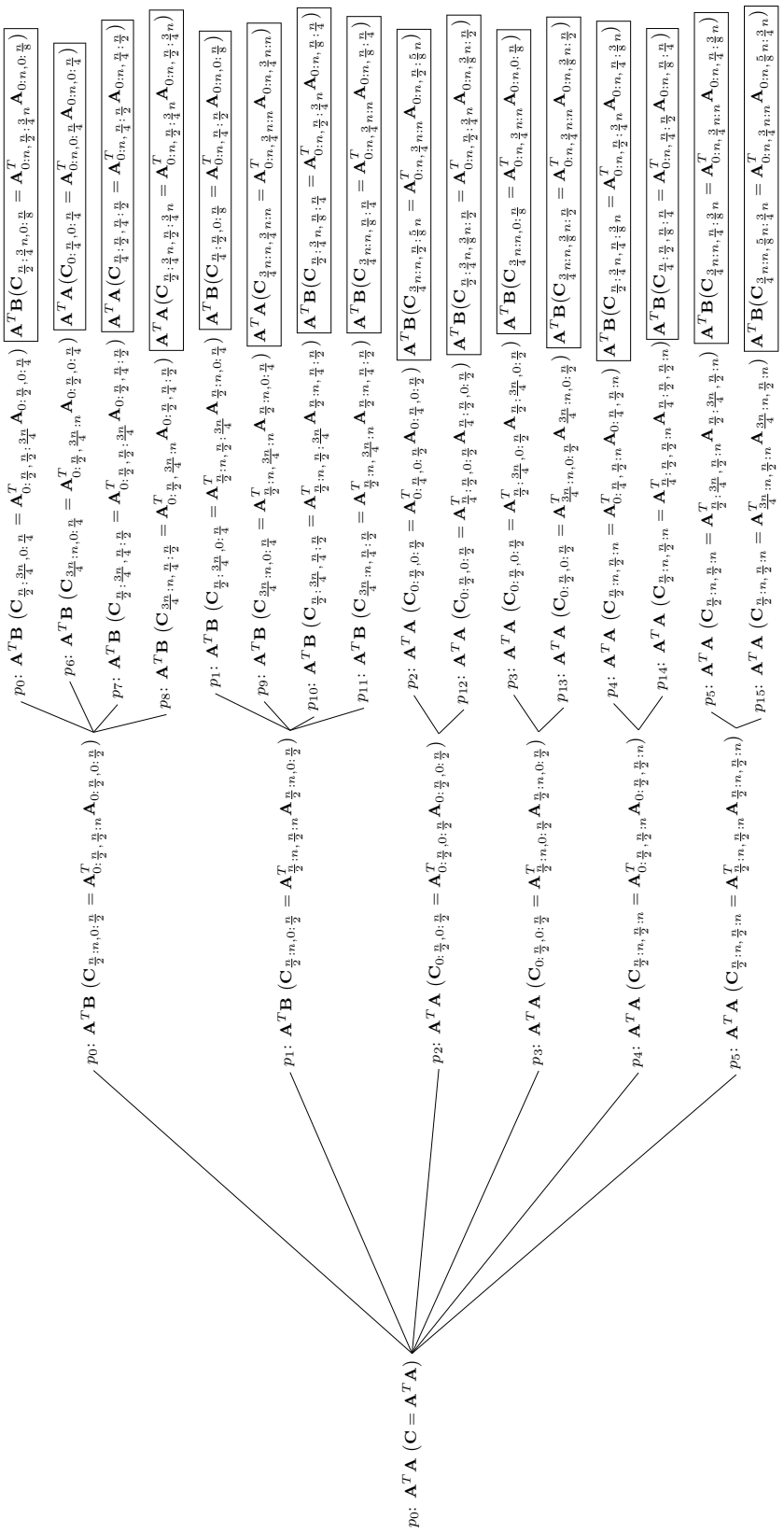


Figure 6.4.1. A tree of 16 processes distributing a matrix $A \in \mathbb{R}^{n\times n}$. Boxed labels on the right-hand side are the leaf nodes of the tree generated by AtA-S, corresponding to computation tasks assigned to corresponding processes in the left-hand size leaf labels.

recursive calls of RecursiveGEMM). This splitting is repeated until possible (i.e., until $P/4/8^k \geq 1$). If by doing so, all $P/4$ processes are used (i.e., $P/4$ is a multiple of 8^k, for some k), all processes work on equally sized matrices. Otherwise, some processes will further split their tasks to smaller matrices, resulting in an additional parallel level. We say that the last parallel level is *complete* when all leaves corresponding to $\mathbf{A}^T\mathbf{A}$ tasks are grouped in bunches of 6 siblings, and when all leaves corresponding to $\mathbf{A}^T\mathbf{B}$ tasks are grouped in bunches of 8 siblings.

The task tree for AtA-S is quite different. In order to avoid concurrent overlapping writes, input matrices are tiled in horizontal and vertical blocks, as depicted in Figure 6.4.2. This way, we ensure that each thread computes a different $\mathbf{C}_{i,j}$. With this new scheme, we make three recursive calls to AtA (instead of 6) and four recursive calls to FastStrassen (instead of 8). Therefore, the number of parallel levels in AtA-S, given P threads, is the following:

$$\ell(P) = \begin{cases} 0 & \text{if } P = 1 \\ 1 & \text{if } P = 2, 3 \\ 1 + k + \text{sign}\left(\frac{P}{2} \mod 4^{\max\{k;1\}}\right) & \text{if } P > 3 \end{cases} \tag{6.7}$$

with $k = \max\left\{k \in \mathbb{N} : \frac{P/2}{4^k} \geq 1\right\}$. In Figure 6.4.1, we show an example of the task tree with 16 processes for AtA-D, and the leaf nodes of the task tree for AtA-S (boxed).

Naive matrix multiplication over Strassen

In our parallel algorithms, we do not rely on Strassen for general $\mathbf{A}^T\mathbf{B}$ matrix multiplication when building the recursion tree, that instead is created by simulating AtANaive. This is done with the goal of optimizing the resources of distributed architectures, as the naive general matrix multiplication algorithm does not allocate the additional memory required by Strassen, resulting in a faster memory management. Furthermore, Strassen's algorithm would possibly cause a hardly manageable workload unbalance between processes implementing an $\mathbf{A}^T\mathbf{A}$ multiplication, and those that would be in charge of computing the intermediate matrix sums appearing in Strassen's algorithm. However, Strassen's algorithm can still be used at leaf-level computation.

6.4.2 Shared-memory AtA

A shared-memory parallel algorithm for the $\mathbf{A}^T\mathbf{A}$ multiplication represents a faster alternative to the distributed-memory approach when run on local machines, rather than on clusters of nodes.

For our shared-memory implementation of the $\mathbf{A}^T\mathbf{A}$ product, we rely on OpenMP to efficiently distribute the workload between threads. Each thread simulates the recursion of AtANaive as described in Section 6.4.1. The workload is distributed so that each thread writes in a different memory location, hence there is no need of handling data hazards of any kind. Instead, the problem is divided in a fashion that makes it embarrassingly parallel. We shall refer to our multi-threaded algorithm for $\mathbf{A}^T\mathbf{A}$ as to AtA-S.

AtA-S in detail

Let us denote with P the number of available threads. Our algorithm for multi-threaded machines, AtA-S, can be divided into two phases. During the first phase, one task is assigned to each thread by simulating the recursion of AtANaive, as described in Section 6.4.1. In order to prevent memory collisions and to achieve embarrassing parallelism, tasks are organized so that each thread writes on a different and disjoint memory location. This is done by dividing the resulting matrix $\mathbf{C}$ into four blocks, as shown in Section 6.3, whereas $\mathbf{A}$ is tiled vertically or horizontally, instead of in 2×2 blocks (see Figure 6.4.2). This procedure avoids concurrent writing management and relies on the equality:

$$\mathbf{C}_{i,j} = \mathbf{A}_{i,1}\mathbf{B}_{1,j} + \mathbf{A}_{i,2}\mathbf{B}_{2,j} = \mathbf{A}_{i,*}\mathbf{B}_{*,j}, \tag{6.8}$$

for $i, j = 1, 2$. Such instruction and data assignment allows for a faster execution, since threads never need to synchronize.

During the second phase of AtA-S, each thread retrieves its task from the tree $\mathcal{T}$, specifying which routine (either AtA or FastStrassen) the corresponding thread must call, and on which sub-matrices of $\mathbf{A}$ and $\mathbf{C}$ it must operate. Since the sub-problems are disjoint, at the end of the computation each thread only needs to synchronize with the others, then the algorithm stops. In Algorithm 5 we provide the pseudo-code of AtA-S.

Algorithm 5 AtA-S- Shared

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$
Output: Lower triangular part of $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{A}$

```
1: procedure AtA-S(A)
2:     Generate tree T;
3:     parfor each leaf-node v of T do
4:         Get task t from node v;
5:         if t.computationType = A^T A then
6:             C_{t.C.offset} = A^T A(A_{t.A.offset});
7:         else if t.computationType = A^T B then
8:             C_{t.C.offset} = A^T B(A_{t.A.offset}, A_{t.B.offset});
9:     end parfor
```

Computational Complexity of AtA-S

?? We study the time complexity $T(n, P)$ of AtA-S to perform the multiplication $\mathbf{A}^T\mathbf{A}$ on a $n \times n$ matrix $\mathbf{A}$ and distributing the workload between P processes.

At first, the algorithm needs to generate the task tree and each process has to retrieve its task. These procedures have the same complexity as a BFS visit on a tree with P leaves, hence $O(P)$.

The time complexity of the second step corresponds to the one of the most expensive leaf task, which appears at the end of a path of RecursiveGEMM calls.

At level l, the size of the product matrix $\mathbf{C}$ is reduced to a block of size $n/2^l \times n/2^l$, resulting from a multiplication between $n/2^l \times n$ and $n \times n/2^l$ matrices. Thus, the total complexity is reduced by $4^{\ell(P)}$, being $\ell(P)$ the number of levels in the task tree. Hence the total complexity of the algorithm is:

$$T(n, P) = O(P) + O\left(\frac{1}{4^{\ell(P)}} n^{\log_2 7}\right). \tag{6.9}$$

Notice that $\ell(P)$ is a discrete, non injective function. Hence, especially with few processes, the speedup behaves like a step function. Despite this behaviour, $\ell(P) \approx \log_4 P$, meaning with large number of processes we achieve a theoretical linear speedup.

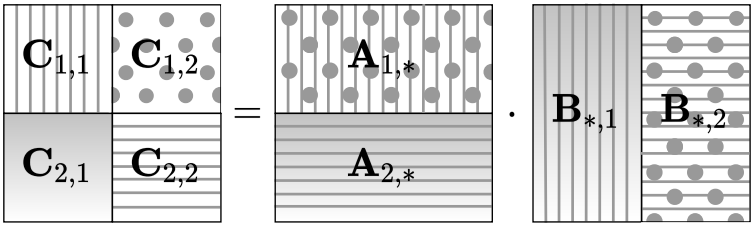


Figure 6.4.2. Example of multiplication with vertical/horizontal tiling, instead of the standard 2×2 blocks.

6.4.3 Distributed-memory AtA

Modern computers are equipped with an ever-increasing number of cores inside CPU chips. However, when it comes to massive volumes of data, computationally intensive tasks such as matrix multiplication are simply prohibitive, even for the most recent 16- or 32-cores chipsets, and even with hyper-threading capabilities. Distributed parallelism plays a crucial role in this setting, as it allows to distribute the workload between multiple machines. In such an environment, providing fast distributed algorithms for operations in Linear Algebra, including $\mathbf{A}^T\mathbf{A}$ multiplication, is a key task to limit bottlenecks.

In this section, we describe a distributed algorithm for $\mathbf{A}^T\mathbf{A}$, that works for any matrix size and with arbitrarily many processes and cores. We shall refer to this algorithm as AtA-D. AtA-D follows a distribute-compute-retrieve paradigm, as initially the input matrix $\mathbf{A}$ is stored on the root process only, and distributed to other processes according to their tasks. Finally, the resulting matrix $\mathbf{C} = \mathbf{A}^T\mathbf{A}$ is retrieved back by the root process. We implement a parallel communication scheme to limit data transfer overhead.

AtA-D in detail

Let P be the number of distributed processes. In AtA-D, each process p first builds the task tree $\mathcal{T}$ as described in Section 6.4.1. To understand in detail how $\mathcal{T}$ is used in AtA-D, we shall refer to the example of Figure 6.4.1. As we said, each node represents a task, but only tasks contained in leaf nodes correspond to an actual matrix multiplication. Inner nodes instead represent tasks assigned only to the

fathers of the nodes branching out of them, and they are necessary to retrieve and combine the portions of the result matrix scattered among different processes, and eventually to send them, level by level, up to the root process, p_0. In the example of Figure 6.4.1, $\mathcal{T}$ is the task tree for $P = 16$ processes on a square matrix. Leaf nodes are generated so that processes p_0, p_1 and $p_6 \ldots, p_{11}$ share the workload to compute $\mathbf{C}_{2,1}$. The remaining half of the processes is devoted to compute $\mathbf{C}_{1,1}$ and $\mathbf{C}_{2,2}$. If the number of distributed processes is not enough to make a complete level, as in this example, instead of calling multiple tasks on different tiles of the matrices, processes perform either a $\mathbf{A}^T\mathbf{A}$ or a $\mathbf{A}^T\mathbf{B}$ operation on vertically and horizontally tiled sub-matrices at the leaf-level. For instance, observe the first batch of sibling-leaves in Figure 6.4.1. To compute $\mathbf{C}_{\frac{n}{2}:n,0:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{2}}$, AtANaive would perform 8 recursive calls to $\mathbf{A}^T\mathbf{B}$; in AtA-D, each of these calls is served by one distributed process, if available. When this is not the case, as in the example that we are considering, processes p_0, p_6, p_7, p_8 divide $\mathbf{A}_{0:\frac{n}{2},\frac{n}{2}:n}$ and $\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{2}}$ in vertical tiles so as to compute the related portions of $\mathbf{C}$ as depicted in Figure 6.4.2.

When the computation is over, partial results are collected by the fathers of each group of siblings (processes p_i, $i = 0, \ldots, 5$ in the example of Figure 6.4.1). This operation is iterated by traversing the tree up to its root, p_0, and allows for a convenient parallel communication reducing data transfer overhead. In order to optimize the communication and to reduce the exchanged data volume, we encode the sub-matrices resulting from $\mathbf{A}^T\mathbf{A}$ operations as packed lower triangular matrices. Nevertheless, the entire operation, once it returns to the root process, still produces a standard square matrix. In Algorithm 6, we provide the pseudocode of AtA-D. In line 11, if the process has to fulfill a $\mathbf{A}^T\mathbf{B}$ task, it sends to its father the entire sub-matrix $\mathbf{C}_{t.\mathbf{C}.offset}$; otherwise, it only sends low($\mathbf{C}_{t.\mathbf{C}.offset}$). In lines 7 and 9, $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{B}$ may refer to AtA or `blas_?syrk`, and to FastStrassen or `blas_?gemm`, respectively. As we shall see in Section 6.5, the real benefit of using our implementation of AtA and FastStrassen arises on matrices with larger size, therefore they are favourable when handling larger volumes of data.

Computational and Communication Complexity of AtA-D

In contrast to parallel algorithms for distributed matrices, AtA-D does not include any communication between processes at computation time, as the input matrix is scattered among distributed processes so that they own the exact portions of $\mathbf{A}$ on which they have to operate. In AtA-D, the computational cost on a matrix of size n and with P processes, $C(n, P)$, depends on the number of recursive levels that can be layered with the available resources, having fixed the load balancing parameter α. For $\alpha = 0.5$, the computational complexity of AtA-D is given by the time for computing $\mathbf{A}^T\mathbf{B}$ on matrices of size at most $n/2^{\ell(P)} \times n/2^{\ell(P)-1}$, that is $O\left(\left(n/2^{\ell(P)}\right)^2 \cdot n/2^{\ell(P)-1}\right)$, where $\ell(P)$ is the number of parallel levels defined in Equation 6.6.

We can express the communication cost for matrix distribution and result retrieval in terms of latency and bandwidth costs of a distributed algorithm, denoted with $L(n, P)$ and $BW(n, P)$, respectively, using the same definitions introduced in [15] and adopted also in [86]. Latency cost is the communicated message count, whereas

bandwidth is expressed in terms of communicated words count. Messages and words counts are computed along the critical path of the distributed algorithm, as defined in [145]. In AtA-D, this corresponds to the sequence of communication operations carried out by the root process p_0. The number of exchanged messages sent along the critical path is given by the number of children of process p_0, at each level. In our implementation, after the first parallel level, p_0 works on a $\mathbf{A}^T\mathbf{B}$ task and shares its workload with 7 other processes at each parallel level. When the compute phase is over, at each level $l \in \{2, \ldots, \ell(P)\}$ process p_0 collects partial results from its (at most) seven children; at level $l = 1$, it retrieves the entire matrix $\mathbf{C} = \mathbf{A}^T\mathbf{A}$ by combining together the results of its five siblings. This operation is carried out both for data distribution and result collection. Hence, $L(n, P) = O(2[7 \cdot (\ell(P) - 1) + 5])$.

During the data distribution phase, message size (i.e., portions of input matrix $\mathbf{A}$) decreases when descending from the root down to the leaves of $\mathcal{T}$. In the first level, p_0 distributes two matrices of size $n/2 \times n/2$ to the the other process that is in charge to carry out $\mathbf{A}^T\mathbf{B}$ tasks, and one sub-matrix of the same size to each of its four siblings that have to compute $\mathbf{A}^T\mathbf{A}$. For each level $l \in \{2, \ldots, \ell(P)\}$, the root process sends matrices of size $n/2^l$ to at most 7 other processes. Hence, during the distribution phase, $BW(n, P)$ is $O(5\,(n/2)^2 + 7 \cdot \sum_{l=2}^{\ell(P)} (n/2^l)^2) = O(5(n/2)^2 + 7/12 n^2(1 - 1/4^{\ell(P)-2}))$. During the result retrieval phase, when traversing the path from leaves up to the root, message size (i.e., portions of resulting matrix $\mathbf{C}$) increases. In this phase, similar considerations as for the distribution phase hold. By considering that processes sending symmetric portions of $\mathbf{C}$ only store its lower triangular (low($\mathbf{C}$)), it holds that the bandwidth during this phase amounts to $O((n/2)^2 + 4(n(n+2)/8) + 7 \cdot \sum_{l=2}^{\ell(P)} (n/2^l)^2) = O((n/2)^2 + n(n+2)/2 + 7/12 n^2(1 - 1/4^{\ell(P)-2})$. Hence, the total bandwidth is $BW(n, P) \leq 6(n/2)^2 + \frac{n(n+2)}{2} + 7/6 n^2(1 - 1/4^{\ell(P)-2})$. From this analysis, we see that computation has the prominent role in time complexity $T(n, P) = C(n, P) + L(n, P) + BW(n, P)$. This fact will be confirmed by our experimental results, presented in Section 6.5, where we see how increasing the matrix sizes provides an always increasing benefit in using the distributed algorithm, proving that communication cost $L(n, P) + BW(n, P)$ is absorbed by the computational cost, $C(n, P)$, for growing values of n.

6.5 Performance Evaluation

We evaluate the performance of our algorithms with an extensive set of experiments over multiple benchmarks.

6.5.1 Experimental Setup

All tests reported in this section were run on TeraStat[1], a cluster of 12 compute nodes, each equipped with 2 sockets of Intel Xeon E5-2630v3 8 cores, 2.4 Ghz, 4 GB RAM per core.

We test our algorithms and benchmark solutions on square and tall matrices, generated randomly. We carry out experiments in both single and double floating point precision, to highlight the fact that our algorithm achieves good performance in both settings.

[1] https://www.dss.uniroma1.it/en/node/6554

Algorithm 6 ATA-D- Distributed

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$
Output: Lower triangular part of $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{A}$

```
1:  procedure ATA-P(A)
2:      Generate tree T
3:      for each node v of T in the path from my leaf to the root do
4:          Get my task t from node v;
5:          if v is a leaf then
6:              if t.computationType = A^T A then
7:                  C_{t.C.offset} = A^T A(A_{t.A.offset});
8:              else if t.computationType = A^T B then
9:                  C_{t.C.offset} = A^T B(A_{t.A.offset}, A_{t.B.offset});
10:         if t.father ≠ my ID then
11:             Send C_{t.C.offset} to t.father;
12:         else
13:             Receive C_{children.t.C.offset} from my children;
14:             Sum over the sub-matrices and store result in C;
```

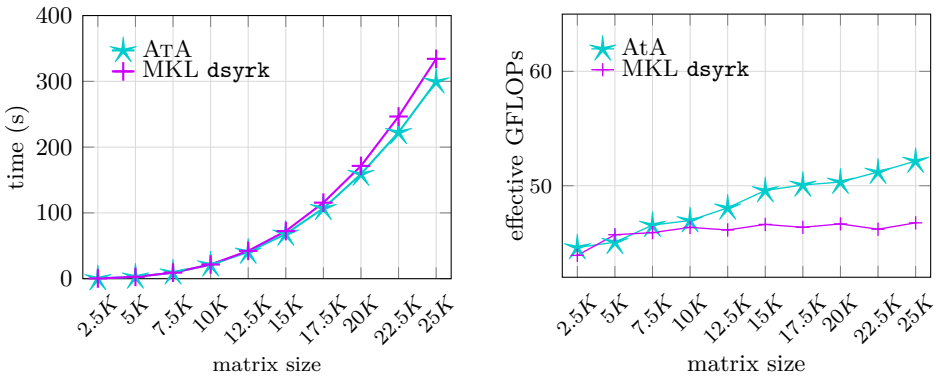
In the tests, we exploit the Intel Math Kernel Library (MKL) both by integrating BLAS routines for basic matrix operations, and for the validation of the proposed algorithms through performance comparisons with shared and distributed memory parallel benchmark solutions. MKL is a framework that includes routines and functions optimized for Intel and compatible processor-based computers, and provides C/C++ interfaces and the acceleration of libraries for Linear Algebra (including BLAS and ScaLapack) within several third-party math libraries. [55, 142].

6.5.2 Metrics

To compare the performance of our algorithms against benchmark methods, we use the average elapsed time in seconds, and the effective GFLOPs. Effective GFLOPs is a measure for comparing classical and fast matrix multiplication algorithms. For classical algorithms, which perform $2n^3$ floating point operations, Equation 6.10 gives the actual GFLOPs; for fast matrix multiplication algorithms, it gives the performance relative to classical algorithms, but does not accurately represent the number of floating point operations performed [37]. For fair comparisons, we calculate the metrics as:

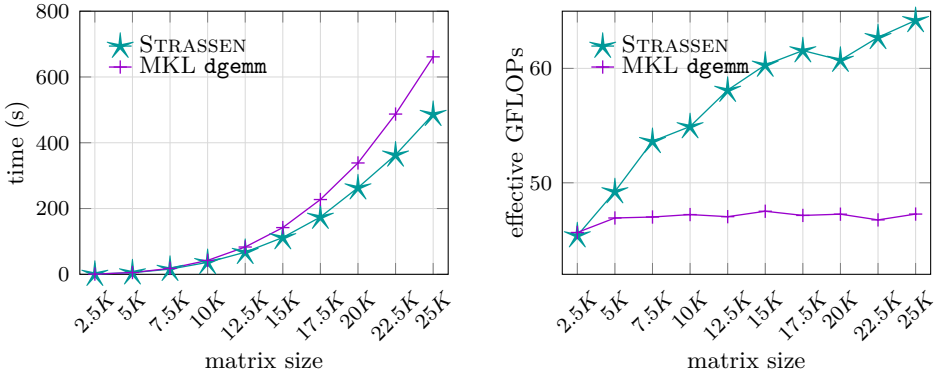
$$\text{effective GFLOPs} = \frac{rn^3}{\text{execution time in seconds} \cdot 10^9} \tag{6.10}$$

where $r = 1$ when we test algorithms specifically built for the $\mathbf{A}^T\mathbf{A}$ product, whereas $r = 2$ when algorithms for the general matrix multiplication are tested.



(a) Elapsed time. **(b)** Effective GFLOPs.

Figure 6.5.1. AtA vs Intel MKL `dsyrk`



(a) Elapsed time. **(b)** Effective GFLOPs.

Figure 6.5.2. FastStrassen vs Intel MKL `dgemm`.

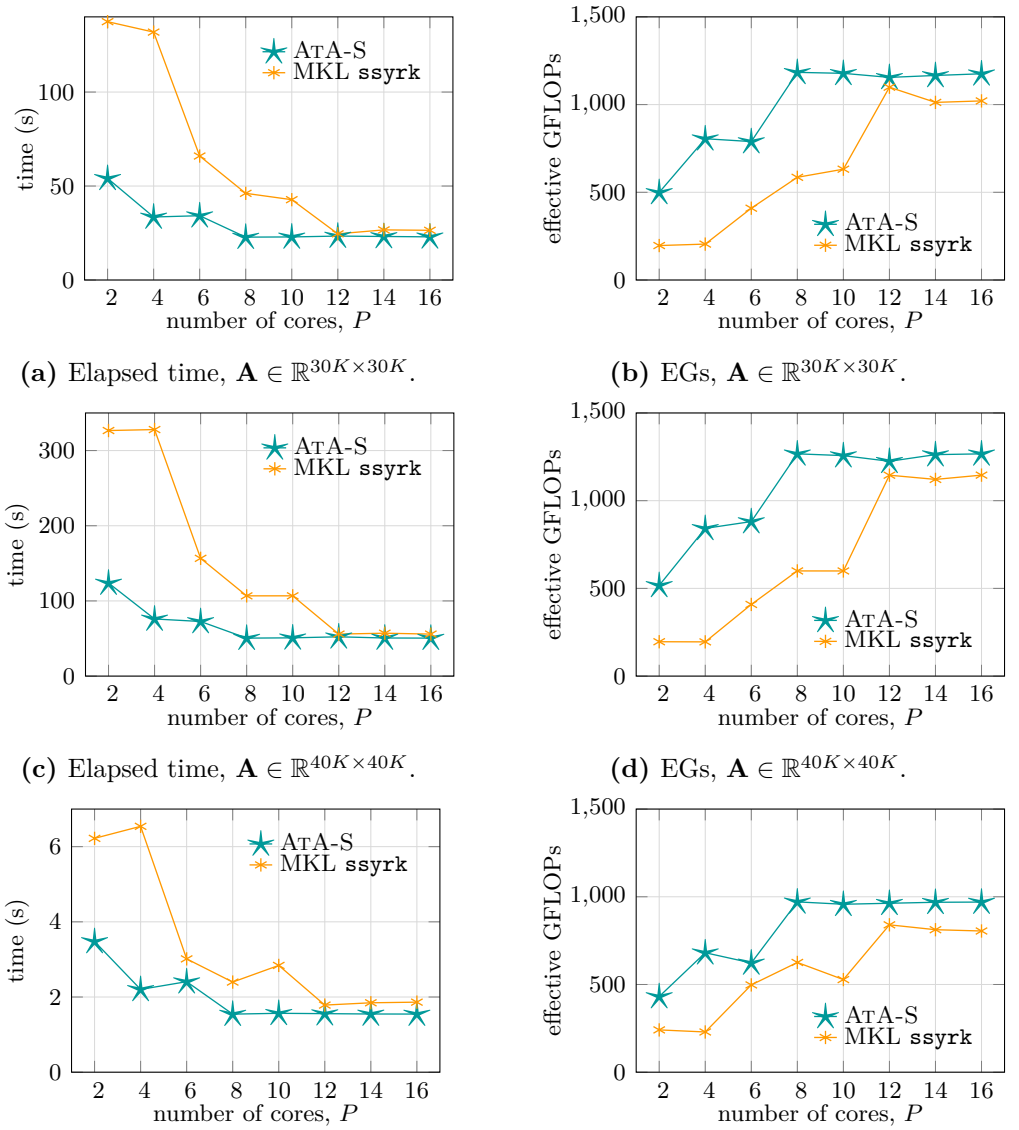
6.5.3 Sequential

Figures 6.5.1 and 6.5.2 show the execution time and effective GFLOPs of the sequential AtA and FastStrassen routines, respectively. Their performance is compared to the Intel MKL counterparts: `dsyrk` and `dgemm`. The experiments are carried out on matrices of growing matrix size (from $2.5 \cdot 10^3$ to $2.5 \cdot 10^4$), and run on a single Intel core. The time difference between our solutions and the ones implemented by Intel MKL grows with the matrix size, reflecting the lower computational cost of our approach. Figure 6.5.2 shows how Strassen's algorithm benefits from the pre-memory-allocation strategy described in Section 6.3.3.

6.5.4 Shared memory

For evaluating the shared memory parallel implementation of the $\mathbf{A}^T\mathbf{A}$ product, AtA-S, we compare it against the Intel MKL implementation of the BLAS routine `ssyrk`, for single precision symmetric rank-K update. For both methods, we always use a 16 thread setup, and we analyse the execution time and the effective GFLOPs (Equation 6.10 with $r = 1$) while varying the number of available cores. In light of

the sequential experiments shown in Figures 6.5.1 and 6.5.2, we compare AtA-S and MKL `ssyrk` on larger matrices, where tests highlight more interesting results. In particular, we run experiments on square matrices of size $3 \cdot 10^4 \times 3 \cdot 10^4$, $4 \cdot 10^4 \times 4 \cdot 10^4$ and on tall matrices of size $6 \cdot 10^4 \times 5 \cdot 10^3$.



(a) Elapsed time, $\mathbf{A} \in \mathbb{R}^{30K \times 30K}$.

(b) EGs, $\mathbf{A} \in \mathbb{R}^{30K \times 30K}$.

(c) Elapsed time, $\mathbf{A} \in \mathbb{R}^{40K \times 40K}$.

(d) EGs, $\mathbf{A} \in \mathbb{R}^{40K \times 40K}$.

(e) Elapsed time, $\mathbf{A} \in \mathbb{R}^{60K \times 5K}$.

(f) EGs, $\mathbf{A} \in \mathbb{R}^{60K \times 5K}$.

Figure 6.5.3. Experimental results of AtA-S and Intel MKL BLAS `dsyrk` in terms of elapsed time in seconds (first row) and effective GFLOPs (second row), varying the number of available cores S on fixed matrix sizes and with a 16 threads configuration.

Figure 6.5.3 summarizes our results. As anticipated by the study of the computational complexity, the execution time is reduced by $1/4$ at each complete parallel level. Figures 6.5.3a-6.5.3e show how our algorithm can compete with the MKL implementation when the core availability is large, and that significantly outperforms the Intel implementation in the $P \leq 10$ cores setup. Furthermore, we show

in Figures 6.5.3b-6.5.3f that AtA-S is capable not only of accomplishing a large amount of floating point operations per second, but also that its performance growth rate is consistent with the step-wise behaviour of the time complexity studied in Section **??**. This justifies sporadic thinnings in performance gap between the two methods. From Figure 6.5.3, we can observe that the performance of both methods stall when more than 8 cores are used. Indeed, multi-threaded MKL automatically chooses the optimal number of threads (in our architecture, this corresponds to 16 threads). For a fair comparison, we use the same setup in AtA-S. Performance scales with the number of available cores, but, when hyper-threading is enabled, 8 cores are enough to reach the 16-thread plateau. Therefore, performance cannot increase significantly for $P > 8$.

6.5.5 Distributed memory

To complete our performance evaluation, we also compare our implementation for distributed architectures of AtA, AtA-D, with fast distributed algorithms for matrix multiplication. We recall that AtA-D differs from standard methods for distributed matrix multiplication, as it does not perform computations on distributed matrices. Instead, in AtA-D the input matrix $\mathbf{A}$ is only stored by the root process, p_0, that first distributes it among other processes cooperating to perform the $\mathbf{A}^T\mathbf{A}$ product, and then collects the partial result of each process to combine them. This approach makes our method unsuitable for distributed chains of operations, since for every operation, the matrix must be repeatedly scattered and gathered back, thus introducing communication overhead, but our results highlight that it is an efficient alternative for distributing single $\mathbf{A}^T\mathbf{A}$ operations. At the current state-of-the-art, there are a variety of methods for multiplying distributed matrices, but in the most recent literature there are three algorithms which stand out:

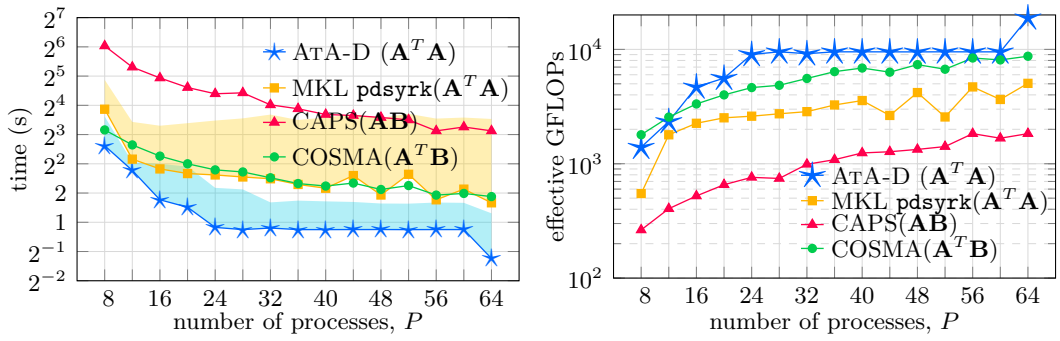
1. Intel MKL ScaLapack `p?syrk`: the Math Kernel Libraries (MKL) are an optimized collection of tools for sequential and parallel computing in Numerical Linear Algebra. In particular, ScaLapack is library of high-performance dense Linear Algebra routines for parallel distributed-memory machines. In ScaLapack, distributed processes are organized in 2D grids of size $m_P \times n_P = P$. For each value of P, we set optimal m_P and n_P by calling `MPI_Dims_create`. We analyse the execution time required to perform the $\mathbf{A}^T\mathbf{A}$ matrix multiplication by the built-in ScaLapack function `pdsyrk`, and the time to retrieve the result of the operation.

2. CAPS[2]: the Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication [13] is a distributed algorithm for general square matrix multiplications $\mathbf{AB}$. Soon after CAPS, the same authors proposed CARMA [37], that also handles rectangular matrices. Nevertheless, it was not possible to test this method as it relies on Cilk Plus, a tool for parallel computing now marked as deprecated [111].

3. COSMA[3]: differently from CAPS, this communication-optimal algorithm for

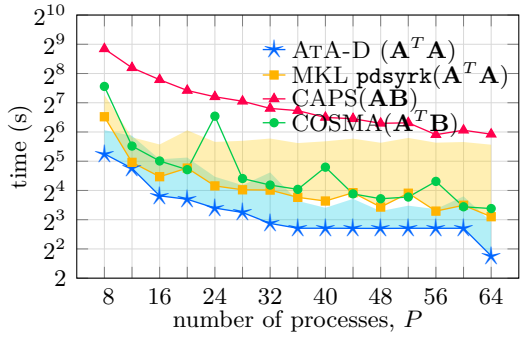
[2] https://github.com/lipshitz/CAPS/
[3] https://github.com/eth-cscs/COSMA

general matrix multiplication does not rely on Strassen's algorithm, instead, it uses red-blue pebble game to precisely model the matrix multiplication dependencies. In [86], the authors show that COSMA outperforms all previously proposed frameworks for general matrix multiplication. It also works for multiplication on transposed matrices, and therefore we test it to perform $\mathbf{A}^T\mathbf{B}$ products.

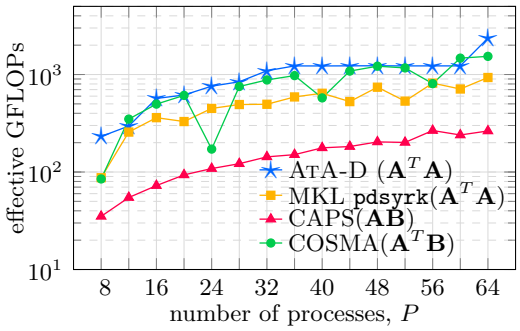


(a) Elapsed time. $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{10K \times 10K}$.

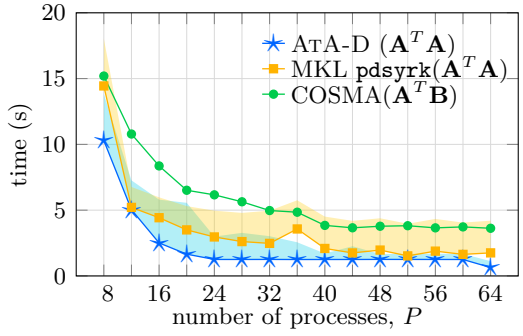
(b) EGs. $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{10K \times 10K}$.



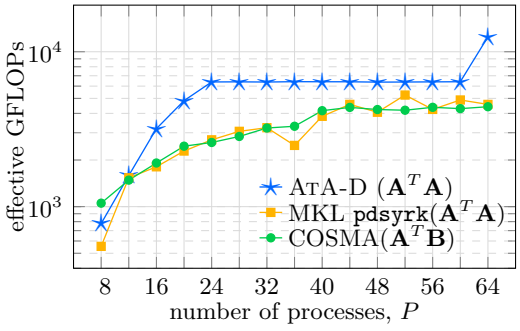
(c) Elapsed time. $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{20K \times 20K}$.



(d) EGs. $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{20K \times 20K}$.



(e) Elapsed time. $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{60K \times 5K}$.



(f) EGs. $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{60K \times 5K}$.

Figure 6.5.4. Experimental results of AtA-D, Intel MKL ScaLapack pdsyrk, CAPS and COSMA in terms of elapsed time in seconds (first row) and effective GFLOPs (second row), varying the number of distributed processes P on fixed matrix sizes.

To simulate massively distributed architectures, in our experiments, we reserve only one core per distributed process. As a consequence, each process has small

n	SM (16 cores)	DM (96 cores)	Speedup
30K	45.16 s	21.24 s	2.13
40K	106.19 s	43.96 s	2.42
50K	221.63 s	81.77 s	2.71
60K	863.32 s	129.08 s	6.69

Table 6.5.1. Comparing shared memory (SM) and distributed memory (DM) implementations of $\mathbf{A}^T\mathbf{A}$ on very large square $n \times n$ matrices.

memory availability (4GB RAM/core).

The results of our experiments for the distributed-memory solution are shown in Figure 6.5.4. In Figures 6.5.4a,6.5.4c and 6.5.4e, marked lines represent the compute time of all considered methods. The shaded areas above the curves describing AtA-D and `pdsyrk` represent the additional time required for communication, i.e., for retrieving the resulting matrix to the root process. We consider two groups of square matrices, having size 10^4 and $2 \cdot 10^4$ (Figures 6.5.4a, 6.5.4b and 6.5.4c, 6.5.4d, respectively), and one set of tall matrices of size $6 \cdot 10^4 \times 5 \cdot 10^3$. Because CAPS does not operate on rectangular matrices, we could not test it on the latter set of experimental configurations. As we can observe from Figure 6.5.4, scalability of AtA-D is nonlinear and it rather follows an almost-stepwise trend with respect to P. This is a consequence of Equation 6.6, that shows how some values of P allow for a more effective and balanced workload between processes. This is evident for small values of P (when a greater availability of processes weighs significantly on the workload of each process), as well as for $P = 64$. Despite the different nature of the parallelism implemented in AtA-D with respect to the benchmark methods analysed in this section, our experiments corroborate the efficiency of the task distribution implemented in AtA-D.

Finally, in order to study the scalability of AtA-D with respect to AtA-S, and to validate the possibility of integrating the two methods, we compare AtA-S and AtA-D on very large matrices of growing size and report results in Table 6.5.1. AtA-S works on 16 cores with 16 threads, whereas AtA-D works on 6 distributed nodes, each equipped with 16 cores, for a total of 96 cores. Each node executes a distributed process calling either AtA-S for $\mathbf{A}^T\mathbf{A}$-type products, or parallel MKL `dgemm` for $\mathbf{A}^T\mathbf{B}$-type multiplications. The times reported in Table 6.5.1 for AtA-D also include communication time (for distributing data and collecting results). Speed-up is computed as T_{SM}/T_{DM}, where T_{SM} and T_{DM} are the execution times of the shared and the distributed-memory algorithms, respectively. In accordance with our computational and communication cost analysis (Section 6.4.3), the speed-up of AtA-D over AtA-S increases when the size of the input matrix increases, as the computation cost overwhelms the communication overhead. Furthermore, the shared-memory implementation suffers when considering larger matrices, since frequent memory access slows down execution (two $60K \times 60K$ matrices require 57 GB out of the 64 GB available on the test machine), consequently decreasing performance. This is highlighted by the results of Table 6.5.1, where we can observe

that $60K \times 60K$ matrices require high computation time, dominated by time for memory management.

6.6 Conclusions

In this chapter we proposed AtA, an algorithm for the $\mathbf{A}^T\mathbf{A}$ product, that reduces the computational complexity of commonly employed algorithms, and that is conveniently implementable in practice on matrices defined on arbitrary domains and of any size and aspect ratio. The computational cost of AtA benefits from the fast matrix multiplication introduced by Strassen's algorithm, and is cache-oblivious. We show that AtA can be efficiently implemented in shared and distributed memory environments. In the shared memory implementation of AtA, tasks are assigned to parallel threads so that they work in perfect parallelism. Our theoretical analysis is supported by experiments that prove the excellent performance of our implementations in comparison with state-of-the-art counterparts.

Chapter 7

A Hybrid Solver for Quasi-Block Diagonal Linear Systems

In this chapter we present a linear solver for a class of *sparse* linear systems that we call *Quasi-Block Diagonal (QBD)*. The solver implements a preconditioned version of the Jacobi iterative algorithm for linear systems on a system supporting a hybrid shared and distributed memory parallelism. Together with proofs of the convergence property of our preconditioned algorithm, we provide experimental results carried out on a cluster of multi-threaded nodes, and we validate them by means of comparisons with the multi-threaded linear solver for cluster interface implemented in the Intel MKL library. The results presented in this chapter are published in [8] and [9].

7.1 Introduction and preliminary notions

Implementing fast linear solvers is an evergreen task in numerical Linear Algebra, as it appears as an intermediate step in a number of wider problems. Linear systems $\mathbf{Ax} = \boldsymbol{b}$, where $\mathbf{A}$ is the coefficient matrix, may have some structural and mathematical properties that can be conveniently exploited. For this reason, many algorithms have been devised in order to take advantage of the characteristics of $\mathbf{A}$ for accelerating the solution of the linear system and to make them applicable on HPC architectures. A very characterizing feature of matrices is their number of zero entries. Depending on this, a matrix may be classified either as *dense* or as *sparse*.

Definition 7.1.1. *A matrix* $\mathbf{A} \in \mathbb{K}^{n \times n}$ *is* sparse *if its number of non-zero entries is* $O(n)$.

Intuitively, a matrix is sparse if the majority of its entries is zero. The non-zero entries of a sparse matrix are its *sparsity pattern*. In order to spare memory, sparse matrices are usually represented using sparse formats instead of classic full 1/2-D arrays. Depending on the sparsity pattern of the matrix, it is crucial to use the best fitting format in order to use memory devices efficiently. We hereby list some well known sparse formats that we will be using in this chapter. In the following, nz is the number of non zero entries of a sparse matrix, $\mathbf{A}$.

- **Coordinate (COO).** The coordinate format stores the non zero entries of $\mathbf{A}$ in a $nz \times 3$ array, $\mathbf{C}$ such that $c_{k,0} = a_{i,j}$, $c_{k,1} = i$ and $c_{k,2} = j$.
- **Compressed Sparse Row (CSR).** This format uses three arrays: M, I and E. In M the non zero elements of $\mathbf{A}$ are stored, row-wise. I represents the indexes of the first element of each row, i.e., $I(k) = \hat{k}$, where $M(\hat{k})$ is the first non zero entry of $\mathbf{A}$ in row k. Finally, E is the column indexes array, meaning that $E(k)$ is the column index in $\mathbf{A}$ of $M(k)$.
- **Compressed Sparse Column (CSC).** Analogous to CSR format, except that the role of rows and columns is swapped.
- **Ellpack (ELL).** The Ellpack format stores $\mathbf{A}$ with two 2D arrays, M and I. The i-th row of M is filled in with all the non zero elements of the corresponding rows of $\mathbf{A}$. The original column index of all such entries are stored in the i-th row of I.

There is not an universal format that is convenient for all possible sparsity patterns, and researchers decide which format to use depending on specific data. For instance, COO is advantageous when $\mathbf{A}$ is very sparse and its sparsity pattern does not present any particular structure. CSR and CSC are also good for generic sparse matrices and they do not store any unnecessary element, and they are preferred for accessing the rows and columns efficiently, respectively; yet they lack in efficiency as acceding to data requires a double addressing. The Ellpack format instead may waste data storage when the number of non zero elements per row of $\mathbf{A}$ varies significantly. On the other end, it is more efficient when the number of non zero elements of $\mathbf{A}$ is balanced among its rows. More sparse formats are described in [122].

Sparse matrices arise from problems in several fields, including circuit simulations, power network analysis and graph theory, as well as from the discretization of partial differential equations (PDEs) when modelling phenomena spanning the widest scientific range, including meteorology, fluid dynamics and complex biological processes. In engineering, Finite Element Model (FEM) (see e.g., [50, 88, 123]) is widely used to model structural components of vehicles, big infrastructures, prostheses, etc., and reproduces these objects through matrices depicting their physical and mechanical structures. Such matrices are usually very large and sparse, and denser blocks are distributed along their diagonal, without any other specific assumption on their sparsity pattern; we refer to them as *Quasi-Block Diagonal (QBD) matrices*. They represent the coefficient matrices of the PDEs that model the systems under study, and that are discretized and solved numerically integrating multiple large, sparse linear systems. In this chapter, we present a hybrid implementation of preconditioned Jacobi to solve QBD linear systems. The Jacobi algorithm belongs to the class of *iterative* linear solvers. In contrast to *direct* solvers, where an exact solution is computed, iterative solvers produce a sequence of solutions that converge to the exact one. The most commonly used direct solvers are the Gaussian Elimination (GE) and the Cholesky factorization. They are mainly used when the coefficient matrix $\mathbf{A}$ of the linear system to solve is dense. This is because such factorizations usually affect the sparsity pattern of $\mathbf{A}$ causing a phenomenon

known as *fill-in*, that is when some operation transforms a sparse matrix into a full one. This is an effect that must be avoided in order to take advantage of the large number of zero entries of a matrix, [62].
The advantages of iterative algorithms for solving linear systems are that they avoid fill-in, and that they can be interrupted when the desired solution accuracy is reached. The work presented in [123] represents an extensive study on iterative algorithms for numerical Linear Algebra. Some iterative methods are able to reach solution convergence only when the coefficient matrix of the linear system to be solved has some specific mathematical properties. This is also the case of the Jacobi algorithm, where it is required that the coefficient matrix is *strictly diagonally dominant*:

Definition 7.1.2. *A matrix* $\mathbf{A} = (a_{i,j}) \in \mathbb{R}^{n \times n}$ *is* strictly diagonally dominant *if for all* $i \in \{1, \ldots, n\}$ *it holds that*

$$|a_{i,i}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{i,j}|,$$

i.e., the absolute value of the diagonal entry of each row is greater than the sum of the absolute values of the off-diagonal entries of the row.

The Jacobi algorithm works as follows: let $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ a linear system where the coefficient matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is strictly diagonally dominant. Let $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$ be an initial guess for the solution of the system (e.g., $\boldsymbol{x}^{(0)} = \mathbf{0}$). Repeat until convergence:

$$x_i^{(k)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{i,j} x_j^{(k-1)} \right) \tag{7.1}$$

for all $i = 1, \ldots, n$. Each $\boldsymbol{x}^{(k)} = (x_1^{(k)}, \ldots, x_n^{(k)})$ is an approximated solution to the linear system $\mathbf{A}\mathbf{x} = \boldsymbol{b}$ and is a refinement of the previous approximated solution. If $\epsilon > 0$ is the desired solution accuracy, the iterative procedure is interrupted when $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\| < \epsilon$. Observe that the Jacobi algorithm does not alter the sparsity pattern of the original matrix $\mathbf{A}$; furthermore, it is a very good candidate for parallelization due to being inherently divisible into parallel tasks, [98]
In the work described in this chapter, we implement a *preconditioned* version of the Jacobi algorithm. Preconditioning a matrix, or a linear system, means to apply some operation on the linear system that ease its solution by guaranteeing efficiency properties. Our preconditioning phase is inspired by the factorization computed in the Spike algorithm, that is instead applied on banded matrices (see Section 7.2). This preliminary operation is particularly convenient for QBD matrices, that can then be stored very efficiently, reducing memory requirements. The solver here presented is implemented using a parallel hybrid approach that combines together distributed and shared memory paradigms and is run on a cluster.

The distributed-memory model is managed by the MPI (Message Passing Interface) library that allows distributed and independent parallel processes to cooperate and to synchronize through communication, [137]. Each distributed process runs

on a multi-core system where parallel operations are supported by the OpenMP API [34].

The multi-processor system is a particularly suitable environment for our application, since most operations can be computed in perfect parallelism and communication is limited to a small amount of data. Comparisons with other parallel solvers, provided by high-performance libraries (Intel MKL and PARDISO), are reported to validate our software.

7.2 Related Work

Linear solvers for sparse matrices have been under investigation for long years. Iterative methods are particularly convenient for sparse linear systems, since they do not alter their sparsity patterns: Jacobi and Gauss-Seidel algorithms, together with Krylov subspaces-based methods (e.g., CG, MINRES, GMRES, etc. see e.g., [123]) have been used, modified and adapted to specific problems in numerous occasions, see e.g., [21, 124, 128]. Direct solvers may be adapted in order to make them more suitable for sparse matrices, as for instance ILU and IChol factorizations (see e.g., [123]). A direct-method approach is used also in Spike algorithm, that inspired the preconditioning phase of the solver here proposed. Spike was introduced in [113] and [112] and it is a hybrid solver for sparse and dense banded linear system. Although several promising preconditioners for sparse, non symmetric and indefinite linear systems have been developed, see e.g., [54], the one introduced in Spike algorithm properly fits our problem.

In [100] a multi-threaded version of Spike was implemented for shared-memory architectures using OpenMP and offering a competitive alternative to BLAS LU-based solver. A different multi-threaded implementation is described in [23] and compared to PARDISO routines for direct solvers. In [97] the Spike family of solvers is combined with the general sparse solver PARDISO to produce a fast, robust hybrid software, PSpike, and compared with some direct and iterative solver packages. A hybrid approach is instead introduced in [96]. An extensive analysis of different approaches to perform the factorization phase of Spike is given in [48]. Our solver uses the matrix splitting shown in [23] for preconditioning, and then solves the preconditioned linear system using the Jacobi algorithm, that is suitably implemented in hybrid multi-processors/threading architectures, see also [127]. Another class of sparse matrices are almost block diagonal matrices, that were described in [4] and arise in discretizations of boundary value problems for ordinary differential equations (BVODEs) and of related partial differential equations (PDEs). They can be interpreted as block banded matrices, and therefore they have a less general sparsity pattern than QBD matrices, defined in Section 7.3.

The fastest linear solvers are implemented in routines provided by high-performance Linear Algebra libraries. Some of them also supply implementations for parallel environments. In particular we compare our hybrid solver with Intel MKL PARDISO routines for cluster interfaces (see [55]).

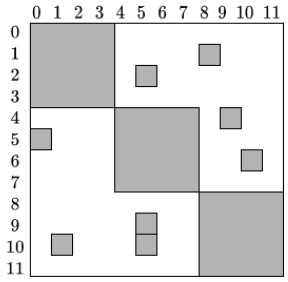
7.3 Preconditioned Jacobi for Quasi-Block Diagonal Linear Systems

Before describing the idea of our parallel implementation, we give the definition of Quasi-Block Diagonal (QBD) matrices and we introduce the notion of *D-sparse matrices*.

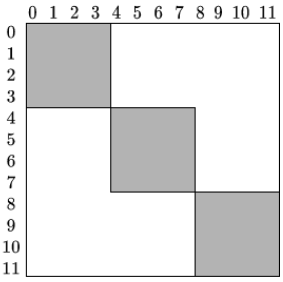
Definition 7.3.1. ***(D-sparse matrix)*** *Let* $\mathbf{D} \in \mathbb{R}^{n \times n} = diag(\mathbf{D}_0, \ldots, \mathbf{D}_{k-1})$ *be a block diagonal matrix. We say that a matrix* $\mathbf{R} \in \mathbb{R}^{n \times n}$ *is* $\mathbf{D}$-sparse *if it is sparse and if its sparsity pattern does not overlap the one of* $\mathbf{D}$*, that is* $r_{i,j} \neq 0$ *implies* $d_{i,j} = 0 \ \forall i, j \in \{1, \ldots, n\}$.

Definition 7.3.2. ***(Quasi-Block Diagonal matrix)*** *Let* $\mathbf{R}$ *and* $\mathbf{D} \in \mathbb{R}^{n \times n}$ *be a* $\mathbf{D}$*-sparse and a block diagonal matrix, respectively. A* quasi-block diagonal *matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *is a matrix defined as* $\mathbf{A} = \mathbf{D} + \mathbf{R}$.

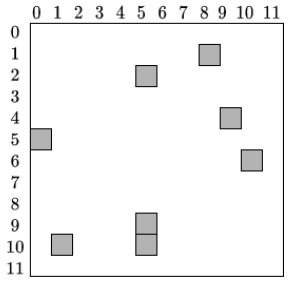
A simple example of how quasi-block diagonal matrices are split in a block diagonal matrix and a $\mathbf{D}$-sparse matrix, is given in Figures 7.3.1a, 7.3.1b and 7.3.1c.



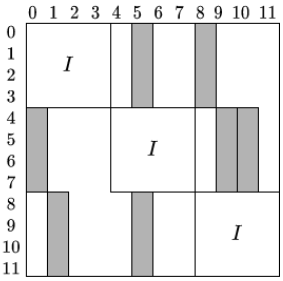
(a) $\mathbf{A} = \mathbf{D} + \mathbf{R}$.



(b) $\mathbf{D}$.



(c) $\mathbf{R}$.



(d) $\mathbf{S} = \mathbf{D}^{-1}\mathbf{A} = I + \mathbf{D}^{-1}\mathbf{R} = I + \mathbf{G}$.

Figure 7.3.1. Sparsity patterns and splittings.

The parallel implementation that we propose uses preconditioned Jacobi to solve linear systems $\mathbf{Ax} = \boldsymbol{b}$, where matrix $\mathbf{A}$ is quasi-block diagonal. As described in [23], we precondition the linear system $\mathbf{Ax} = \boldsymbol{b}$, where $\mathbf{A} = \mathbf{D} + \mathbf{R}$, by left-multiplying both sides of the system by $\mathbf{D}^{-1}$. The resulting system is $(\mathbf{I} + \mathbf{D}^{-1}\mathbf{R})\boldsymbol{x} = \mathbf{D}^{-1}\boldsymbol{b}$, that we can write in compact form as $\mathbf{Sx} = \boldsymbol{f}$, denoting $\mathbf{S} = \mathbf{D}^{-1}\mathbf{A} = \mathbf{I} + \mathbf{D}^{-1}\mathbf{R}$, and $\boldsymbol{f} = \mathbf{D}^{-1}\boldsymbol{b}$. The sparsity pattern of $\mathbf{S}$ for the example in Figures 7.3.1a-7.3.1c is shown in Figure 7.3.1d. We denote as G the matrix $\mathbf{D}^{-1}\mathbf{R} = \mathbf{S} - \mathbf{I}$, and we

observe that matrix $\mathbf{G}$ is $\mathbf{D}$-sparse and its nonzero entries are distributed in small sub-columns having the same column indexes as the nonzero entries of $\mathbf{R}$. Usually $\mathbf{S}$ is sparser than $\mathbf{A}$, in particular when $\mathbf{R}$ is very sparse and its elements have common column indexes. The Jacobi algorithm is implemented to solve the transformed system $\mathbf{Sx} = \boldsymbol{f}$.

7.3.1 Algorithm convergence

In the algorithm we propose, we conveniently apply the Jacobi method to the preconditioned system $\mathbf{Sx} = \boldsymbol{f}$. To guarantee the convergence of the method, matrix $\mathbf{S}$ has to be strictly diagonally dominant, as we prove in Proposition 7.3.1. A more detailed proof of this result is in [36]. In the following, given a matrix $\mathbf{M}$, we denote with $m_{i,j}$ the element of row i and column j.

Proposition 7.3.1. *If* $\mathbf{A} = \mathbf{D} + \mathbf{R}$ *is strictly diagonally dominant, also* $\mathbf{S} = \mathbf{D}^{-1}\mathbf{A} = \mathbf{D}^{-1}(\mathbf{D} + \mathbf{R})$ *is strictly diagonally dominant.*

Proof. First, observe that all diagonal elements of matrix S are equal to 1: $s_{i,i} = 1$, $\forall i = 1, \ldots, n$. Hence, we have to show that $||\mathbf{D}^{-1}\mathbf{R}||_\infty < 1$. In order to do so, denoting with $d_{i,j}$ and $\tilde{d}_{i,j}$ the elements of $\mathbf{D}$ and $\mathbf{D}^{-1}$ respectively, we can write:

$$\sum_{k=1}^{n} \left[|\tilde{d}_{i,k}| \cdot \left(|d_{k,k}| - \sum_{\substack{j=1 \\ j \neq k}}^{n} |d_{k,j}| \right) \right] \leq 1 \qquad \forall i = 1, \ldots, n. \tag{7.2}$$

Considering matrix $\mathbf{G}(= \mathbf{S} - I)$, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, it holds that:

$$|s_{i,j}| = |g_{i,j}| = \left| \sum_{k=1}^{n} \tilde{d}_{i,k} r_{k,j} \right|.$$

Summing on j, we can write the following inequality:

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} |s_{i,j}| \leq \sum_{k=1}^{n} \left[|\tilde{d}_{i,j}| \cdot \sum_{\substack{j=1 \\ j \neq i}}^{n} |r_{i,j}| \right]. \tag{7.3}$$

Since $\mathbf{A}$ is by hypothesis strictly diagonally dominant, so is $\mathbf{D}$, and therefore

$$\sum_{j=1}^{n} |r_{k,j}| < |d_{k,k}| - \sum_{\substack{j=1 \\ j \neq i}}^{n} |d_{k,j}|. \tag{7.4}$$

Combining inequalities (7.2), (7.3) and (7.4), the thesis follows. □

7.4 Hybrid Preconditioned Jacobi Implementation

Our solver implements preconditioned Jacobi in a hybrid distributed and shared memory fashion. We shall refer to it as to hybrid preconditioned Jacobi (HPJ). The distributed memory parallelism is managed by the MPI library that allows distributed and independent parallel processes to cooperate and to synchronize through communication. The execution of each process is accelerated by deploying

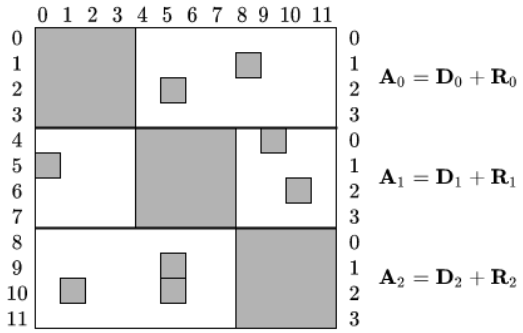


Figure 7.4.1. Matrix $\mathbf{A}$ partitioned horizontally into $\mathbf{A}_0$, $\mathbf{A}_1$ and $\mathbf{A}_2$, where $\mathbf{A}_i = \mathbf{D}_i + \mathbf{R}_i$, for $i = 0, 1, 2$.

multi-threading, enabled by the OpenMP API.
Let $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ be the quasi-block diagonal linear system to solve. HPJ works when the input problem is assigned to distributed processes as follows: the coefficient matrix $\mathbf{A}$ is partitioned horizontally in P rectangular sub-matrices, where P is the number of distributed processes. We shall call $\mathbf{A}_p$ (and, consequently, $\mathbf{D}_p$ and $\mathbf{R}_p$) the p-th sub-matrix of $\mathbf{A}$, $p \in \{0, 1, \ldots, P-1\}$. Similarly, let us call $\boldsymbol{b}_p$ the portion of the vector of known terms $\boldsymbol{b}$ having the same row indexes as $\mathbf{A}_p$, and let us define $\boldsymbol{f}_p$ analogously. The cuts are made along the diagonal blocks of $\mathbf{A}$, and hence of $\mathbf{D}$. In the simple example of Figure 7.4.1 and in the discussion that follows in this section, the number of partitions is the same as the number of blocks, hence each partition has exactly one block. More general configurations will be taken into account in Section 7.5, where we shall discuss about experimental results. Three phases may be distinguished in HPJ: *I. Preconditioning*, *II. Communication* and *III. Solving* phases. We shall now describe them in detail.

I. Preconditioning phase As explained in Section 7.3, the first step to execute is the preconditioning $\mathbf{D}^{-1}\mathbf{A}\boldsymbol{x} = \mathbf{D}^{-1}\boldsymbol{b}$. The particular shape of the precondition matrix allows to compute $\mathbf{D}^{-1}$ explicitly in a convenient way. We use the fact that the inverse of a block diagonal matrix $\mathbf{D} = diag(\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_{P-1})$ is again a block diagonal matrix, composed of the inverse of each block, that is $\mathbf{D}^{-1} = diag(\mathbf{D}_0^{-1}, \mathbf{D}_1^{-1}, \ldots, \mathbf{D}_{P-1}^{-1})$. This property allows to perform matrix inversion explicitly and in perfect parallelism without the need for distributed processes to communicate. Because of this, we choose to compute $\mathbf{D}^{-1}$ explicitly instead of integrating the preconditioning process inside the Jacobi algorithm. This choice is corroborated by the fact that also the matrix multiplication $\mathbf{G} = \mathbf{D}^{-1}\mathbf{R}$ can be carried out in perfect parallelism: thanks to the sparsity patterns of $\mathbf{D}$ and $\mathbf{R}$, it holds that $\mathbf{G} = [\mathbf{G}_0; \mathbf{G}_1; \ldots; \mathbf{G}_{P-1}]$ where $\mathbf{G}_p = \mathbf{D}_p^{-1}\mathbf{R}_p$. The last operation of this phase is the matrix-vector product, $\boldsymbol{f}_p = \mathbf{D}_p^{-1}\boldsymbol{b}_p$. Integrating the preconditioning process inside the solver is usually more advisable than inverting the precondition matrix explicitly, as this is usually a more complex operation. Indeed, a well chosen precondition matrix commonly induces easily-solvable linear systems that are solved multiple times within the the overall framework. In our case though, the whole

preconditioning phase (including matrix inversion) can be carried out by distributed processes in a perfectly parallel and independent way, without the need for communication. This is rarely the case in general scenarios. Besides, once the preconditioning phase is over, all processes can safely de-allocate $\mathbf{D}$, $\mathbf{D}^{-1}$ and $\mathbf{R}$.

Figure 7.4.2 depicts this phase for the simple example of Figure 7.4.1.

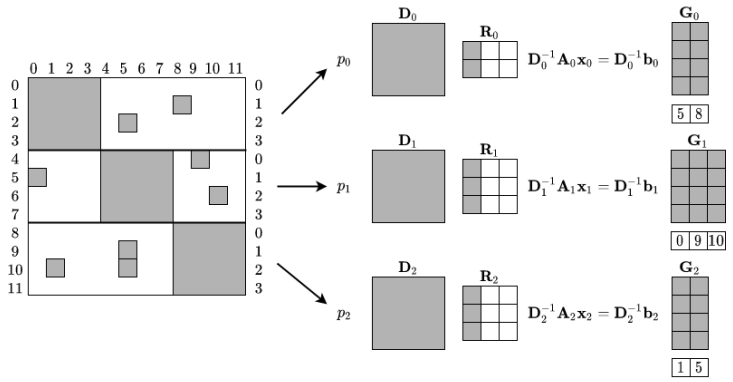


Figure 7.4.2. After partitioning $\mathbf{A}$, each process p is in charge of computing $\mathbf{G}_p$ and $\boldsymbol{f}_p$ independently. The sparse format used to store $\mathbf{G}$ is a simplified Ellpack format, see Section 7.4.1.

II. Communication phase After the preconditioning phase, distributed processes are ready to solve the preconditioned linear system. Each process p computes $\boldsymbol{x}_p$, that is the portion of the approximated solution $\boldsymbol{x}$ having the same rows as $\mathbf{G}_p$.

Since $\mathbf{R}$ and $\mathbf{G}$ are $\mathbf{D}$-sparse, the column indexes of $\mathbf{G}_p$ do not intersect the row indexes of $\boldsymbol{x}_p$, meaning that the approximated solution $\boldsymbol{x}_p$ depends on components of $\boldsymbol{x}$ that are not in $\boldsymbol{x}_p$ and that are going to be computed by other processes. For this reason, before applying the Jacobi algorithm, communication among process must be established: depending on the nonzero column indexes of $\mathbf{G}_p$, each process p sends an integer to all other processes as follows: if the indexes of some nonzero columns of $\mathbf{G}_p$ are in the same range as the rows of block $\mathbf{D}_q$, process p sends to process q an integer $ncols_p^q$ saying how many columns of $\mathbf{G}_p$ lay on the q-th block, and an array of length $ncols_p^q$ containing the indexes of such columns. In this way, processor q is aware of what entries of $\boldsymbol{x}_q$ it must send to processor p at each Jacobi iteration. This preliminary communication phase is depicted in Figure 7.4.3 for the example of Figure 7.4.1.

If, contrarily, none of the column indexes of $\mathbf{G}_p$ is in the same range as the row indexes of block $\mathbf{D}_r$, process p sends -1 to process r; this means that none of the entries of $\boldsymbol{x}_r$ appears in the equations that process p has to solve, and process r will not have to send anything to process p during the Jacobi iterations.

III. Solving phase When solving a portion of the approximated solution by implementing the Jacobi algorithm, each process p computes $\boldsymbol{x}_p^{(k+1)} = \boldsymbol{f}_p - \mathbf{G}_p\boldsymbol{x}|_{\mathbf{G}_p}^{(k)}$, being $\boldsymbol{x}|_{\mathbf{G}_p}$ the vector of the components of the approximated solution $\boldsymbol{x}$ having the same indexes as those of the columns of $\mathbf{G}_p$.

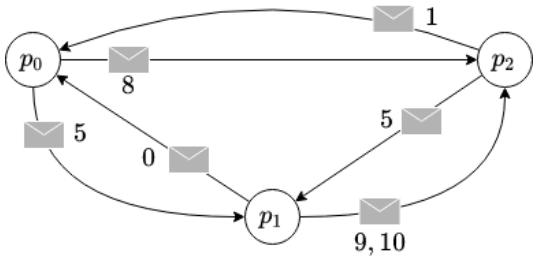


Figure 7.4.3. Processes tell each other what are the indexes of the component of the approximated solution that they are going to be needing at each iteration of the Jacobi algorithm.

During each iteration, processes properly exchange the updated values of the approximated solution and the partial errors, $\epsilon_p^{(k+1)} = ||\boldsymbol{x}_p^{(k+1)} - \boldsymbol{x}_p^{(k)}||$. In this way all distributed processes can compute the global error $\epsilon^{(k+1)} = \sqrt{\sum_{p=0}^{P-1} \epsilon_p^{(k+1)^2}}$ so that they meet the stopping criteria at the same iteration: the algorithm stops either when $\epsilon^{(k+1)}$ is below a threshold t or when the maximum number of iterations is reached. For all our testes, we set $t = 10^{-15}$ and the maximum number of iterations to be 1000. A scheme representing the solving phase for our example is depicted in Figure 7.4.4.

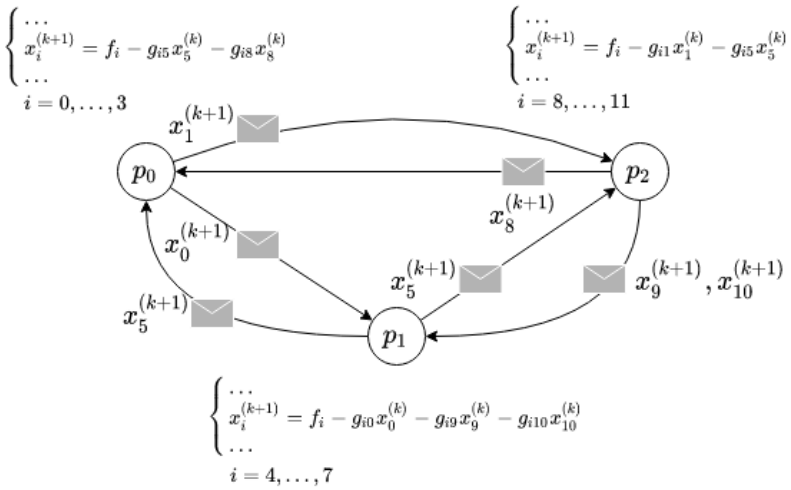


Figure 7.4.4. Each process p computes its portion of solution, $\boldsymbol{x}_p$. Then processes exchange the components of $\boldsymbol{x}_p$ depending on the information established during the communication phase.

7.4.1 Technical details

In our solver we apply the Intel MKL LAPACK and CBLAS OpenMP-threaded routines `dgesv` and `dgemv` in order to compute $\mathbf{D}_p^{-1}$ and $\boldsymbol{f}_p = \mathbf{D}_p^{-1}\boldsymbol{b}_p$, respectively. This can be done because matrices $\mathbf{D}_p$ are stored as a dense matrix. Matrices $\mathbf{R}_p$ are

instead stored in coordinates format, whereas matrices $\mathbf{G}_p$ use a modified Ellpack format, where the column indexes matrix is an array that only has as many rows as the number of diagonal blocks that process p stores. Because of the different and specific sparse formats that we adopted, we implemented the matrix multiplication $\mathbf{G}_p = \mathbf{D}_p^{-1}\mathbf{R}_p$ with the OpenMP directives (`#pragma omp`).

7.5 Performance Evaluation

We have tested our solver on randomly generated strictly diagonally dominant, quasi-block diagonal matrices. We compared the execution time required to compute a solution having order of 10^{-16} precision with the sequential version of the algorithm, and with Intel MKL PARDISO routines for cluster interfaces.

7.5.1 Experimental setup

Performance have been tested on 'Galileo' [140], installed in CINECA (Bologna, Italy - IBM NeXtScale, Linux Infiniband Cluster consisting of 360 nodes 2 x 18-cores Intel Xeon E5-2697 v4 (Broadwell) processors (2.30 GHz), and 15 nodes 2 x 8-cores Intel Haswell (2.40 Ghz) processors (endowed with 2 NVIDIA K80 GPUs, that we do not use in our implementation). In our tests, we generate $P = k^2$ distributed processes for different values of k. We involve k cluster compute nodes, and k tasks per node, each using 4 cores for multi-threading.

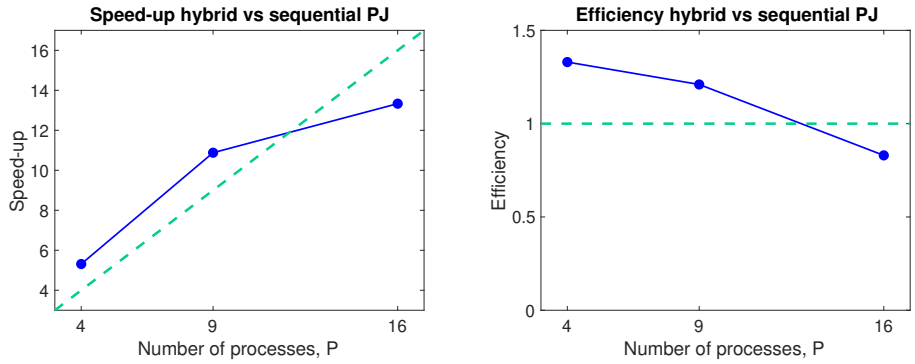
7.5.2 HPJ vs Sequential

We assess the performance of HPJ in terms of speed-up and efficiency, with respect to the sequential version of HPJ, that is, HPJ run with only one MPI process, and with the LAPACK `dgesv` function for solving general linear systems provided by Intel MKL. These two benchmarks are run using 4 cores. We compute the speed-up as the ratio T_s/T_p, where T_s and T_p are the execution times required by the sequential and the parallel implementations, respectively. Efficiency is computed as S/P, where S is the speed-up and P is the number of distributed processes, since we are running sequential preconditioned Jacobi and `dgesv` with 4 cores. For both batches of experiments, the average execution time is computed on randomly generated quasi-block diagonal linear systems of size $n = 10^4$. The number of diagonal blocks is randomly chosen in the range $\{25, \ldots, 100\}$; the size of each block ranges in the set $\{100, 200, 300, 400\}$. We generate random row and column coordinates for the non zero entries of matrix $\mathbf{R}$.

The outcomes of the first set of experiments, comparing HPJ and the sequential preconditioned Jacobi, are shown in Figures 7.5.1 (a) and (b). Speed-up is super-linear when the number of MPI processes P is 4 and 9, and goes slightly below the linear trend for $P = 16$, where the efficiency is 0.83.

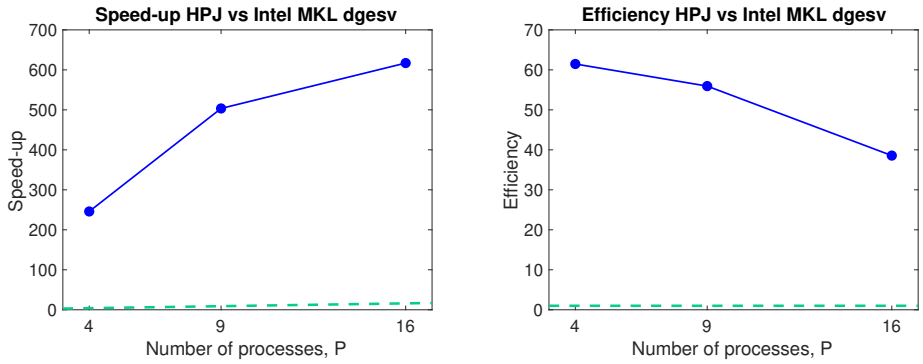
In Figures 7.5.2 (a) and (b), we show the speed-up and the efficiency of HPJ with respect to the Intel MKL implementation of the LAPACK `dgesv` routine.

The phenomenal performance of HPJ in this case are due to the fact that LAPACK does not provide sparse formats for storing sparse matrices; as a consequence, its performance is poor on quasi-block diagonal linear systems. In both experiments,



(a) (b)

Figure 7.5.1. Speed-up (a) and efficiency (b) of hybrid vs sequential preconditioned Jacobi.



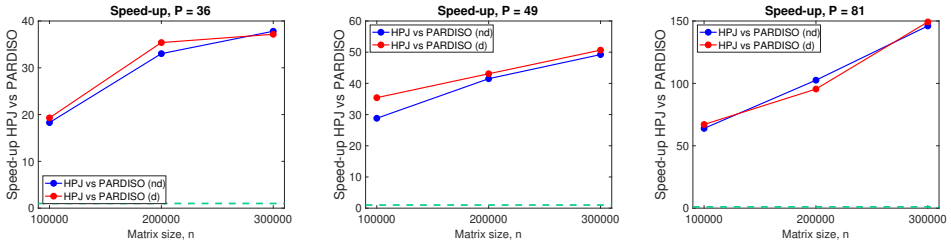
(a) (b)

Figure 7.5.2. Speed-up (a) and efficiency (b) of HPJ versus Intel MKL `dgesv`.

super-linearity is also a consequence of the greater memory availability of our hybrid solver, since it is run on more than one compute node, as explained in Section 7.5.1.

7.5.3 HPJ vs Intel MKL PARDISO

We have compared our solver HPJ with Intel MKL PARDISO, that is a software package of the Intel MKL library for solving large sparse linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. The Intel MKL library provides routines for cluster interfaces that integrate PARDISO functions and that follow a hybrid MPI/OpenMP approach. In particular, we used the `cluster_sparse_solver` routine for symbolic factorization, numerical factorization, solution retrieval and termination. PARDISO allows to store data either with CSC or CSR sparse formats. We opted for the second, as it is favourable when row-major is the default layout for array storage, that it the case of our implementation. The `cluster_sparse_solver` routine supports both distributed (d) and non distributed (nd) inputs. In the first case, matrix $\mathbf{A}$ is partitioned among distributed processes. In the non distributed fashion instead, the MPI master process (the one having rank = 0) is the only one to read the whole matrix and to manage workload distribution among processes. Both configurations were compared with HPJ. We used the MPI timer to track the execution time for computation steps. In



(a) (b) (c)

Figure 7.5.3. Speed-up of HPJ vs distributed (d) and non-distributed (nd) Intel MKL PARDISO for $P = 36$ (a), $P = 49$ (b) and $P = 81$ (c) MPI processes on matrix sizes 10^5, $2 \cdot 10^5$ and $3 \cdot 10^5$.

`cluster_sparse_solver`, this corresponds to the time for numerical factorization, backward substitution and iterative refinement. In our experiments, we set two steps of iterative refinement. Initially, the nested dissection algorithm from the METIS package is used for reordering the input matrix in order to reduce the amount of fill-in during the numerical factorization phase.[1] Figures 7.5.3(a-c) show the speed-up of HPJ with respect to Intel MKL PARDISO in both the distributed and non-distributed fashion. We show how the speed-up changes varying the matrix size, n, for different numbers of MPI processes, $(P = 36, 49, 81)$. In this set of experiments, when quasi-block diagonal random matrices of size n are generated, the number of diagonal blocks ranges in the set $\{n/1000, \ldots, n/500\}$, and the size of each block in the set $\{500, 600, 700, 800, 900, 1000\}$. We generate random row and column coordinates for the non zero entries of matrix $\mathbf{R}$. For every interval of row indexes defined by the row indexes of each diagonal block, we generate a random number of non zero entries of $\mathbf{R}$ in the set $\{25, 40, 55, 70, 85, 100\}$.

The average execution times of HPJ and Intel MKL PARDISO are barely comparable, and the latter suffers from poor scalability. This fact holds for both distributed and non distributed configurations and is supported by the technical description of Intel PARDISO for clusters package, given in [78], where it is shown how PARDISO scaling factor is low when the number of threads dedicated to each MPI process is small. In that work, the authors tested the Intel PARDISO for clusters solver on two smaller matrices, with $15.7 \cdot 10^6$ and $12 \cdot 10^6$ non zero entries, and on a larger one, having $5 \cdot 10^8$ non zero entries, and they analyse the behaviour of the solver for different numbers of threads per MPI process. Only one MPI process per compute node is generated. A configuration with 4 threads per process (that is the same one adopted in our experiments) is considered too, and results in a low speed-up (approximately, the value of the speed-up is 9 and 12 on 16 MPI processes on the two smaller matrices, and less than 4 on 16 MPI processes for the larger linear system). The number of non zero elements of our largest matrices ranges between $9 \cdot 10^7$ and $1.5 \cdot 10^8$, hence our experiments represent an intermediate situation between the two scenarios considered in [78]). Performance improves when all the available threads in each compute nodes are activated, suggesting that the

[1] More details on Intel MKL `cluster_sparse_solver` available options can be found on page: https://software.intel.com/en-us/mkl-developer-reference-c-cluster-sparse-solver-iparm-parameter

best configuration for Intel MKL PARDISO to set up is to reserve an entire compute node per MPI process. HPJ, instead, does not suffer from this drawback, as it does not require such a large hardware availability to achieve high performance.

7.6 Conclusions

In this chapter we proposed a hybrid MPI/OpenMP parallel solver for clusters of multi-threaded processors, that uses preconditioned Jacobi to solve sparse linear systems and is particularly suitable for quasi-block diagonal matrices, that commonly emerge from finite element analysis. The preconditioned linear system has provable convergence properties. In our implementation, the preconditioning phase is executed in perfect parallelism among processes, and independent computations are accelerated by using OpenMP for multi-threading. Furthermore, the solving phase implies a negligible communication cost. To reduce the memory storage consumption, we used specific sparse matrix formats that allow to exploit the specific pattern of non-zero elements of quasi-block diagonal matrices. Our hybrid implementation was tested against the analogous sequential version and the `dgesv` routine implemented in the Intel MKL library, revealing impressive performance and exceptional scalability. Comparisons with solvers provided by the high performance library Intel MKL PARDISO show that our solver is the fastest in all considered experimental configurations.

Conclusions

This thesis tackles two major topics: node identifiability and failure identification in communication networks in the context of Boolean Network Tomography (Part I), and high performance computing solutions to some standard problems of Numerical Linear Algebra (Part II).

The first part of the thesis analyses the problem of evaluating network performance in terms of node failure detection by means of end-to-end measurements. This technique allows to gain knowledge on the state of the network while avoiding the obstacles due to commercial factors, that often prevent private organizations from sharing sensitive data on their portion of the Internet network. In addition, analysing end-to-end measurements also provides a convenient solution for communication networks maintenance to restrict the possible area of malfunctions, limiting expensive direct investigations when networks are distributed in vast geographical areas and when cataclysms and accidents causing network faults make human intervention too dangerous.

Firstly, we showed fundamental background on BNT techniques. We introduced the property of *node identifiability*, that is the capability of a node to be uniquely identified as 'failed', given a set of measurement paths. In Chapters 2 and 3, we studied a primal-dual optimization problem on the maximum number of nodes that can be identified given a certain number of paths, and on the minimum number of paths that need to be placed in order to make a desired number of nodes identifiable. We took into account different constraints, including the average and the maximum path length, and the general routing scheme adopted. We proposed an algorithm for generation of a regular topology that proves the tightness of both lower and upper bounds under arbitrary routing.

After the formulation of theoretical bounds on node identifiability, the thesis tackles the problem of node failure detection under uncontrollable routing (Chapter 4). We provided a greedy algorithm based on a Bayesian approach, that has the goal of providing maximal information on the network (in terms of failed nodes) by probing the least number of paths. The performance of this algorithm is naturally and inherently upper-bounded by Boolean Network Tomography limits (i.e., probing all available paths). Our tests show how this limit is always reached by probing a very small percentage of the total set of measurement paths. The decision on the paths to test is driven by stochastic metrics, that have provable optimal approximations and that are easily computable. We also extend our results by considering possible dynamic changes in the network.

The results provided in the first part of this thesis leave room for further deepen-

ing: in the study our bounds on node identifiability, we neglect other requirements, besides those related to routing and hop count for QoS, for example cost, technology constraints and more advanced performance requirements of such networks, which could be considered in a future study, with a more technology and topology specific point of view. The node failure scenario that we consider is generic and agnostic. In some cases though, as for instance when geological or human disasters occur, prior failure probability might not be considered distributed uniformly among nodes. Also, our approach for failure detection can be adapted to take into account different specifications, as peer-to-peer communication, client-server architectures, three-tier networks.

The second part of this thesis describes algorithms and parallel implementations for two major issues in Numerical Linear Algebra. In fact, numerical algorithms have a vast field of application in applied Science and Engineering, and often represent a bottle-neck in complex systems simulations.

In Chapter 6, we introduced an algorithm for computing the matrix product $\mathbf{A}^T\mathbf{A}$, that is an intermediate operation in several problems in Linear Algebra and Differential Geometry. The algorithm is cache-oblivious and based on Strassen's algorithm for matrix multiplications. The computational cost of the algorithm reduces the cost of the naive $\mathbf{A}^T\mathbf{A}$ multiplication. We provide scalable, shared and distributed implementations of the algorithm that makes use of routines integrated in state-of-the-art libraries for Linear Algebra. Thanks to the minimization of memory allocation and communication between distributed processes, our implementation outperforms the state-of-the-art counterparts represented by the MKL library by Intel and other pieces of research solutions.

In Chapter 7 we describe a linear solver for systems of equations that come from system simulations in Engineering, whose coefficient matrix is said *quasi-block diagonal*. The linear solver uses preconditioned Jacobi for taking advantage of the shape of the systems, and uses ad-hoc sparse formats for saving memory storage requirements. We have developed a hybrid implementation that combines together both the distributed and shared memory approaches. Despite the hardness related to distributing tasks and elaborating data between independent processes in linear solvers, we showed how to minimize data exchange between distributed components by taking advantage of the sparsity pattern of quasi-block diagonal systems. As a result, our tests show optimal memory use and outstanding performance when compared with routines for sparse linear solvers integrated in state-of-the-art libraries for cluster interfaces.

Acknowledgements

I would like to express my gratitude to my advisor Prof. Annalisa Massini for her endless support, her tireless mentoring and for encouraging my personal growth. I am deeply grateful to Prof. Novella Bartolini for her extremely valuable guidance and assistance. The profound knowledge that they shared with me and their sympathetic advice have been indispensable for the success of my Ph.D. and for my entry into the world of Research.

I would like to extend my sincere thanks to Prof. Volker Mehrmann for his inspiring assistance during my visiting at the TU Berlin.

I would like to offer my special thanks to the external reviewers, Prof. Simone Silvestri, Prof. Francesco Lo Presti and Prof. Murat Manguoğlu, for their comments and meticulous suggestions that helped me improve the quality of my work.

I am thankful to my beloved family for their unconditioned love and immense support.

I am sincerely and enormously grateful for and to all my old and new people that were next to me during my Ph.D. To my loved ones, my friends and my colleagues go my most heart-felt affection and gratitude.

Bibliography

[1] D. Achlioptas, A. Clauset, D. Kempe, and C. Moore. On the bias of traceroute sampling: Or, power-law degree distributions in regular graphs. *J. ACM*, 56(4), July 2009.

[2] A. Adams, T. Bu, T. Friedman, J. Horowitz, D. Towsley, R. Cáceres, N. G. Duffield, F. Lo Presti, S. B. Moon, and V. Paxson. The use of end-to-end multicast measurements for characterizing internal network behavior. *IEEE Communications magazine*, 38(5):152–159, 2000.

[3] J. Alman and V. V. Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

[4] P. Amodio, JR Cash, G. Roussos, RW Wright, G. Fairweather, I. Gladwell, GL. Kraut, and M. Paprzycki. Almost block diagonal linear systems: sequential and parallel solution techniques, and applications. *Numerical linear algebra with applications*, 7(5):275–317, 2000.

[5] V. Arrigoni, N. Bartolini, and A. Massini. Topology agnostic bounds on minimum requirements for network failure identification. *IEEE Access*, 9:6076–6086, 2021.

[6] V. Arrigoni, N. Bartolini, A. Massini, and F. Trombetti. Failure localization through progressive network tomography. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications (INFOCOM 2021)*, 2021.

[7] V. Arrigoni, F. Maggioli, A. Massini, and E. Rodolà. Efficiently parallelizable strassen-based multiplication of a matrix by its transpose. submitted to conference, 2021.

[8] V. Arrigoni and A. Massini. Hybrid solver for quasi block diagonal linear systems. In *International Conference on Parallel Processing and Applied Mathematics*, pages 129–140. Springer, 2019.

[9] V. Arrigoni and A. Massini. Solving Quasi Block Diagonal Linear Systems. https://womencourage.acm.org/2019/wp-content/uploads/2019/07/womENcourage_2019_paper_60.pdf, 2019. Online; accessed 2 January 2021.

[10] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H.H. Liu, J. Padhye, B.T. Loo, and G. Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI) 18)*, pages 419–435, 2018.

[11] G. K. Atia and V. Saligrama. Boolean compressed sensing and noisy group testing. *IEEE Trans. on Inf. Theory*, 58(3), March 2012.

[12] Aurora Fiber Optic Networks. Last accessed November 26, 2019.

[13] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the 24^{t}h Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'12*, pages 193–204. ACM, 2012.

[14] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. In *Proceedings of the 23^{r}d Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'11*, pages 1–12. ACM, 2011.

[15] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)*, 59(6):1–23, 2013.

[16] N. Bartolini, T. He, V. Arrigoni, A. Massini, F. Trombetti, and H. Khamfroush. On fundamental bounds on failure identifiability by boolean network tomography. *IEEE/ACM Transactions on Networking*, 2020.

[17] N. Bartolini, T. He, and H. Khamfroush. Fundamental limits of failure identifiability by boolean network tomography. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[18] Y. Bejerano and R. Rastogi. Robust monitoring of link delays and faults in IP networks. In *IEEE INFOCOM*, 2003.

[19] A. R. Benson and G. Ballard. A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices*, 50(8):42–53, 2015.

[20] A.R. Benson and G. Ballard. A framework for practical parallel fast matrix multiplication. In *Proceedings 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP'15*, pages 42–53, 2015.

[21] D. P. Bertsekas and J. N. Tsitsiklis. Some aspects of parallel and distributed iterative algorithms—a survey. *Automatica*, 27(1):3–21, 1991.

[22] M. D. Bogdan. Measuring digital development: Facts and figures 2019, 2019.

[23] E. S. Bolukbasi and M. Manguoglu. A multithreaded recursive and nonrecursive parallel sparse direct solver. In *Advances in Computational Fluid-Structure Interaction and Flow Simulation*, pages 283–292. Springer, 2016.

[24] R.P. Brent. Algorithms for matrix multiplication, 1970.

[25] R.P. Brent. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Numerische Mathematik*, 16:145–156, 1970.

[26] Cooperative Association for Internet Data Analysis. Available: CAIDA.

[27] A. Charara, D. Keyes, and H. Ltaief. Batched triangular dense linear algebra kernels for very small matrix sizes on gpus. *ACM Transactions on Mathematical Software (TOMS)*, 45(2):1–28, 2019.

[28] A. Charara, H. Ltaief, and D. Keyes. Redesigning triangular dense matrix computations on gpus. In *European Conference on Parallel Processing*, pages 477–489. Springer, 2016.

[29] J. Chen, X. Qi, and Y. Wang. An efficient solution to locate sparsely congested links by network tomography. In *2014 IEEE International Conference on Communications (ICC)*, pages 1278–1283. IEEE, 2014.

[30] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 55–66, 2004.

[31] M. Cheraghchi, A. Karbasi, S. Mohajer, and V. Saligrama. Graph-contrained group testing. In *IEEE Trans. on Inf. Theory*, number 1, 2012.

[32] M. Coates, R. Castro, R. Nowak, M. Gadhiok, R. King, and Y. Tsang. Maximum likelihood network topology identification from edge-based unicast measurements. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):11–20, 2002.

[33] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

[34] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[35] P. D'Alberto and A. Nicolau. Adaptive strassen's matrix multiplication. In *Proceedings 21st International Conference on Supercomputing*, pages 284–292. ACM, 2007.

[36] N. D'Alessandro. Comparison of direct and iterative methods applied to almost block diagonal matrices, master thesis, computer science department, sapienza university of rome, italy. 2019.

[37] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 261–272. IEEE, 2013.

[38] F. Desprez and F. Suter. Impact of mixed-parallelism on parallel implementations of the Strassen and Winograd matrix multiplication algorithms. *Concurr. Comput.: Pract. Exper.*, 16(8):771–797, 2004.

[39] R. Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 1943.

[40] B. Du, M. Candela, B. Huffaker, A. C. Snoeren, and K. C. Claffy. Ripe ipmap active geolocation: mechanism and performance evaluation. *ACM SIGCOMM Computer Communication Review*, 50(2):3–10, 2020.

[41] N. G. Duffield. Network tomography of binary network performance characteristics. *IEEE Trans. on Inf. Theory*, 52, 2006.

[42] N. G. Duffield, J. Horowitz, F Lo F. Presti, and Don Towsley. Multicast topology inference from measured end-to-end loss. *IEEE Transactions on Information Theory*, 48(1):26–45, 2002.

[43] N. G. Duffield, J. Horowitz, F. Lo Presti, and D. Towsley. Network delay tomography from end-to-end unicast measurements. In *Thyrrhenian Internatinal Workshop on Digital Communications*, pages 576–595. Springer, 2001.

[44] N. G. Duffield and F. Lo Presti. Network tomography from measured end-to-end delay covariance. *IEEE/ACM Transactions On Networking*, 12(6):978–992, 2004.

[45] N. G. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Inferring link loss using striped unicast probes. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2, pages 915–923. IEEE, 2001.

[46] N.G. Duffield. Simple network performance tomography. In *ACM IMC*, 2003.

[47] J. Dumas, C. Pernet, and A. Sedoglavic. On fast multiplication of a matrix by its transpose. In *Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*, ISSAC '20, page 162–169, New York, NY, USA, 2020. Association for Computing Machinery.

[48] V. Eijkhout and R. van de Geijn. The spike factorization as domain decomposition method; equivalent and variant approaches. In *High-Performance Scientific Computing*, pages 157–169. Springer, 2012.

[49] D. Eliahu, O. Spillinger, A. Fox, and J. Demmel. Frpa: A framework for recursive parallel algorithms. Technical report, University of California at Berkeley Berkeley United States, 2015.

[50] H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Oxford University Press, USA, 2014.

[51] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM review*, 46(1):3–45, 2004.

[52] B. Eriksson, G. Dasarathy, P. Barford, and R. Nowak. Toward the practical use of network tomography for internet topology discovery. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[53] H. Esfandiar, A. Karbasi, and V. Mirrokni. Adaptivity in adaptive submodularity. *arXiv preprint arXiv:1911.03620*, 2019.

[54] M. Ferronato, C. Janna, and G. Pini. A generalized block fsai preconditioner for nonsymmetric linear systems. *Journal of Computational and Applied Mathematics*, 256:230–241, 2014.

[55] Developer Reference for Intel® Math Kernel Library C. 2019.

[56] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symp. Foundations of Computer Science (FOCS)*, pages 285–297. IEEE, 1999.

[57] K. Fujii and S. Sakaue. Beyond adaptive submodularity: Approximation guarantees of greedy policy with adaptive submodularity ratio. In *International Conference on Machine Learning*, pages 2042–2051, 2019.

[58] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings 39th International Symposium on Symbolic and Algebraic Computation*, page 296–303, 2014.

[59] D. Ghita, H. Nguyen, M. Kurant, K. Argyraki, and P. Thiran. Netscope: Practical network loss tomography. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[60] D. Golovin and A. Krause. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *Journal of Artificial Intelligence Research*, 42:427–486, 2011.

[61] G. H. Golub et al. *Milestones in matrix computation: the selected works of Gene H. Golub with commentaries*. Oxford University Press, 2007.

[62] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, USA, 1996.

[63] B. Grayson, A. Shah, and R. van de Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6:3–12, 1995.

[64] P. Gupta and P. R. Kumar. Critical power for asymptotic connectivity in wireless networks. In *Stochastic analysis, control, optimization and applications*, pages 547–566. Springer, 1999.

[65] T. He, N. Bartolini, H. Khamfroush, I. Kim, L. Ma, and T. La Porta. Service placement for detecting and localizing failures using end-to-end observations. In *IEEE ICDCS*, 2016.

[66] T. He, A. Gkelias, L. Ma, K. K. Leung, A. Swami, and D. Towsley. Robust and efficient monitor placement for network tomography in dynamic networks. *IEEE/ACM Transactions on Networking*, 25(3):1732–1745, June 2017.

[67] N.J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Softw.*, 16(4):352–368, 1990.

[68] É. Hosszu, J. Tapolcai, and G. Wiener. On a problem of rényi and katona. 2013.

[69] A. S. Householder. *Numerische Mathematik*, volume 8. Springer-Verlag, 1966.

[70] Y. Huang, N. Feamster, and R. Teixeira. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review*, 38(5):53–58, 2008.

[71] S. Hunold, T. Rauber, and G. Runger. Combining building blocks for parallel multi–level matrix multiplication. *Parallel Computing*, 34:411–426, 2008.

[72] S. Huss-Lederman, E.M. Jacobson, A. Tsao, T. Turnbull, and J.R. Johnson. Implementation of strassen's algorithm for matrix multiplication. In *Proceedings ACM/IEEE Conference on Supercomputing*, 1996.

[73] H. Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, page 326–333, New York, NY, USA, 1981. Association for Computing Machinery.

[74] A. Johnsson, C. Meirosu, and C. Flinta. Online network performance degradation localization using probabilistic inference and change detection. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.

[75] R. K. Jones, F. Alizadeh-shabdiz, E. J. Morgan, and M. G. Shean. Server for updating location beacon database, August 2008.

[76] M. Kadhum, M. H. Qasem, A. Sleit, and A. Sharieh. Efficient mapreduce matrix multiplication with optimized mapper set. In *Computer Science On-line Conference*, pages 186–196. Springer, 2017.

[77] G. Kakkavas, D. Gkatzioura, V. Karyotis, and S. Papavassiliou. A review of advanced algebraic approaches enabling network tomography for future network infrastructures. *Future Internet*, 12(2):20, 2020.

[78] A. Kalinkin and K. Arturov. Asynchronous approach to memory management in sparse multifrontal methods on multiprocessors. *Applied Mathematics*, 2013, 2013.

[79] A.R. Kamal, C.J. Bleakley, and S. Dobson. Failure detection in wireless sensor networks: A sequence-based dynamic approach. *ACM Transactions on Sensor Networks (TOSN)*, 10(2):1–29, 2014.

[80] A. Karbasi and M. Zadimoghaddam. Sequential group testing with graph constraints. In *2012 IEEE information theory workshop*, pages 292–296. Ieee, 2012.

[81] G. Katona. On separating systems of a finite set. *Journal of Combinatorial Theory*, 1(2):174–194, 1966.

[82] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[83] R. R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. Detection and localization of network black holes. *IEEE INFOCOM*, 2007.

[84] A. N. Krylov. On the numerical solution of equation by which are determined in technical problems the frequencies of small vibrations of material systems. *Izvestiia Akademij nauk SSSR*, (4):491–539, 1931.

[85] B. Kumar, C.-H. Huang, R. Johnson, and P. Sadayappan. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. In *Proceedings Seventh International Parallel Processing Symposium*, pages 582–588, 1993.

[86] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[87] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proceedings of the conference on applications, technologies, architectures, and protocols for computer communication*, pages 283–294, 2000.

[88] M. G. Larson and F. Bengzon. *The finite element method: theory, implementation, and applications*, volume 10. Springer Science & Business Media, 2013.

[89] H. Li, Y. Gao, W. Dong, and C. Chen. Taming both predictable and unpredictable link failures for network tomography. *IEEE/ACM Transactions on Networking*, 26(3):1460–1473, June 2018.

[90] M. Li, Y-L. Wu, and C-R. Chang. Available bandwidth estimation for the network paths with multiple tight links and bursty traffic. *Journal of Network and Computer Applications*, 36(1):353–367, 2013.

[91] Q. Luo and J. Drake. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *Proceedings ACM Symposium on Applied Computing, SAC'95*, pages 221–226, 1995.

[92] L. Ma, T. He, A. Swami, D. Towsley, K. K. Leung, and J. Lowe. Node Failure Localization via Network Tomography. In *ACM IMC*, 2014.

[93] L. Ma, T. He, A. Swami, D. Towsley, and K.K. Leung. On optimal monitor placement for localizing node failures via network tomography. *Elsevier Performance Evaluation*, 91:16–37, September 2015.

[94] L. Ma, Ting He, Ananthram Swami, Don Towsley, and Kin K. Leung. Network capability in localizing node failures via end-to-end path measurements. *IEEE/ACM Transactions on Networking*, June 2016.

[95] L. Ma, Z. Zhang, and M. Srivatsa. Neural network tomography. *arXiv2001.02942,cs.NI*, 2020.

[96] M. Manguoglu. A domain-decomposing parallel sparse linear system solver. *Journal of computational and applied mathematics*, 236(3):319–325, 2011.

[97] M. Manguoglu, A. H. Sameh, and O. Schenk. Pspike: A parallel hybrid sparse linear system solver. In *European Conference on Parallel Processing*, pages 797–808. Springer, 2009.

[98] A. Margaris, S. Souravlas, and M. Roumeliotis. Parallel implementations of the jacobi linear algebraic systems solve. *arXiv preprint arXiv:1403.5805*, 2014.

[99] C. Mark, A. O. Hero III, Nowak Robert, and B. Yu. Internet tomography. *IEEE Signal Processing Magazine*, 19(3):47–65, 2002.

[100] K. Mendiratta and E. Polizzi. A threaded spike algorithm for solving general banded systems. *Parallel Computing*, 37(12):733–741, 2011.

[101] R. Mok, V. Bajpai, A. Dhamdhere, and K. Claffy. Revealing the load-balancing behavior of youtube traffic on interdomain links. In *Passive and Active Measurement Conference (PAM)*, Mar 2018.

[102] T. Muhammed and R.A. Shaikh. An analysis of fault detection strategies in wireless sensor networks. *Journal of Network and Computer Applications*, 78:267–287, 2017.

[103] M. Mukamoto, T. Matsuda, S. Hara, K. Takizawa, F. Ono, and R. Miura. Adaptive boolean network tomography for link failure detection. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 646–651. IEEE, 2015.

[104] G. L. Nemhauser and L. A. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of operations research*, 3(3):177–188, 1978.

[105] J. Von Neumann and H. H. Goldstine. Numerical inverting of matrices of high order. *Bulletin of the American Mathematical Society*, 53(11):1021–1099, 1947.

[106] N.X. Nguyen and P. Thiran. The boolean solution to the congested IP link location problem: Theory and practice. In *IEEE INFOCOM*, 2007.

[107] M. Ovsjanikov, M. Ben-Chen, J. Solomon, A. Butscher, and L. Guibas. Functional maps: a flexible representation of maps between shapes. *ACM Transactions on Graphics (TOG)*, 31(4):1–11, 2012.

[108] S. Pan, P. Li, D. Zeng, S. Guo, and G. Hu. A Q -learning based framework for congested link identification. *IEEE Internet of Things Journal*, 6(6):9668–9678, 2019.

[109] E. Peise and P. Bientinesi. Algorithm 979: recursive algorithms for dense linear algebra—the relapack collection. *ACM Transactions on Mathematical Software (TOMS)*, 44(2):1–19, 2017.

[110] J. P. Pickett. *The American heritage dictionary of the English language*. Houghton Mifflin Harcourt, 2018.

[111] Intel Cilk Plus. 2009. Last accessed 07-01-2021.

[112] E. Polizzi and A. Sameh. Spike: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113–120, 2007.

[113] E. Polizzi and A. H. Sameh. A parallel hybrid banded system solver: the spike algorithm. *Parallel computing*, 32(2):177–194, 2006.

[114] D. M. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[115] F. Lo Presti, N. G. Duffield, J. Horowitz, and D. Towsley. Multicast-based inference of network-internal delay distributions. *IEEE/ACM Transactions On Networking*, 10(6):761–775, 2002.

[116] M. H. Qasem, A. A. Sarhan, R. Qaddoura, and B. A. Mahafzah. Matrix multiplication of big data using mapreduce: a review. In *2017 2nd International Conference on the Applications of Information Technology in Developing Renewable Energy Processes & Systems (IT-DREPS)*, pages 1–6. IEEE, 2017.

[117] Y. Qiao, J. Jiao, Y. Rao, and H. Ma. Adaptive path selection for link loss inference in network tomography applications. *PloS one*, 11(10):e0163706, 2016.

[118] Y. Qiao, G. Wang, X-S Qiu, and R. Gu. Network loss tomography using link independence. In *2012 IEEE Symposium on Computers and Communications (ISCC)*, pages 000569–000574. IEEE, 2012.

[119] W. Ren and W. Dong. Robust network tomography: k-identifiability and monitor assignment. In *IEEE INFOCOM*, 2016.

[120] A. Rényi. On random generating elements of a finite boolean algebra. *Acta Sci. Math. Szeged*, 22(75-81):4, 1961.

[121] A. Roy, P. Kar, S. Misra, and M.S. Obaidat. D3: Distributed approach for the detection of dumb nodes in wireless sensor networks. *International Journal of Communication Systems*, 30(1):e2913, 2017.

[122] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations-version 2. 1994.

[123] Y. Saad. *Iterative Methods for Sparse Linear Systems: Second Edition*. Society for Industrial and Applied Mathematics, 2003.

[124] Yousef Saad and Henk A Van Der Vorst. Iterative solution of linear systems in the 20th century. *Numerical Analysis: Historical Developments in the 20th Century*, pages 175–207, 2001.

[125] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[126] P. Selin, K. Hasegawa, and H. Obara. Available bandwidth measurement technique using impulsive packet probing for monitoring end-to-end service quality on the internet. In *The 17th Asia Pacific Conference on Communications*, pages 518–523. IEEE, 2011.

[127] A. Shi, W. Shen, Y. Li, L. He, and D. Zhao. Implementation and analysis of jacobi iteration based on hybrid programming. In *2010 International Conference On Computer Design and Applications*, volume 2, pages V2–311. IEEE, 2010.

[128] V. Simoncini and D. B. Szyld. Recent computational developments in krylov subspace methods for linear systems. *Numerical Linear Algebra with Applications*, 14(1):1–59, 2007.

[129] A. Singla, C-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

[130] F. Song, J. Dongarra, and S. Moore. Experiments with Strassen's algorithm: From sequential to parallel. In *Proceedings Parallel and Distributed Computing and Systems, (PDCS)*, 2006.

[131] A. J. Stothers. On the complexity of matrix multiplication. *Journal of Complexity*, 19:43–60, 2003.

[132] G. Strang. *Linear Algebra and Its Applications, Fourth Ed.* Thomson Brooks/Cole, 2006.

[133] V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[134] R.R. Swain, P.M. Khilar, and S.K. Bhoi. Heterogeneous fault diagnosis for wireless sensor networks. *Ad Hoc Networks*, 69:15–37, 2018.

[135] S. Tati, S. Silvestri, T. He, and T. La Porta. Robust network tomography in the presence of failures. In *IEEE ICDCS*, 2014.

[136] S. Tati, S. Silvestri, T. He, and T. La Porta. Robust network tomography in the presence of failures. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 481–492. IEEE, 2014.

[137] CORPORATE The MPI Forum. Mpi: A message passing interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, page 878–883, New York, NY, USA, 1993. Association for Computing Machinery.

[138] M. Thottethodi, S. Chatterjee, and A.R. Lebeck. Tuning strassen's matrix multiplication for memory efficiency. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 36–36. IEEE, 1998.

[139] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[140] UG3.3: GALILEO UserGuide. 2018. Last accessed 11 Jan 2019.

[141] Y. Vardi. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American statistical association*, 91(433):365–377, 1996.

[142] E. Wang, Q. Zhang, B. Shenand G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[143] J. H. Wilkinson. *The algebraic eigenvalue problem*, volume 87. Clarendon press Oxford, 1965.

[144] V.V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings Forty-fourth Annual ACM Symposium on Theory of Computing*, page 887–898, 2012.

[145] C-Q. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed Computing Systems*, pages 366–373. IEEE, 1988.

[146] W. Zeng, R. Guo, F. Luo, and X. Gu. Discrete heat kernel determines discrete riemannian metric. *Graphical Models*, 74(4):121–129, 2012.

[147] R. Zhang, S. Newman, M. Ortolani, and S. Silvestri. A network tomography approach for traffic monitoring in smart cities. *IEEE Transactions on Intelligent Transportation Systems*, 19(7):2268–2278, 2018.