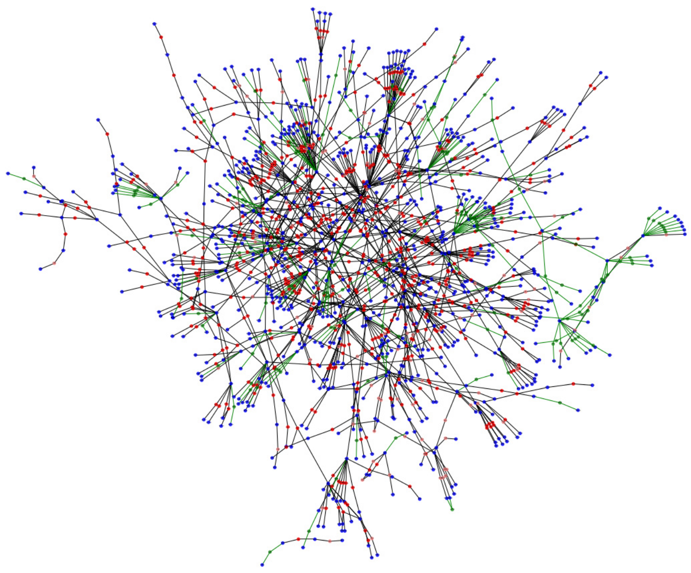
# Efficient Multi-Level Measurements and Modeling of Computer Networks



JEAN-FRANÇOIS GRAILET

Doctoral College in Electrical Engineering and Computer Science
School of Engineering and Computer Science
UNIVERSITY OF LIÈGE - BELGIUM

SUPERVISOR: PROF. BENOIT DONNET

A thesis submitted in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY (PHD) IN COMPUTER SCIENCE

SEPTEMBER 2021

Montefiore Institute

The picture on the cover of this thesis is a graphical render of a neighborhood – subnet bipartite graph generated from a snapshot of AS6453 (TATA Communications, Inc.) captured with `SAGE` [54] from the EdgeNet cluster on May 19th, 2021. The concepts of neighborhood, snapshot, bipartite graph and the topology discovery tool `SAGE` are all discussed throughout this thesis.

## Abstract

Since the 1980's, the Internet has steadily grown in size to deliver various services to an increasingly large public, which amounts to billions of users as of the beginning of the 2020's. As a consequence, its infrastructure has considerably evolved too, which prompted the research community to investigate the topology of the Internet at multiple levels. The highest level is the AS-level (**A**utonomous **S**ystem), an Autonomous System being a vast computer network operated by a single company, such as an ISP (**I**nternet **S**ervice **P**rovider). Not only the way Autonomous Systems communicate with each other has been investigated, but also their internal networks has drawn the attention of the research community, as many works focused on the mapping and the modeling of the router-level, i.e., how routers are interconnected either within a single AS or at the borders of adjacent ASes.

The Internet router-level has been investigated not only to discover its structure, but also to understand its dynamics. Starting from the 2000's, the research community focused on how the routers balance the traffic between several links to handle the increasingly large amount of network traffic, a process which is commonly denoted as load balancing (a form of traffic engineering). Additionnal works on the router-level aimed at characterizing *meshes* of routers, i.e., routers that are directly connected at the data link layer (or Layer-2) to manage a large number of links as if they were a single router. In the meantime, the research community also explored other levels of the Internet to complement router-level maps, such as a subnet-level, a subnet (short for *subnetwork*) being a group of network interfaces that can contact each other directly at the data link layer.

This thesis aims at designing new topology discovery techniques to efficiently map the intra-domain topologies of large networks by exploring their different levels and the relationships that exist between these levels, all while dealing with the effects of load balancing. Three different levels are investigated: the router-level, the subnet-level, and the hop-level. The hop-level characterizes how routers or meshes exchange packets at the network layer (or Layer-3), and can therefore be used to study the internal forwarding of a network without extensively discovering its router-level.

Not only this thesis provides new topology discovery schemes to map each level, but it also combines them to increase their accuracy. In particular, it introduces a new subnet inference methodology that takes advantage of alias resolution (i.e., the process of determining whether or not a group or pair of network interfaces belong to the same device) to solve ambiguous scenarios, as well as a topology mapping scheme that relies on both subnet-level data and alias resolution to build comprehensive hop-level maps of intra-domain topologies. Moreover, these maps can be interpreted with bipartite formalisms to more easily study their topological features. All topology discovery methods developed for this thesis are comprehensively elaborated and assessed in this document. The tools that implement these methods (`WISE` and `SAGE`), their source code, and the data they could collect on various Autonomous Systems are publicly available.

**Keywords:** topology discovery; fingerprinting; alias resolution; subnet inference; neighborhood; load balancing; bipartite graph; `WISE`; `SAGE`

# ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor Prof. Benoit Donnet for his patience, knowledge, understanding and availability. My Ph. D. has been a rocky path, but I have hopefully had a rock'n'roll supervisor who helped me keep things rolling with background, advice and research directions on any subject I worked on for the past six years, and I ultimately learned a lot from him. I have read and heard enough stories from other (former) Ph. D. students to know that few people have (or had) the chance to have a supervisor like him.

This thesis would probably not have been delivered in the present shape without the technical support from the SeGI (**Se**rvice **G**énéral d'**I**nformatique; the IT guys of the ULiège) and the other network operators who agreed to help me validate and improve my tools. Discovering the Internet topology is a very challenging task, because the scientist can never completely dismiss the possibility that his conclusions result from his own biases or errors, rather than from the actual properties of the measured networks. Being able to validate my tools on groundtruth networks was crucial, not only to publish my works, but also to ensure I was heading in the right direction. I would therefore like to thank Simon François, Nicolas Ghaye, Axel Bodart, Christophe Lemaire and Benoit Piret.

Last but not least, I would like to thank my family, colleagues, friends and other acquaintances for their company and support for the past six years. I am especially grateful towards the people who kept in touch and/or spent time with me (when it was possible) since the outbreak of the COVID-19 pandemic in early 2020. Without you, I have no idea how I would have gone through all this.

# Author's declaration

This thesis was written using the *University of Bristol Thesis Template* [21] created by V. BREÑA-MEDINA, and under the license *Creative Commons CC BY 4.0* [29]. The original template was modified in order to include a glossary and to allow separating the content of the document in several parts.

Several chapters of this thesis discuss the validation of multiple tools (including new tools developed for this thesis) when deployed to measure specific groundtruth networks. For confidentiality and security concerns, the groundtruth data used to validate the new tools and to compare them to earlier works will not be disclosed, even if the groundtruth data is outdated at the time of writing.

# Table of Contents

# LIST OF TABLES

# List of Figures

# Glossary

**alias candidate** An IP address considered as a potential alias for one or several other IP addresses.
**See also:** Sec. 2.2.

**contra-pivot (interface)** In a subnet, denotes an interface located on the ingress router.
**Comment:** measurement-wise, an interface may be a contra-pivot when its properties (trail, TTL distance) differ from those of the pivot interface(s) from the same address prefix.
**See also:** Sec. 5.1.2, Sec. 6.1.1.

**echoing (trail)** A trail is said to be echoing if it features no anomalies, if its associated IP address is identical to its target IP address and if this IP address occurs only once in the `traceroute` path, as the last hop before obtaining a reply from the target.
**See also:** Sec. 6.1.4.

**flickering (trail)** A trail is said to be flickering if it features no anomalies and if it happens, while considering a set of target IP addresses consecutive with respect to the address space and located at the same TTL distance, to appear in turn with a distinct anomaly-free trail.
**Comment:** IP interfaces from a same subnet may appear with two (or more) distinct trails, seemingly appearing in turns, as a result of symmetrical paths in a load balancer.
**See also:** Sec. 6.1.3.

**hop-level graph** Variant of router-level graph where each vertex can model either a single router or a mesh of interconnected routers, both appearing as a single hop in a `traceroute` path.
**See also:** Sec. 9.3.1.

**ingress router** Last router crossed by network packets before they reach a given subnet.
**See also:** Sec. 5.1.2.

**neighborhood** Location in a computer network bordered by a set of subnets that are located at most one hop away from each other.
**Comment:** in practice, a neighborhood consists of a single router or a mesh of routers (therefore a single hop), possibly connected with the help of Layer-2 equipment. A neighborhood is highlighted by subnets exhibiting equivalent trails, and is said to be **identified by** these trails.
**See also:** Sec. 4.1.1, Sec. 10.3.2.

**peer (of a neighborhood)** Given a neighborhood $N_a$ identified by a trail $t_a$, a second neighborhood $N_b$ identified by an anomaly-free trail $t_b$ is a peer of $N_a$ if and only if $t_b$ appears prior to $t_a$ in the `traceroute` paths towards the pivot interfaces of the subnets bordering $N_a$.
**Comment:** $N_b$ is said to be a **direct peer** if the difference in TTL distance between $t_a$ and $t_b$ is equal to 1. Conversely, $N_b$ is said to be an **indirect peer** if the difference in TTL distance between $t_a$ and $t_b$ is greater than 1.
**See also:** Sec. 10.3.2.

**pivot (interface)** In a subnet, denotes an interface that is not located on the ingress router.
**Comment:** measurement-wise, pivot interfaces exhibit similar properties (trail, TTL distance).
**See also:** Sec. 5.1.2, Sec. 6.1.1.

**router-level graph** Graph where each vertex accounts for a router and where each edge models the fact that two routers can exchange packets at the data link layer through a common link.
**See also:** Sec. 2.1, Sec. 9.3.1.

**trail** Given a target IP address, the trail towards this target is the last non-anonymous, non-cycling IP address appearing in the `traceroute` paths towards this target.
**Comment:** intermediate hops between an IP address and its trail are called **anomalies**. A trail is anomaly-free when it is the very last hop in the `traceroute` path before reaching the target.
**See also:** Sec. 6.1.1.

**trail (of a subnet)** Refers to the trail of the pivot interface(s) of a subnet.
**Comment:** a subnet can have multiple trails, e.g., if its pivot interfaces have (aliased) flickering trails. The trails of such a subnet can be accounted for with the alias list that encompass them.
**See also:** Sec. 10.3.2.

**TTL distance** The minimum **T**ime **T**o **L**ive value that should be used in an IP packet to get a reply from a given target IP address.
**Comment:** the TTL distance may vary depending on the type of packet.
**See also:** Sec. 5.1.2.

**vantage point** Computer from which a measurement tool is run.
**Comment:** the scientific literature also uses the synonymous term *monitor*.

**warping (trail)** A trail is said to be warping if it features no anomalies, if it appears for several distinct targets and if the TTL distances observed for each of these targets are varying.
**Comment:** Warping trails are likely a consequence of asymmetrical paths in load balancers.
**See also:** Sec. 6.1.2.

# Acronyms

| | |
|---|---|
| **AS** | Autonomous System. |
| **BE** | Best Effort. |
| **BGP** | Border Gateway Protocol. |
| **CIDR** | Classless Inter-Domain Routing. |
| **DAG** | Directed Acyclic Graph. |
| **DNS** | Domain Name System. |
| **FN** | False Negative. |
| **FP** | False Positive. |
| **ICMP** | Internet Control Message Protocol. |
| **IGMP** | Internet Group Management Protocol. |
| **INSIGHT** | Inferring Networks from Subnet Inference to GrapH Transformations. |
| **IP** | Internet Protocol. |
| **IR** | Intersection Ratio. |
| **ISP** | Internet Service Provider. |
| **IXP** | Internet eXchange Point. |
| **LAN** | Local Area Network. |
| **MPLS** | MultiProtocol Label Switching. |
| **OS** | Operating System. |
| **PLC** | PlanetLab Central. |

| | |
|---|---|
| **PLE** | PlanetLab Europe. |
| **RER** | Redundant Edge Ratio. |
| **RVR** | Redundant Vertex Ratio. |
| **SAGE** | Subnet AGgrEgation. |
| **TCP** | Transmission Control Protocol. |
| **TN** | True Negative. |
| **TP** | True Positive. |
| **TTL** | Time To Live. |
| **UDP** | User Datagram Protocol. |
| **VP** | Vantage Point. |
| **WISE** | Wide and lInear Subnet inferencE. |

# Chapter 1

# Introduction

The modern Internet is a vast and complex network of computers that deliver information and various services across the entire globe. Communications between distant computers is achieved by splitting the data to transmit into smaller chunks which are individually stored in data (or network) packets. These packets are then transmitted, one after the other, from the source to the destination, passing through multiple interconnected devices designed to send each received packet towards the next device that will get it closer to its destination – or towards the destination itself. Routers constitute the vast majority of such devices in the modern Internet, and could be described as numerical high speed post offices, as they read the destination address of any packet they receive on their ports to send them to the next device by picking the right outgoing port. This way of exchanging data between remote computers is often referred to as packet switching [75].

Theorized in 1964 [15], the packet switching paradigm was first implemented by the ARPANET (**A**dvanced **R**esearch **P**rojects **A**gency **NET**work) in 1969 [63]. At the time, the ARPANET only consisted of four university computers located in the United States that could exchange packets by interacting through four interconnected ***I****nterface* ***M****essage* ***P****rocessors* (IMPs) [30], i.e., the ancestors of routers. Figure 1.1 illustrates the early topology of the ARPANET as of December 1969 [94], with the IMPs being depicted by black dots. Being founded by the DARPA (**D**efense **A**dvanced **R**esearch **P**rojects **A**gency), the ARPANET was initially a US military project, but this would soon change: in 1972, Kahn proposed to transform the initial ARPANET into an open communication network by defining the basic rules that would allow computers to join the network regardless of its internal architecture [69]. Such a network would not be controlled by any central authority, and would follow a best effort delivery approach. These foundations would eventually lead to the creation of two of the most prominent network protocols, IP (**I**nternet **P**rotocol) [102] and TCP (**T**ransmission **C**ontrol **P**rotocol) [103], which were both first introduced, along with the term *Internet* itself, by Cerf and Kahn in 1974 [25].

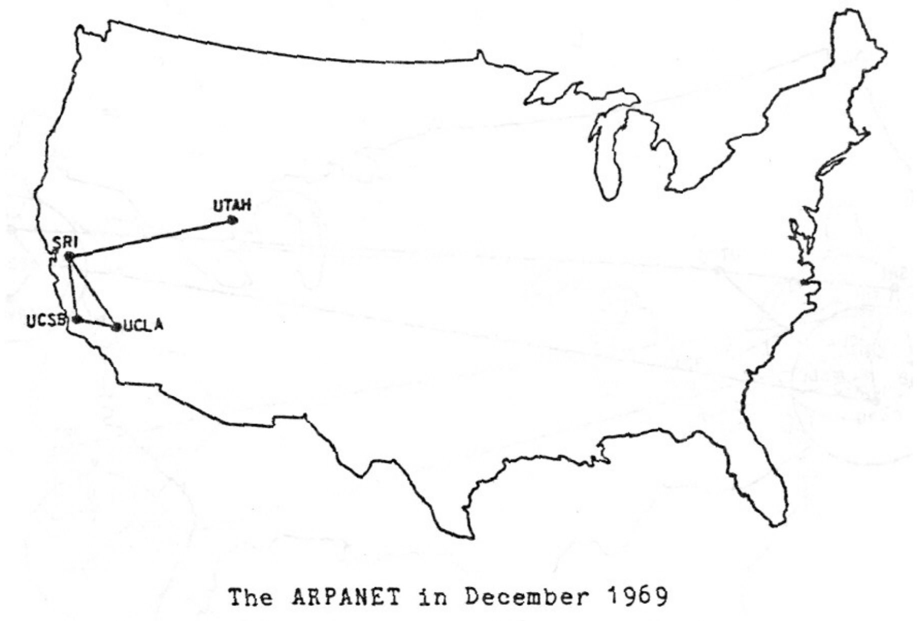


Figure 1.1: The early topology of the ARPANET (December 1969) [94].

The technologies developed with the ARPANET were later reused by the NSF (**N**ational **S**cience **F**oundation; based in the United States) to create the NSFNET (NSF **NET**work), which went online in 1986 [107]. The NSFNET initially connected multiple US-based university campuses together in order to allow researchers and engineers to gain remote access to the supercomputers of the time. The network quickly became congested after going online for the first time, and went through several significant hardware and software improvements to meet the demand, and eventually managed to connect two million computers by 1993 [107]. As the NSFNET opened to private companies and the general public in the early 1990's, via the first ISPs (**I**nternet **S**ervice **P**roviders), it eventually became the first backbone of the Internet as we know it. Figure 1.2 illustrates this backbone as of 1990 [42].

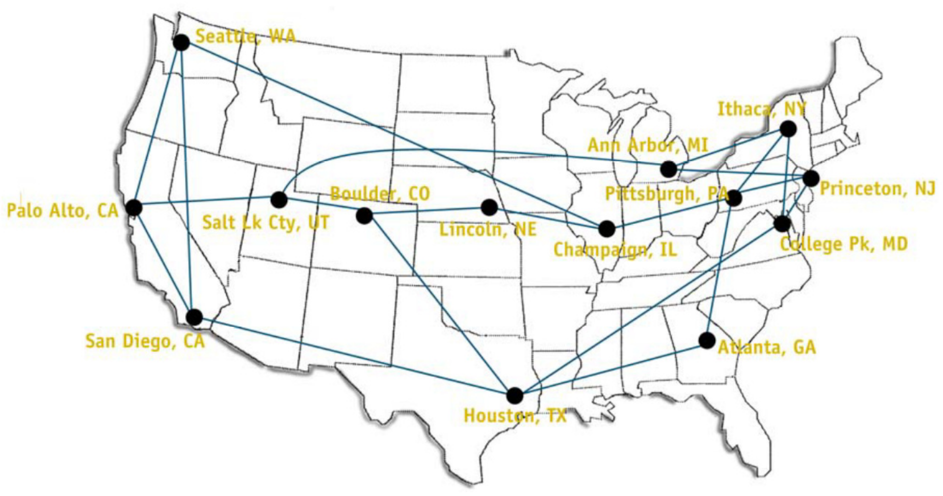


Figure 1.2: The backbone of the NSFNET (circa 1990) [42]. Black circles represent routers.

It is also in the early 1990's that the HTTP (**H**yper**T**ext **T**ransfer **P**rotocol) protocol was designed by Berners-Lee et al. to transfer formatted texts and various pieces of media over TCP/IP [17]. First elaborated by Berners-Lee in 1989 at the CERN (***C**onseil **E**uropéen pour la **R**echerche **N**ucléaire* [1]) in Geneva (Switzerland), the HTTP protocol was subsequently introduced to the research community then to the general public in 1991, effectively kickstarting the ***W**orld **W**ide **W**eb* (WWW), i.e., the information system providing various documents and pieces of media to the entire world through the Internet. The Web would eventually become one of the main selling points of the Internet to the general public, to the point that both are usually confused together in most languages.

Due to the growing popularity of the Web and subsequent technological developments (such as the rise of mobile devices), the Internet has become an extremely large and complex computer network since the 1990's: as of Spring 2021, more than 5 billion users were browsing between 40 and 50 billions of webpages indexed by the search engines Google and Bing [5, 32]. In 2020, the Cisco company announced that 18.4 billion devices were connected to the Internet in 2018, and forecasted that this number would rise to 29.3 billion devices by 2023 [27]. Because of its sheer size, the Internet is structured, at the highest level, as a large number of interconnected **A**utonomous **S**ystems (ASes), a single AS being a computer network operated by a same organization with a unified routing policy [64]. ASes are able to exchange packets between them with the help of BGP (**B**order **G**ateway **P**rotocol) [79], a protocol which allows each AS to advertise to its neighbors which routes towards various parts of the Internet it can offer to them. This mechanisms allows each AS to exchange packets with the entire Internet, as a packet exiting the AS will simply transit through other ASes until reaching its destination.

In 2017, CAIDA (**C**enter for **A**pplied **I**nternet **D**ata **A**nalysis [2]) offered insight on the complexity of the current Internet by illustrating an *AS core*, i.e., a graph that models how the Autonomous Systems of the IPv4 Internet are interconnected [23]. CAIDA later released an updated *AS core* in April 2021, the new render being representative of the IPv4/IPv6 Internet as of January 2020 [24]. Figure 1.3 provides a glance at this latest AS core and clearly demonstrates how much more complex the Internet became over time, especially knowing Figures 1.1 and 1.2 only illustrate a few routers.

Because the Internet is designed to allow any computer to join it regardless of its architecture [69], how the Internet is actually structured has become a whole research topic on its own as it became more complex. In particular, while a network operator administrating a specific AS may know all the technical details of its own network, he may not be aware of the underlying architecture of the neighbor ASes. And since no central authority administrates the entire Internet, its structure and dynamics have become elusive, especially given that most (if not all) ASes do not provide public data on their respective internal network. This is why the scientific community has elaborated various data collection mechanisms to gain insight on how the modern Internet actually works, this specific domain being often referred to as ***Internet topology discovery*** [37, 61]. Topology discovery aims not only at explaining how the modern Internet actually works, e.g., to explain some of its temporary

[1] Also known in English as the European Organization for Nuclear Research.

[2] CAIDA is part of the University of California based in San Diego, California.

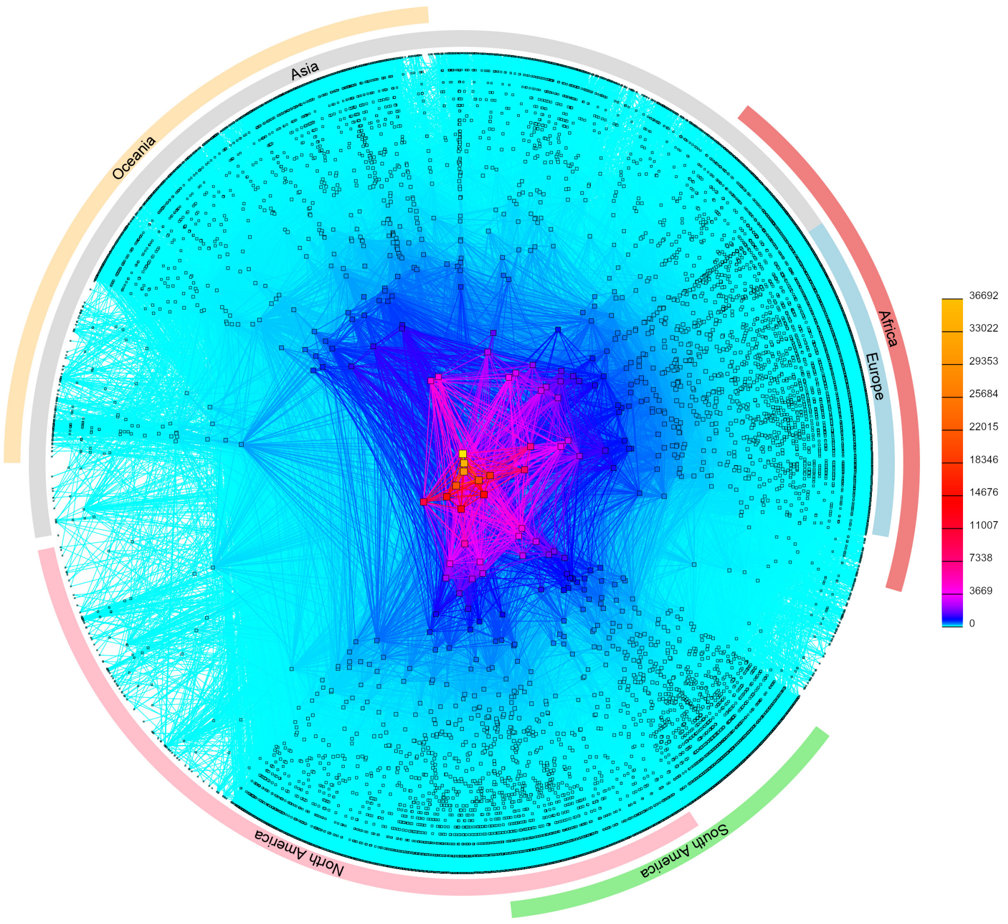


Figure 1.3: CAIDA's *AS core* (IPv4/IPv6 Internet) in January 2020 [24].

issues [115], but also at gathering realistic data on how the Internet works in practice, so that this data can be used to design and assess new technologies, such as new network protocols.

To study the modern Internet, the research community has naturally investigated the AS-level, pictured by CAIDA's *AS core* in Figure 1.3, since the dawn of Internet topology discovery. Typically, each Autonomous System has a certain number of neighbors, i.e., adjacent ASes to which it will forward network packets depending on the routes advertised by each of these ASes via the BGP protocol [79]. The way an AS interacts with its neighbors has notably motivated the community into establishing a classification that accounts for the economic relationships that exist between adjacent ASes: a ***Tier-1*** AS refers to an Autonomous System which exclusively acts as a provider to its neighbors, and conversely, a ***Stub*** AS only behaves as a client to its neighbors. The ***Transit*** ASes constitute a middle ground, as they can assume the role of a provider or be a client in respect of their neighbors. Of course, the classification of an AS will partly be correlated with its role in the Internet: for instance, Tier-1 ASes are usually providing a vast number of routes to (dozens of) thousands of other ASes. Many works in topology discovery aimed at mapping the AS-level of the Internet [37, 61], and in particular, a study conducted by Dimitropoulos et al. in 2006 investigated the discovery of the inter-AS relationships [36]. The routing dynamics between ASes have a considerable impact on the routing at the scale of the Internet, and in particular, a 2009 study by Magnien et al. demonstrated that this may cause the observed routes to never be stable [82]. To better study the AS-level and its dynamics, CAIDA publicly released a dataset about inter-AS relationships in 2013 [22]. In 2017, CAIDA also computed a ranking of all ASes, based on their respective number of neighbors [2] (inferred from various data sources), which notably served as a basis for their *AS cores* [23, 24].

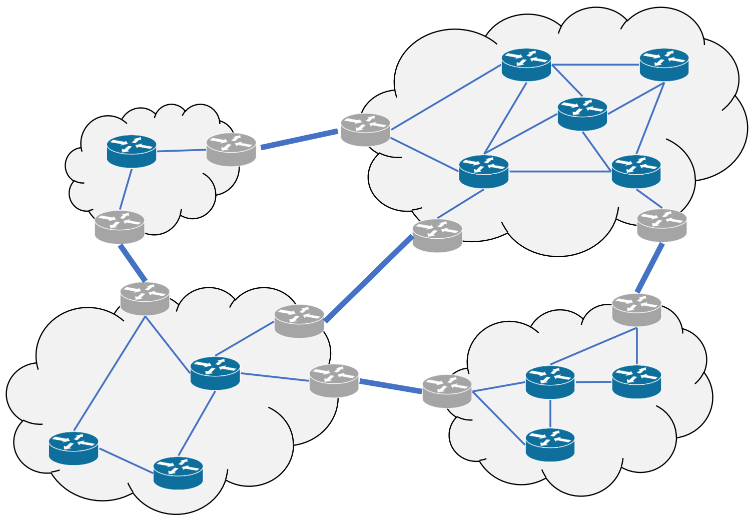


Figure 1.4: A toy network as seen from the AS- and router-levels.

However, studying the AS-level only is not sufficient to explain how the modern Internet works. To give a practical example, in June 2015, Telekom Malaysia (AS4788) announced routes towards approximately 179,000 prefixes to Level3 (AS3549, a major Tier-1 AS), which accepted these routes, causing a massive amount of traffic to be forwarded through AS4788. Because this AS did not have the architecture to handle such a large amount of traffic, many packets were lost and the entire Internet was slowed down as a consequence [115]. Such an example clearly demonstrates that studying intra-domain topologies is also of importance to have an idea of the inner architecture of individual ASes as well as the amount and type of traffic they have been built to handle. Intra-domain topologies can be studied from the perspective of the router-level. In a ***router-level graph***, each vertex accounts for an individual router while each edge models the fact that two routers can exchange packets at the data link layer through a common link. Figure 1.4 illustrates both the AS-level and the underlying router-level graph of a toy network, with the clouds depicting individual ASes and the vertices and edges illustrating routers and the links they share (respectively). Grey routers symbolize AS Border Routers (or ASBRs), i.e., routers that handle inter-AS connections via the BGP protocol [79].

Since the 1990's, many strategies have been elaborated by the research community to map intra-domain topologies. The most basic tool used for this purpose is none other than `traceroute`, which has been developped by Jacobson in 1989 [119]. `traceroute` is able to discover the interfaces of the intermediate routers that exist between the ***monitor*** (also called ***vantage point***), i.e. the computer where it runs, and the destination (usually represented by an IP address). To do so, it primarily relies on the TTL (**T**ime **T**o **L**ive) field of the IP header [102]. This field simply consists of an 8-bit integer value which is decremented each time it arrives at a new device (e.g., a router). When the TTL value drops to 0, and if the receiving device differs from the destination, the packet is not further forwarded. Initially, the purpose of the TTL field is to prevent a packet from cycling indefinitely in the network if it cannot reach its destination. Routers who receive a packet whose TTL value has expired (i.e., has dropped to 0) typically replies to the source with an ICMP `time-exceeded` reply, which the source will be one of the interfaces of the router itself, represented by an IP address [101]. By sending UDP (**U**ser **D**atagram **P**rotocol) [100] packets [3] to the target while increasing the TTL value at each probe, a monitor running `traceroute` can collect the interfaces of each router on the way to the destination, which is traditionally referred to as a `traceroute` ***path*** or ***route***. Packets used to discover network interfaces (in the case of `traceroute`) or other network properties are more broadly referred to as ***probes***. Figure 1.5 illustrates `traceroute` probing on a toy topology, where the circles depict router interfaces and the arrows represent the `traceroute` probes and their respective paths. Many topology discovery tools and systems have relied on `traceroute` to collect network paths and/or discover router interfaces and map the corresponding router-level topology [18, 45, 83, 113].

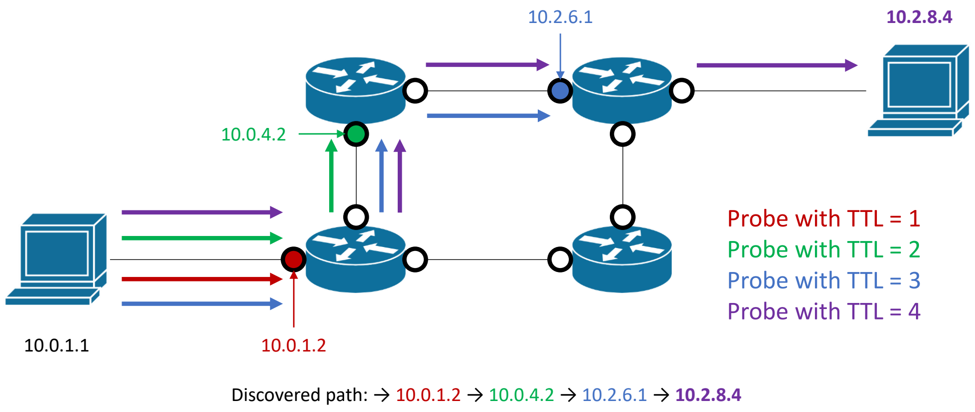


Figure 1.5: How `traceroute` works. Circles represent interfaces while arrows depict probes.

Because routers feature multiple interfaces, the research community has also designed ways to identify whether two IP interfaces belong to the same router. This process of determining if two or more IP addresses (router interfaces) belong to the same device is referred to as ***alias resolution***. Alias resolution is crucial for the accuracy of router-level maps, because a collection of network paths discovered via `traceroute` probing from different monitors, or even from the same monitor, may contain differing interfaces that belong to the same router. Therefore, bad or incomplete alias resolution can suggest topologies that vastly differ from the reality [57]. Numerous alias resolution methods have been designed by the community to identify routers since the 1990's, with some of these methods being embedded in larger topology discover systems [71–74, 86, 87, 113].

[3]Other tools may use protocols other than UDP to perform `traceroute` probing.

However, alias resolution is not the only way to discover routers that has been investigated over time. Starting from the middle of the 2000's, several works have taken advantage of IGMP (**I**nternet **G**roup **M**anagement **P**rotocol) [33] probing. This method relies on specific messages of the IGMP protocol that allow to discover the interfaces and the adjacencies of a multicast-enabled router. Though this method only works for multicast routers, it allows to discover both the interfaces and the adjacent routers of a given router without performing alias resolution. IGMP probing has been used to study inter-AS links [91], to map intra-domain topologies [97], and to infer Layer-2 equipement and its influence on routers [90]. Unfortunately, IGMP probing has become obsolete due to the evolution of filtering policies [84], i.e., routing policies that are intentionally dropping packets matching certain patterns as a way to reinforce the security of systems connected to the Internet. As a consequence, alias resolution is still crucial to map the router-level of intra-domain topologies.

Starting from the 2010's, many works have also explored other levels of the Internet to complement the usual AS- and router-level perspectives. For instance, a few works have focused on the identification of IXPs (**I**nternet e**X**change **P**oints), i.e., network structures which were built to facilitate the interconnection of multiple ASes (such as ISPs) to avoid relying on intermediate transit networks [13, 95]. Other works aimed at mapping PoPs (**P**oints-**o**f-**P**resence), i.e., demarcation (sometimes physical) points where multiple differing networks share a connection [41, 109]. Finally, multiple publications have put forward the topic of ***subnet inference***, i.e., topology discovery methods that aim at identifiying subnets (short for *subnetworks*). A subnet is a group of devices (each represented by one network interface) which are directly connected at the data link layer, i.e., they can contact each other directly without having to send packets to one or more intermediate routers. Subnets can constitute point-to-point links between routers (i.e., a physical link between two routers) or LANs allowing multiple routers to reach each other in a convenient manner. As such, subnet inference techniques have been proposed by the research community to annotate intra-domain router-level maps with the subnets that connect the routers together [58, 70, 116, 118].

Though the topology discovery literature provide numerous techniques to discover various network components, the very task of mapping intra-domain topologies remain challenging. In particular, several alias resolution methods have proved to offer limited accuracy [126] or to be difficult to deploy at the scale of the Internet [16]. Other methods have been rendered practically impossible to use as a consequence of the evolution of filtering practices. In particular, alias resolution methods [31, 86, 110] and `traceroute` variants [111, 112] based on IP options [102] have become obsolete due to the IETF (**I**nternet **E**ngineering **T**ask **F**orce) recommending to drop all IP packets featuring options since February 2014 [44].

One of the major challenges of topology discovery as a whole is ***traffic engineering***. Traffic engineering denotes a collection of methods that have been elaborated over the years by the networking community to deal with the always increasing amount of traffic. One of the most common forms of traffic engineering is ***load balancing***, i.e., the process of sharing the packets to transmit between two network locations among multiple paths, each path often consisting of multiple intermediate routers.

Different load balanced paths are usually preceeded by a ***divergence point***, i.e., a device where the disjointed paths begin, and can be followed by one or more ***convergence points***, i.e., devices where two or more load balanced paths rejoin. Load balancing allow networks to handle a large amount of traffic without increasing the capacity of the links themselves, as replacing these links is usually a costly operation for network operators.

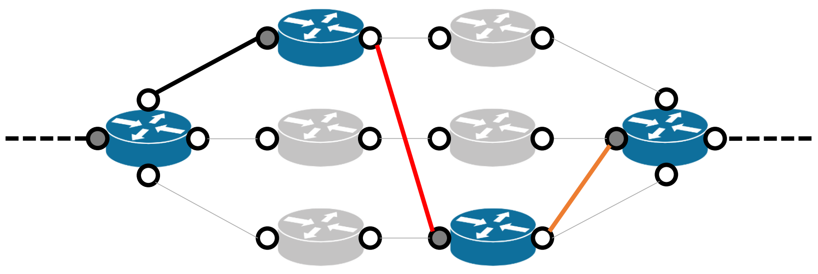


Figure 1.6: The false link discovery problem on a toy load balancer.

Load balanced paths are a challenge for topology discovery because they can induce ***false link discovery***, i.e., the detection of network links that do not exist in the real world. For instance, two consecutive `traceroute` probes can take disjointed load balanced paths, and as a result, the discovered router interfaces will not correspond to an actual network path. Figure 1.6 illustrates this problem with a toy load balancer consisting of three load balanced paths of the same length. In this figure, the leftmost router represent a divergence point, the rightmost router model a convergence point, and the grey circles depict the router interfaces detected via `traceroute` probing. The red edge depicts a link that could be inferred from the discovered path which does not exist in the real network, while the orange edge highlights an inferred link connecting two routers that do share a common link, but not via the discovered interface (i.e., the grey circle on the rightmost router, or convergence point).

Since the middle of the 2000's, several works aimed at identifying and characterizing load balancers in the Internet. Paris `traceroute`, introduced by Augustin et al. in 2006, is a `traceroute` variant that manipulates the headers of the probe packets to stabilize the paths in per-flow load balancers (i.e., to always go through the same load balanced path) [11]. By designing new algorithms taking advantage of Paris `traceroute`, Augustin et al. could detect and study load balancers, and highlighted the existence of symmetrical and asymmetrical load balanced paths [14]. Symmetrical paths refer to load balanced paths that exhibit the same number of routers and links (or hops) in respect of the parallel paths within the same load balancers, much like the load balanced paths illustrated in in Figure 1.6, while asymmetrical paths provide varying number of such components. As a consequence, asymmetrical paths can notably alter the evaluation of distances towards remote interfaces as TTL values, i.e., the minimal TTL value to get a reply from a target remote interface might vary at each probe because of the asymmetrical paths. More broadly speaking, load balancing can impair subnet inference [116] or suggest network structures that do not actually exist [85]. Finally, further research with Paris `traceroute` demonstrated that, as of the begininng of the 2010's, the vast majority of `traceroute` paths will cross at least one load balancer [12].

## 1.1 Research directions

This thesis aims at advancing the state-of-the-art of topology discovery in at least three research directions. First of all, it will explore how multiple topology discovery techniques can be best combined to complement each other, in a holistic manner, as detailed in Sec. 1.1.1. Second, most topology discovery methods elaborated for this thesis will address the problem of mapping a target network at different levels while dealing with the effects of traffic engineering, specifically load balancing, as discussed in Sec. 1.1.2. Finally, Sec. 1.1.3 shortly discusses a third research direction: envisioning new applications of bipartite graphs for network modeling.

### 1.1.1 Holistic topology discovery

An issue with the state-of-the-art of topology discovery is that many tools developped by the research community tend to focus on very specific tasks. For instance, a whole branch of the literature of alias resolution focus on how to best use the IP identification field of the IP protocol [102] to infer routers, and multiple alias resolution techniques have been elaborated partly or solely on that basis [16, 73, 113]. Unfortunately, alias resolution based on the IP identification field is not always possible, as routers may very well implement different behaviours than those expected by state-of-the-art alias resolution tools to handle this header field, as will be explored in this thesis. This makes dedicated tools, such as `MIDAR` (**M**onotonic **ID**-Based **A**lias **R**esolution) [73], unable to handle specific scenarios of alias resolution. It is therefore desirable to envision multiple approaches at the same time to achieve accurate and exhaustive alias resolution.

It should be noted that combining multiple alias resolution techniques is not unheard of, as the topology discovery systems `Rocketfuel` [113] and `AROMA` (**A**ccurate **RO**uter-level **MA**p) [74] already complemented methods based on the IP identification field with the analysis of DNS records, respectively in 2002 and 2007. The authors of `PalmTree` [117] (2011), an analytical alias resolution technique (i.e., discovering aliases by analyzing data such as `traceroute` paths *a posteriori*), presented their work as a tool for complementing other alias resolution methods. Finally, the alias resolution tool `APPLE` (**A**lias **P**runing by **P**ath **L**ength **E**stimation) [87], which has been introduced to the community by Marder in 2020 [4], also combines two different alias resolution methods (`iffinder` [71] and `MIDAR` [73]) with an analytical method to outperform both methods when used separately.

This thesis will nevertheless further the idea, not only in the context of alias resolution, but also at a broader scope. In other words, it will explore the idea of connecting several topics of topology discovery together such that each topic complements the shortcomings or reduces the difficulty of each other. This thesis will therefore elaborate ***holistic topology discovery***, i.e., an approach to topology discovery that considers its various challenges (such as alias resolution and subnet inference) as the interacting parts of a broader problem, rather than as completely separate topics.

[4] In fact, `APPLE` is posterior to the research work on alias resolution presented in this thesis.

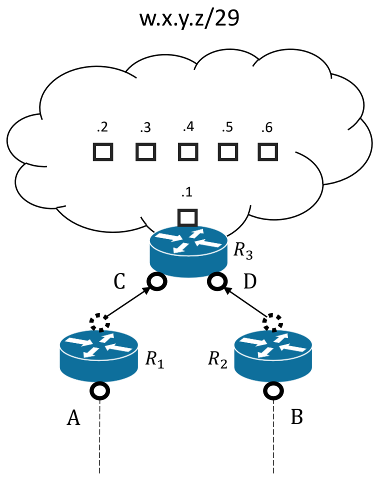
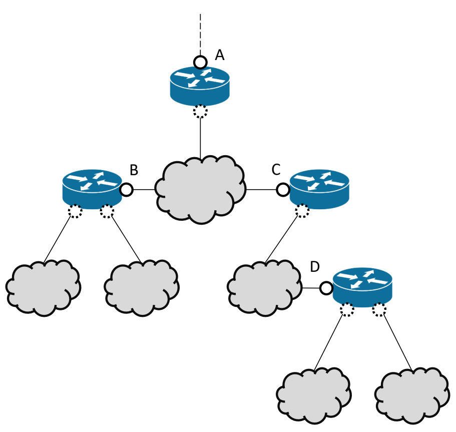


Figure 1.7: A toy subnet inference scenario where alias resolution could prove useful.

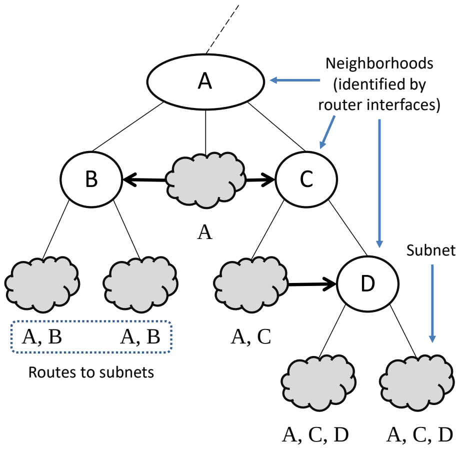
For instance, alias resolution could prove benefitial to subnet inference. Indeed, several works in subnet inference identify subnets partly by identifying the last router crossed before reaching subnet interfaces with the help of `traceroute` probes. In particular, several IP interfaces being associated with the same preceding router (interface) may very well be on the same subnet [116, 118]. However, the last router before a specific subnet, also called ***ingress router***, may be reached through multiple paths as a result of load balancing [12, 14] (see also Sec. 1.1.2). This may cause differing interfaces of the same router to reply to the `traceroute` probes. As a result, the ingress router appears as multiple distinct router interfaces rather than as a single interface. By performing alias resolution on these interfaces, it becomes possible to identify the ingress router by an alias list in order to continue subnet inference with the traditional assumptions. Figure 1.7 shows a toy subnet inference scenario where the ingress router before the subnet `w.x.y.z/29` could appear in `traceroute` data as two distinct interfaces $C$ and $D$. Aliasing these interfaces as $\{C, D\}$ would allow to accurately discover `w.x.y.z/29`. This simple example suggests that alias resolution and subnet inference should not be seen as separate, monolithic topics, as the former may be useful to the latter.

Another example of topology discovery topics complementing each other can be found in `TreeNET` [55]. `TreeNET` is actually the main contribution of my master thesis, which I carried out just before this thesis (during academic year 2014-2015), and is a subnet-based topology mapping tool that builds a tree-like map of a target network (hence its name) using the routes (collected with Paris `traceroute` [11]) towards a collection of inferred subnets (first collected with `ExploreNET` [118], then refined by `TreeNET` [55]). `TreeNET` takes advantage of the idea that a collection of `traceroute` paths should be close to a tree-like structure, and therefore uses the router interfaces found in each path as nodes of the tree while the subnets constitute leaves [5]. A direct benefit of the tree-like map is that the internal nodes correspond to ***neighborhoods***, i.e., network locations bordered by subnets located at most one hop away from each other. On a simple network topology, neighborhoods will

[5]The methodology of `TreeNET` will be detailed later in this thesis.



(a) Toy network (circles depict interfaces)



(b) As a tree (clouds depict subnets)

Figure 1.8: Mapping a simple network with `TreeNET` [55]. Each subnet has a single `traceroute` path.

typically act as a first approximation of the routers, which means the tree-like map accounts for the router-level of the target network. Figure 1.8 illustrates a simple toy network (Fig. 1.8a) as it would be interpreted by `TreeNET` (Fig. 1.8b), each router appearing as a neighborhood.

What is interesting with `TreeNET` in respect of holistic topology discovery is its methodology: indeed, `TreeNET` builds its tree-like map and discovers the neighborhoods of the target network on the basis of its subnets and their respective (Paris) `traceroute` paths. This shows that subnet inference can be more than just a way to complement existing network maps, as put forward by previous research [58, 116, 118], as it can constitute a basis for building a network graph that is representative of the internal routing of the target network. Subnet inference brings other benefits: in Fig. 1.8b, black arrows are connecting some subnets with one or several router interfaces, each corresponding to an internal node (and therefore identifying a neighborhood) found at the same depth. These arrows symbolize the fact that a subnet encompasses the pointed router interface, which suggests this subnet corresponds to a link between two adjacent routers. In other words, subnet inference data can not only kickstart network graph inference, but also make it more exhaustive. While `TreeNET` already takes advantage of the interactions between subnet inference and network graph inference, its limits (see Sec. 1.1.2) eventually motivated the design of better tools that deepen these interactions and which will constitute the main topics of later chapters.

Finally, this thesis will also explore the possibility of using neighborhoods to perform accurate alias resolution. Indeed, in the modern Internet, some locations that appear as single hops in `traceroute` paths may actually be made of several devices connected in mesh that still act as a single hop in respect of packet forwarding. In particular, previous research evidenced that the total number of adjacent devices of an inferred router may be overestimated due to the presence of Layer-2 devices binding several routers into meshes [90], which still behave as single hops. This modern issue also suggests a change of perspective, i.e., to move from the router-level to the hop-level. A ***hop-level graph*** is therefore a variant of router-level graph where each vertex can be either a single router or a mesh of routers. As such, the concept of neighborhood amounts to a measurement-based definition of a network hop. As will be detailed in later chapters, neighborhoods and their respective bordering subnets can suggest potential aliases, which can be used to speed up the discovery of the underlying router-level of the target network. In summary, network graph inference can simplify the problem of alias resolution, which also means subnet inference indirectly helps alias resolution.

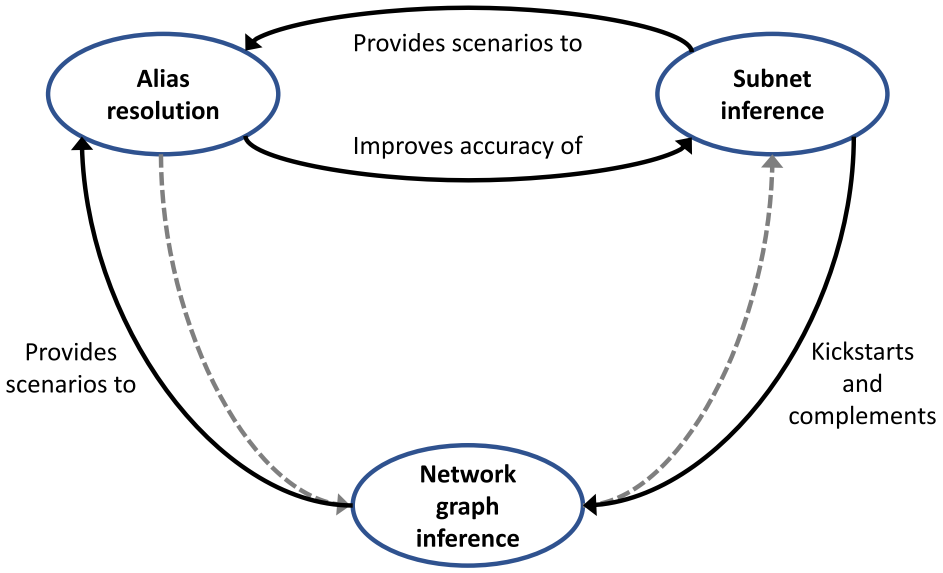


Figure 1.9: A first glance at an holistic approach to topology discovery.

Figure 1.9 provides a first schematic summary of holistic topology discovery as it will be discussed in this thesis. The grey dashed arrows correspond to interactions that are yet to be mentioned and explored, while the plain arrows symbolize interactions between topics that have already been mentioned in this very section and will be detailed in later chapters.

### 1.1.2 Topology discovery in a world of load balancing

A major challenge of topology discovery in today's Internet lies in dealing with the effects of traffic engineering, and more precisely load balancing. Indeed, previous research notably showed that `traceroute` paths may never be stable and will almost always cross a load balancer [12, 82]. Because of this, some assumptions made by state-of-the-art topology discovery tools need to be re-evaluated. In particular, Fig. 1.7 in Sec. 1.1.1 already showed the consequences of symmetrical load balanced paths for subnet inference and how they could be handled with alias resolution to accurately discover the example subnet. A modern subnet inference methodology should also leave some room for the effects of asymmetrical load balanced paths, as state-of-the-art subnet inference tools tend to assume subnet interfaces from a same subnet will appear at similar distances in TTL values [116, 118]. Because of this assumption, the same tools struggle to detect subnets within a target domain where asymmetrical load balancing is experienced (either within the target network itself or within an intermediate network), a shortcoming which was already acknowledged by their authors [116].

The effects of traffic engineering, and more precisely load balancing, also impair the topology mapping of a target network as performed by `TreeNET` [55] (see also Sec. 1.1.1). In particular, previous work in topology discovery evidenced that a set of `traceroute` paths will be closer to a directed acyclic graph of the IP-level [98]. `TreeNET` partially accounts for this fact by ensuring each node in the tree only features one predecessor, and manages this by allowing the internal nodes of a tree-like map to be labeled with more than one IP interface. Multi-label nodes happen when router interfaces vary at given hop counts in `traceroute` paths that otherwise feature the same length and identical last hop(s) [6]. Thanks to this trick, the tree-like map also highlights interfaces of routers that are potentially involved in load balancing architectures [50].

Unfortunately, the methodology of `TreeNET` is not well suited to measure complex networks where load balancing is frequently experienced. Among others, the internal nodes meant to model router interfaces featured in (symmetrical) load balanced paths can be difficult to interpret. And because of asymmetrical paths, subnets can have routes featuring the same last hops but exhibiting differing path lengths, resulting in these subnets bordering distinct neighborhoods rather than the same one. One of the main motivations of this thesis was therefore to comprehensively elaborate neighborhood-based topology mapping while dealing with the effects of traffic engineering in order to overcome the `TreeNET` limits. A key challenge of this task lies not only in how to handle `traceroute` paths in general, but also in how and when alias resolution should be used to find out whether or not some router interfaces detected through `traceroute` probing actually belong to the same device. Interestingly, this latter requirement further justifies the holistic approach discussed in Sec. 1.1.1, and corresponds to the dashed arrow going from alias resolution to network graph inference. As mentioned before, this is critical to achieve accurate topology discovery, as bad or incomplete alias resolution may result in maps vastly differing from the real network [57].

### 1.1.3 Modeling the Internet with bipartite graphs

Finally, while most of this thesis will present and assess new topology discovery methods, it will also briefly discuss a third research direction: modeling the Internet topology with the help of ***bipartite graphs***. In graph theory, a bipartite graph is a graph where vertices are distributed into two disjointed sets (or parties) such that each edge connects one vertex from the first party to a vertex from the second party. An example of bipartite graph is illustrated by Figure 1.10. Bipartite graphs have been widely studied by the scientific community, notably to model social structures, such as those of file-sharing communities [66, 93, 124].

Because the Internet, at its core, consists of routers interconnected by links or LANs which can be easily modeled by subnets, bipartite graphs are naturally suited to model it. In fact, Guillaume et al. already mentioned using bipartite graphs to model the Internet in 2006 [56], but it is not before 2013 that an actual bipartite model would be introduced. That year, Tarissan et al. [114] proposed a model where the two parties respectively represent Layer-2 devices (such as Ethernet switches) and

[6]This will be detailed later in this document.

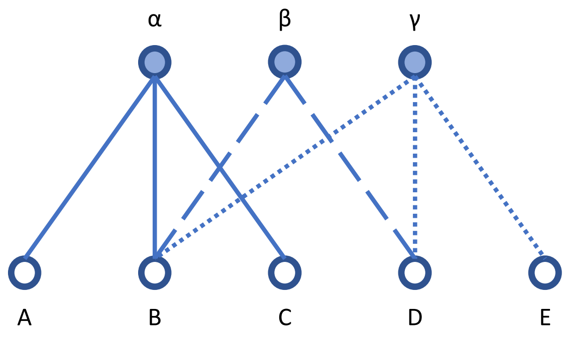


Figure 1.10: An example of bipartite graph. The first party consists of $\alpha$, $\beta$ and $\gamma$.

routers, and thoroughly assessed it using data collected via IGMP probing [83, 91, 96, 97]. One of the motivations of this thesis therefore lies in exploring bipartite formalisms that include subnets in the model to better account for how routers are connected within intra-AS topologies.

## 1.2 Main contributions

This thesis intends to advance the state-of-the-art in respect of the research directions detailed in Sec. 1.1 by answering the following research questions.

1. **Research question 1:** Can multiple topology discovery methods be combined in a holistic manner to improve both the accuracy and the exhaustivity of the collected data ?
2. **Research question 2:** Can topology discovery methods be designed so that they can deal with the effects of traffic engineering ?
3. **Research question 3:** Can the topology of a large target network be captured from a single vantage point ?
4. **Research question 4:** What can we learn from the collected data if we are modeling the measured networks with bipartite/tripartite graphs ?

Research question 1, concerning holistic topology discovery, will be answered in multiple ways throughout this thesis.

- A generic alias resolution framework will be introduced. This framework not only provides multiple state-of-the-art methods, but also takes advantage of IP fingerprinting [121] to identify which methods can be used on a given scenario and which one is the most suitable and/or accurate. When combined with space search reduction (i.e., breaking down the initial problem into smaller problems), notably by taking advantage of the subnet-based tree-like map of `TreeNET` [55], this framework can offer accurate alias resolution while being lightweight in terms of probing, as will be demonstrated with a validation on a groundtruth network [51].

Deployment of this framework in the wild from the PlanetLab testbed [8] will also be used to re-assess the permeability of the Internet to various alias resolution methods [51].

- To advance the state-of-the-art of subnet inference, a new tool called `WISE` (**W**ide and l**I**near **S**ubnet inferenc**E**) will be introduced [52]. This tool takes advantage of the aforementioned alias resolution framework to better identify the ingress router of each subnet, therefore setting it apart from state-of-the-art tools, which rely on *ad hoc* methods that do not consider other topology discovery topics. Measurements in the wild from the PlanetLab testbed [8] with `WISE` will demonstrate that integrating alias resolution in subnet inference is a sound idea [52, 53].
- Finally, the topology mapping tool `SAGE` (**S**ubnet **AG**r**E**gation) will be presented [54]. This tool combines subnet inference, `traceroute` probing and alias resolution in a holistic way to map a given target network as a neighborhood-based **D**irected **A**cyclic **G**raph (DAG) that accounts for its hop-level topology. Measurements on groundtruth networks will demonstrate the `SAGE` methodology can capture snapshots of networks that are true to their actual topology [54].

Research question 2, which consists of determining how the effects of traffic engineering can be dealt with, will also be commented on and answered to more than once/

- In order to elaborate the methodology of `WISE`, the typical consequences of traffic engineering regarding subnet inference will be characterized and quantified with the help of measurements conducted on various Autonomous Systems (ASes) from the PlanetLab testbed [8]. By doing so, it will be demonstrated that the effects of traffic engineering are not only unavoidable in the modern Internet [12], but also vary considerably from one vantage point to another [52, 53].
- By taking account of these effects in the subnet inference methodology of `WISE`, it will be demonstrated that it is possible to discover a majority of (sound) subnets regardless of the vantage point and despite the variations in the observed effects of traffic engineering [52, 53].
- The methodology of `SAGE` will also be carefully elaborated throughout this thesis. First, the basic concepts of `SAGE` (including the concept of neighborhood introduced with `TreeNET` [55]) will be extensively described and assessed in order to demonstrate they are reliable enough to use them as the building blocks of network graphs, despite these concepts relying on partial `traceroute` paths [53]. Second, the complete methodology of `SAGE` will be described to show how it can infer neighborhoods and their adjacencies, still using not only partial `traceroute` paths but also alias resolution to detect convergence points of load balanced paths. Finally, by measuring various Autonomous Systems in the wild from the PlanetLab testbed [8] with `SAGE`, it will be demonstrated that `SAGE` can capture comparable graphs of a same target network, in terms of both vertices and links, despite changing the vantage point [54].

Research question 3, on making the topology mapping methods efficient enough to easily capture a large network, will be answered to both in a practical and an analytical manner.

- The complete workflow of both `WISE` [52, 53] and `SAGE` [54] will be comprehensively described, as each main algorithm will be explained in details along with its pseudo-code. To complement this detailed presentation of both tools, Appendix A will demonstrate that all the algorithmical steps of `WISE` achieve a linear time complexity, and that the additional steps taken by `SAGE` to infer the hop-level graph of a target network can be carried out in quasi-linear time. The time complexity analyses will be completed from time to time by experimental data to show why the provided algorithms are expected to behave well on average (if not all the time).

- `WISE` will be validated on two groundtruth networks to show that, not only it can outperform previous state-of-the-art subnet inference tools (`TreeNET` [55] and `ExploreNET` [118]) in terms of accuracy, but also in terms of total execution time, mostly thanks to its design. Deployment of `WISE` in the wild will include measurements of large ASes [7] from a single vantage point on a quasi-daily basis, a feature that will ease the comparison of the collected data [52, 53].

Finally, research question 4, about the possibilities offered by bipartite/tripartite graphs to study the Internet topology, will be answered to at the end of this thesis.

- Data collected with `SAGE` from both the PlanetLab testbed [8] and the EdgeNet cluster [108] will be transformed into neighborhood – subnet bipartite graphs. It will be subsequently demonstrated that such bipartite graphs can offer insight on the role played by subnets, as well as the shape of a network, its underlying structures (e.g., back-up links) and the local density of its (major) components [54].

- Finally, a tripartite model, designed to better account for the devices involved in the topology of a target network, will be presented and commented on. In particular, it will be showed that such a formalism has the potential to offer a more accurate picture of the router-level of a target domain while suggesting where Layer-2 equipment is likely to be used.

## 1.3 Publications

The publications that have been written in the context of this thesis are listed below. I am the first author for all of them, as I did the research work and measurements presented in each paper, though a major part of the writing has been done by my supervisor B. Donnet for the first three papers, the first one also being partly written by my former master thesis supervisor F. Tarissan [8]. I did most of the writing in the last three publications (and all of it for the last one).

[7]Or very responsive, i.e., many IP addresses within their prefixes reply to direct probes.

[8]F. Tarissan supervised me during an internship at the LIP6 (***L**aboratoire d'**I**nformatique de **P**aris **6***, a research laboratory located at the university Pierre and Marie Curie in Paris) for my master thesis in Spring 2015. He was a lecturer (*maître de conférences*) at the time and now works for the CNRS (***C**entre **N**ational de la **R**echerche **S**cientifique*).

- *TreeNET: Discovering and Connecting Subnets* [55]
  **J.-F. Grailet**, F. Tarissan, B. Donnet
  Proceedings of Traffic and Monitoring Analysis Workshop (TMA), April 2016
- *Towards a Renewed Alias Resolution with Space Search Reduction and IP Fingerprinting* [51]
  **J.-F. Grailet**, B. Donnet
  Proceedings of Network Traffic Measurement and Analysis Conference (TMA), June 2017
- *Discovering Routers in Load-balanced Paths* [50]
  **J.-F. Grailet**, B. Donnet
  Proceedings of ACM CoNEXT Student Workshop (CoNEXT), December 2017
- *Revisiting Subnet Inference WISE-ly* [52]
  **J.-F. Grailet**, B. Donnet
  Proceedings of Network Traffic Measurement and Analysis Conference (TMA), June 2019
- *Virtual Insanity: Linear Subnet Discovery* [53]
  **J.-F. Grailet**, B. Donnet
  IEEE Transactions on Network and Service Management (TNSM), Volume 17, Issue 2, pp. 1268–1281, June 2020
- *Travelling Without Moving: Discovering Neighborhood Adjacencies* [54]
  **J.-F. Grailet**, B. Donnet
  Proceedings of Network Traffic Measurement and Analysis Conference (TMA), September 2021

## 1.4 Outline

For convenience's sake, this thesis will be divided in three parts. The first part will discuss the topic of **alias resolution** in three chapters. Chapter 2 will review the state-of-the-art of alias resolution and comment on its limitations. Chapter 3 will subsequently introduce an alias resolution framework designed to overcome the limitations of the state-of-the-art, notably by including IP fingerprinting [121] in its methodology. Chapter 4 will conclude this first part with an evaluation of this framework on a groundtruth network after including it in `TreeNET` [51]. Data collected in the wild with this framework will be used to study the relationships between profiles of fingerprints and the alias resolution methods selected by the framework. The same data will also be used to re-assess the permeability of the modern Internet to state-of-the-art alias resolution methods.

The second part will extensively discusss **subnet inference**. Chapter 5 will first review the state-of-the-art subnet inference methods elaborated up to this point, and comment on their merits and limitations. Then, Chapter 6 will characterize and quantify various issues caused by traffic engineering policies that can impair state-of-the-art subnet inference methods. Chapter 7 subsequently presents a new subnet inference tool accounting for these challenges called `WISE` (**W**ide and l**I**near **S**ubnet inferenc**E**) [52], and comprehensively describes its algorithms. Chapter 8 will conclude this second part by validating `WISE` on two groundtruth networks while comparing it with the state-of-the-art and by discussing measurements collected from various Autonomous Systems from the PlanetLab testbed [8]. By doing so, it will be demonstrated that `WISE` can usually discover the same subnets within a target network, regardless of the vantage point [52, 53]. The collected data will also be used to re-evaluate previous claims regarding the distribution of the subnet prefix length in the Internet [53].

The third and final part, on **network graph inference and modeling**, will discuss not only topology discovery methods aiming at mapping the hop-level of a target network, but also network modeling with bipartite graphs. Chapter 9 will begin this part by reviewing the state-of-the-art regarding both topics. Then, Chapter 10 will discuss in detail the interpretation of `traceroute` paths, i.e., how they can be best handled for topology mapping in the modern Internet. Chapter 11 will subsequently introduce `SAGE` (**S**ubnet **AG**gr**E**gation) [54], a new topology discovery tool that can map the hop-level of a target domain while identifying the subnets that correspond to the network links. Much like Chapter 7, this chapter will describe in detail all main algorithms. Chapter 12 will validate `SAGE` on two groundtruth networks by assessing the links it discovered and will evaluate its ability to capture comparable graphs of a same target network regardless of the vantage point, using, again, data collected from the PlanetLab testbed [8]. Chapter 13 will conclude this third part by discussing the opportunities offered by bipartite graphs for studying the modern Internet, i.e., data collected by `SAGE`, notably from the EdgeNet cluster [108], will be transformed into bipartite graphs, and the properties of these graphs will be commented on to demonstrate they are consistent with the measured topologies [54]. A tripartite model, designed to study the router-level and the presence of Layer-2 equipment, will also be briefly discussed.

Each part will come with its own research questions and conclusion, the latter also providing a short commentary on how each part fits in the holistic approach first described in Sec. 1.1.1. Finally, a general conclusion will be provided at the end of this thesis to summarize its main contributions and to discuss future research directions. Additionally, interested readers can refer to Appendix A to review the time complexity analyses of each algorithm presented throughout this document, as well as some experimental results that further justify the complexity analyses or the soundness of the algorithm themselves.

# Part I

## Alias Resolution

# Chapter 2

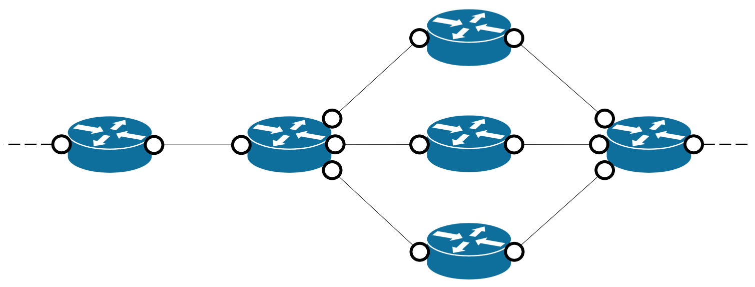
# Introduction to Alias Resolution

This chapter introduces the concepts and history of alias resolution. First of all, the main definitions and motivations of alias resolution are presented (Sec. 2.1), along with a few additional practical definitions that will be used throughout the entire thesis (Sec. 2.2). A representative sample of alias resolution methods elaborated by the research community is presented following this (Sec. 2.3). Applications of alias resolution, including scenarios for which state-of-the-art methods are not always well suited, are discussed in Sec. 2.4. Sec. 2.5 concludes this chapter by summarizing the limits of the state-of-the-art and introduces the research questions of this first part.

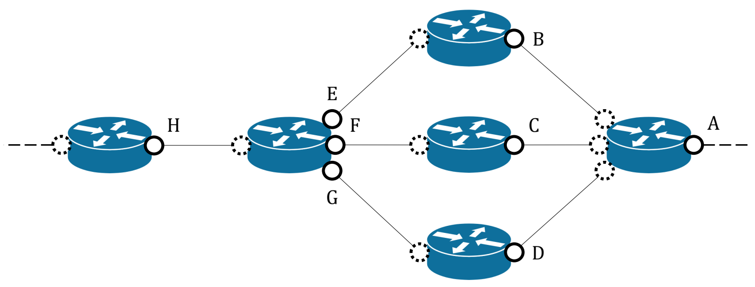
## 2.1 Main definitions and motivations

In topology discovery, an ***alias*** denotes an IP interface which belongs to the same device as another given IP interface. In other words, for two given IP interfaces $A$ and $B$, denoted in practice by their addresses, $A$ is said to be an alias of $B$ if $A$ and $B$ are located on the same device. Therefore, from the perspective of a measurement tool, two interfaces of an individual router should be aliases of each other. ***Alias resolution*** therefore denotes the process of inferring whether two IP interfaces belong to the same device or not.

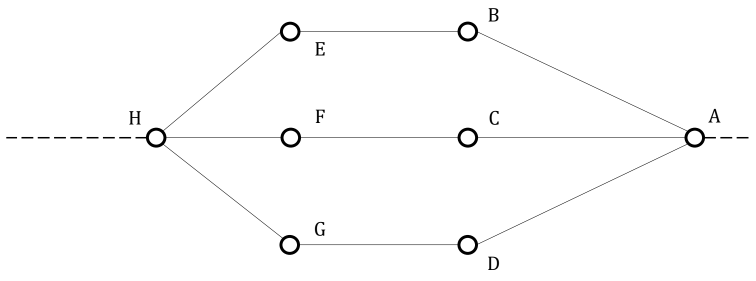
Determining whether or not a pair or a group of IP interfaces belong to the same device is crucial to study the structure of the Internet. In particular, accurately identifying the routers within a given **A**utonomous **S**ystem (AS) is essential to study the behaviour and performance of its intra-domain routing. Furthermore, accurate intra-domain data can assist network protocol designers with evaluating their new proposals on realistic topologies featuring the same characteristics as the intra-domain topologies of existing ASes. A common way to model an intra-domain network consists of building a ***router-level graph***, i.e., a graph where each vertex models a single router and where



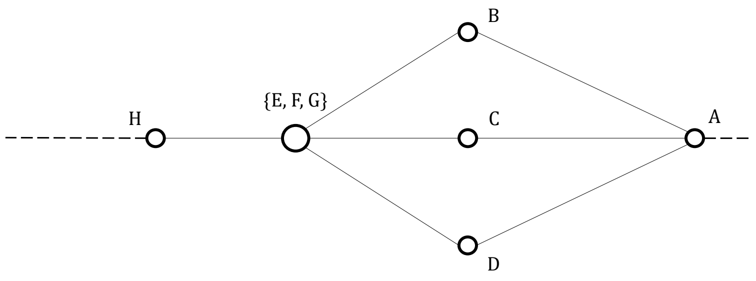
(a) Toy network (circles depict individual interfaces)



(b) Interfaces discovered by probing (plain circles)



(c) Possible router-level graph (no alias resolution)



(d) Possible router-level graph (with alias resolution)

Figure 2.1: A toy network where alias resolution helps to discover a reliable topology.

each edge models the fact that two routers are directly connected together through a common link. In the real world, links modeled in a router-level graph correspond to point-to-point links or Local Area Networks (LANs) binding two or more routers together. Routers and links are usually considered to be the building blocks of the backbone of the Internet.

A simple scenario showing the usefulness of alias resolution is given in Fig. 2.1. A toy network consisting of six routers is first depicted in Fig. 2.1a, where black circles represent interfaces. In Fig. 2.1b, the interfaces discovered through probing (from the same vantage point) are presented as plain circles, each interface being denoted by a letter. Fig. 2.1c shows the router-level graph that could be inferred from the discovered interfaces without using alias resolution first, and Fig. 2.1d illustrates what could be inferred if alias resolution is applied before creating the router-level graph. In Fig. 2.1c, two additional vertices (therefore two additional routers) $E$ and $G$ are portrayed with a total of four additional links ($B \leftrightarrow E$, $E \leftrightarrow H$, $D \leftrightarrow G$ and $G \leftrightarrow H$). In fact, these links are duplicates of two real life links ($C \leftrightarrow F$ and $F \leftrightarrow H$) and could be said to be *false* links, as they do not truly correspond to actual links shown in Fig. 2.1a. As Fig. 2.1d shows, alias resolution prevents inferring these additional routers and links by aliasing together $E$, $F$, and $G$ into one single identifier $\{E, F, G\}$, leading to an accurate router-level graph where there are as many routers and links as there are in the actual network.

To complement this toy example, it is worth noting that the importance of alias resolution has already been comprehensively studied by the research community. In particular, a study conducted by Gunes and Sarac in 2007 showed that incomplete or incorrect alias resolution can lead to drastic changes in the properties of the discovered topologies [57].

## 2.2 Additional practical definitions

The main concepts of alias resolution can be completed with additional definitions and, for practical purposes, these will be used for the rest of this thesis. When multiple IP interfaces are all found to belong to one device, the list of these interfaces is referred to as an ***alias list***, as each item in the list constitutes an alias for the other items. Likewise, two IP interfaces that are alias of each other are usually referred to as an ***alias pair***.

Prior to performing alias resolution, the IP interfaces that could hypothetically be aliases for each others can be referred to as ***alias candidates***.

In the literature, alias resolution methods are often categorized into ***active methods*** and ***passive methods***. An active method consists of inferring aliases by actively probing the alias candidates, while a passive method will rely on existing data or on data collected without directly probing the alias candidates. Such is the case, for instance, of methods based on DNS records (see Sec. 2.3.5).

Finally, an interesting property for alias resolution is ***alias transitivity***. Let us imagine a scenario where three interfaces $A$, $B$, and $C$ are alias candidates for each others. If it is possible to discover that $A$ is an alias of $B$, and if $B$ turns out to be an alias of $C$, then $A$ and $C$ can be considered to be aliases of each other as well, since it is impossible for $B$ to be located on two completely separate devices.

## 2.3 State-of-the-art alias resolution methods

Since the early 2000's, the research community has elaborated many alias resolution schemes. The subsequent sections describe some of the most successful and prominent methods.

### 2.3.1 ICMP `Port-unreachable` reply (`iffinder`)

One of the earliest active methods that was developed consisted of exploiting the `Port-unreachable` message defined by the ICMP protocol [101]. A TCP [103] or UDP [100] packet targeting an unlikely high port number is first sent towards an alias candidate. Subsequently, and due to the choice of the port number, the destination router will most likely reply with a `Port-unreachable` ICMP message. However, the source IP address of this message could differ from the address of the target interface, notably because RFC 1122 [20] recommends using the address of the interface emitting the `Port-unreachable` ICMP message as its source address. By sending a TCP or UDP probe to each alias candidate and comparing the source IP address of the replies, it becomes possible to discover alias pairs or lists. Elaborated around 2000 by K. Keys, the open source tool `iffinder` implements this approach using UDP probes [71].

While accurate, this alias resolution method is dependent on the permeability of the target network to TCP/UDP probes. Indeed, such probes can be either filtered by the target network itself, either dropped by an intermediate router between the vantage point and the target domain. Moreover, some routers can be configured not to reply with such a `Port-unreachable` ICMP message at all, as a security measure against port scanning attacks. The extent to which this method could be used nowadays is discussed in Chapter 4.

### 2.3.2 Sequence of IP identifiers (`Ally`)

In the standard for the IP protocol, the header of an IP packet includes a 16 bits-long *identification* field [102]. The content of this field is usually referred to as an *IP identifier* or *IP ID* in the literature. Originally designed to help reassemble fragmented datagrams, this field has become a major tool for alias resolution. Indeed, a common implementation for generating IP identifiers simply consists of using the lattest value of a 16 bits counter which is incremented between each sent packet. As a consequence, consecutive packets sent by the same router will feature consecutive IP identifiers. What makes this behavior interesting for alias resolution is that the counter used by a given router usually remains the same for all of its interfaces.

The `Ally` component of `Rocketfuel`, elaborated by Spring et al. around 2002, is an active alias resolution method that takes advantage of IP identifiers [113]. To test whether two IP interfaces belong to the same device, two probes [1] are first sent to both interfaces. After receiving the respective replies, containing the IP identifiers $x$ and $y$, a third probe is sent to the interface that replied first (the one which sent $x$) in order to get a third IP identifier $z$. If $x$, $y$, and $z$ forms a sequence of consecutive numbers while the delta between $x$ and $z$ is small enough, then both IP interfaces can be considered to belong to the same router.

Although it is considered to be an accurate method, `Ally` needs, for a given set of alias candidates, to test each possible pair to obtain complete alias lists. In other words, this method has a $\mathcal{O}(N^2)$ time complexity for $N$ alias candidates, discouraging usage on larger sets of alias candidates.

[1] Since the IP ID comes from the IP header, the encapsulated packet can be anything: UDP, ICMP, etc.

### 2.3.3 IP identifier velocity (`RadarGun`, `MIDAR`)

To overcome the `Ally` computational cost while increasing accuracy, the research community has explored another way to take advantage of the *identification* field of the IP header [102]. Instead of collecting identifiers for each hypothetical alias pair, several probes can be sent to a given interface in order to compute the *IP ID velocity*, i.e., the rate at which IP identifiers emitted by this interface increase. Two IP interfaces can then be aliased providing their respective sequences show close enough values and if the corresponding velocities can be used to predict identifiers of each other with a delta that is small enough.

This approach, which still falls under the active category, was first put into practice by Bender et al. with `RadarGun` in 2008 [16]. This approach was later improved by Keys et al. with `MIDAR` (**M**onotonic **ID**-Based **A**lias **R**esolution) in 2013 [73]. `MIDAR` models IP ID sequences as time series and relies on the monotonicity of each sequence to decide whether two IP interfaces are aliases of each other. Both `RadarGun` and `MIDAR` achieve linear time complexity, i.e. $\mathcal{O}(N)$ where $N$ denotes the total number of alias candidates, making both tools fit for large-scale alias resolution.

### 2.3.4 IP prespecified timestamp option (`Pythia`)

Another well known active method relies on the prespecified timestamp option of the IP protocol [102]. This option enables timestamp values from up to four IP interfaces to be queried using a single probe [2]. If the probe crosses a device which features one of the listed interfaces, a timestamp value will be recorded for this interface. As a result, and to determine whether a subset of alias candidates (up to four) belongs to the same device, a probe can be sent to one these candidates while requesting timestamps for each of them. For instance, for candidates $A$, $B$, $C$, and $D$, a probe packet will be sent to $A$ while requesting timestamp values for $A$, $B$, $C$, and $D$. If the reply packet comes with several timestamp values that are different but similar in range (or identical), then the corresponding interfaces likely share the same clock, therefore suggesting they are located on the same device.

This active method was first introduced to the research community by Sherry et al. in 2010 [110]. It was thoroughly evaluated in 2012 by de Donato et al. [31], and Marchetta et al. introduced a tool entirely based on this approach in 2013: `Pythia` [86]. Unfortunately, this method has become problematic to use due to the options of the IP protocol being reevaluated as security issues over the years. This issue was already highlighted in 2012 by de Donato et al. [31], and in particular, RFC 7126 (February 2014) recommended dropping any IP packet using the prespecified timestamp option [44].

### 2.3.5 Reverse DNS

A common passive alias resolution method consists of using reverse DNS, i.e., finding the DNS record associated with a given IP interface. Indeed, DNS records often contain suggestions on the ISP, the use and even the location of an IP interface. Therefore, similarities in DNS records for a set of alias

[2] Again, the IP datagram can encapsulate anything, since this technique relies on a feature of the IP protocol.

candidates can suggest that these interfaces belong to the same device. This method is categorized as passive since collecting DNS records does not require directly probing the alias candidates.

Reverse DNS has been used since the beginning of the 2000's. `Rocketfuel` already took advantage, in 2002, of the naming conventions of ISPs to distinguish backbone networks from PoPs (**P**oints-**o**f-**P**resences) to accurately map the router-level of their networks [113]. The topology discovery tool `AROMA` (**A**ccurate **RO**uter-level **MA**p), elaborated by Kim and Harfoush in 2007 to improve the `Rocketfuel` methodology in terms of accuracy, discovered links and probing work, includes actual alias resolution through reverse DNS among its methods [74].

However, a study by Zhang et al. showed that some IP interfaces can be incorrectly named, which can result in detecting false positives, i.e. alias pairs that do not correspond to actual routers [126]. More broadly speaking, as each ISP has its own naming conventions, reverse DNS can be difficult to tune compared to systematic methods (e.g., based on IP identifiers). However, in 2019, Luckie et al. introduced a machine learning-based approach able to ascertain regexes to match router names and therefore discover alias pairs on the basis of DNS records [80].

### 2.3.6 Analytical methods (`apar`, `kapar`, `PalmTree`, `APPLE`)

While many alias resolution methods rely on active probing of the alias candidates, several other techniques (aside those using reverse DNS, cf. Sec. 2.3.5) have explored the possibility of discovering alias pairs by carefully analyzing the network topology rather than probing the alias candidates. Elaborated by Gunes and Sarac around 2009, `apar` (**a**nalytic and **p**robe-based **a**lias **r**esolver) uses route paths discovered via `traceroute` to analytically infer alias pairs [60]. This tool was later optimized by K. Keys in 2010 and renamed `kapar` in the process [72]. Both tools could be considered as passive as they rely on preexisting data.

In 2011, Tozal and Sarac introduced `PalmTree`, an alias resolution tool which takes advantage of standard IP assignment practice to minimize the amount of probes, achieving a linear time complexity (i.e., $\mathcal{O}(N)$ with $N$ alias candidates) [117]. Finally, `APPLE` (**A**lias **P**runing by **P**ath **L**ength **E**stimation), which was introduced in 2020 by A. Marder, relies on the return path lengths of the probes and on multiple vantage points to filter potential alias pairs [87].

An interesting common denominator of `PalmTree` and `APPLE` is that all their respective authors share the opinion that their analytical features should be used mostly to complement active methods rather than to replace them. The benefits of such an approach were, in practice, demonstrated with `APPLE` [87], as it combines its alias pruning method with `iffinder` and `MIDAR` to outperform both.

## 2.4 Applications of alias resolution

As already highlighted in Sec. 2.3, many works in alias resolution puts as much emphasis on efficiency as they do on accuracy. More broadly speaking, many state-of-the-art methods either aimed at decreasing the total number of probes required for accurate alias resolution, either achieved linear time complexity with respect to the total number of alias candidates. On the one hand, AROMA was designed to drastically reduce the overall probing work required for topology mapping in respect of Rocketfuel, including when it comes to alias resolution [74]. On the other hand, the alias resolution tools RadarGun [16], PalmTree [117], and MIDAR [73] were all designed to have a linear time complexity to outperform the Ally method [113], which suffers from a quadratic time complexity. It should not come as a surprise that there is such a focus on efficiency, as topology discovery tools often operate on large networks or large portions of the Internet, making large-scale deployment of methods such as Ally difficult to envision.

However, alias resolution can also be very useful to shine light on simpler scenarios that will be explored throughout this entire thesis. This section will give two practical examples where alias resolution can ease the discovery of topological data on a very localized part of a target network.

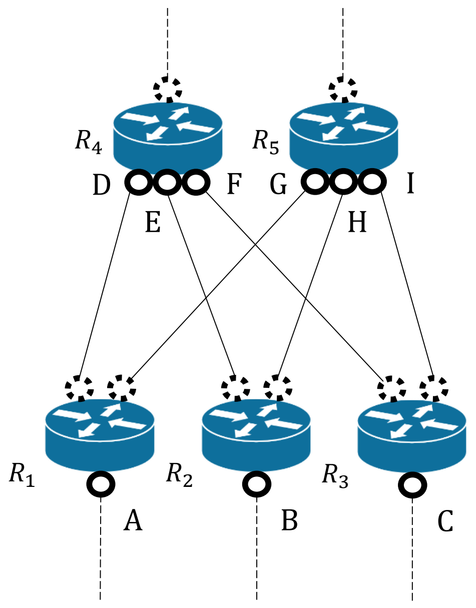


Figure 2.2: A toy load balancing architecture.

A first practical application of alias resolution would be the detection of ***convergence points*** while probing the router interfaces of a target network. In order to prevent network congestion where there is an increase in traffic, network providers typically use architectures featuring multiple paths to balance the load between multiple links, an approach which is traditionally referred to as ***load balancing***. Such an approach notably helps network operators to deal with increasing network traffic without increasing link capacity. Research by Augustin et al. aimed at enumerating as many paths as possible to quantify load balancing in the wild [11, 14] and highlighted the existence of convergence points in the Internet, i.e., routers where load balanced paths converge. Multiple interfaces of these routers can be revealed by the probing as a result of load balancing. For instance, a router where three load balanced paths converge might reply with a different interface for each of these paths, just like in the toy topology shown in Fig. 2.1. As a result, failing to identify these convergence points while analyzing paths can lead to incorrect conclusions on the target network. A toy network that depicts a mesh of routers corresponding to a simplified load balancing architecture is shown in Fig. 2.2. In this figure, plain black circles model the interfaces revealed by the probing. The set of interfaces $D$, $E$ and $F$ as well as the set $G$, $H$ and $I$ belong respectively to routers $R_4$ and $R_5$. Without alias resolution, each of these convergence points would be modeled by no less than three routers. Correctly identifying $\{D, E, F\}$ and $\{G, H, I\}$ is therefore crucial to accurately map the provided example network.

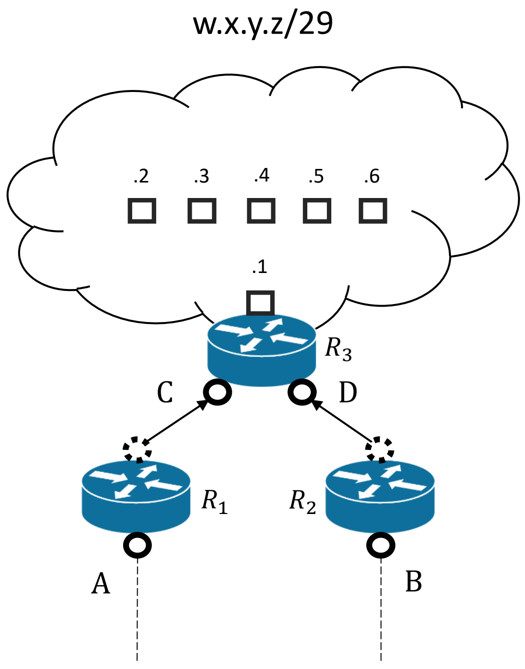


Figure 2.3: A toy subnet inference scenario.

Fig 2.3 shows another toy network where the top router ($R_3$) acts as a gateway to a subnet, i.e., a set of IP interfaces which can reach each other directly through a LAN. This subnet is depicted by a large cloud, while the squares and circles symbolize subnet interfaces and router interfaces discovered via network probing, respectively. Because of the multiple paths leading to router $R_3$, both of its interfaces $C$ and $D$ could appear in the collected data as the final interfaces crossed before reaching the subnet interfaces. For a tool to try to discover the top subnet `w.x.y.z/29` would require identification of the gateway router, which would mean that it should be able to determine whether $C$ and $D$ belong to the same device. Localized alias resolution can solve this issue.

In both previous scenarios, the sets of alias candidates can be implied from the collected data and are fairly small. For such simple use cases of alias resolution, deploying solutions designed for large-scale measurements would induce unnecessary large amounts of probes. In particular, in the case of `MIDAR`, dozens of probes are sent to each alias candidate in order to compute a time series for each candidate. In comparison, replicating the `Ally` method only needs three probes to test each alias pair. Therefore, deploying large-scale solutions for such scenarios not only becomes questionable but also means that a new approach which is better suited to small sets of alias candidates is required.

## 2.5 Closing comments: towards a revisited alias resolution

As discussed in Sec. 2.3, several state-of-the-art alias resolution methods are difficult to use either on a wide scale, or at all. For instance, methods based on the prespecified timestamp option of the IP protocol are now virtually impossible to use as a consequence of the IP packets with options being broadly filtered (cf. Sec 2.3.4), and the approach implemented by `iffinder` (cf. Sec. 2.3.1) can be impaired by modern security policies (e.g., preventing port scanning attacks). As already recommended by the authors of analytical tools (cf. Sec. 2.3.6), one way to mitigate the limitations inherent to each alias resolution method would be to combine them.

Not only combining several approaches should be considered, but a modern alias resolution scheme should also be designed keeping efficiency and minimal probing costs in mind. While several state-of-the-art solutions are adapted for Internet-scale alias resolution (such as `MIDAR`, cf. Sec. 2.3.3), they are too costly in terms of probes to solve simpler scenarios of alias resolution such as those mentioned in Sec. 2.4. It should therefore be possible to analyze both large and small alias resolution scenarios in linear time with as few probes as possible.

This first part of the thesis will address both aforementioned challenges by answering the following research questions, with each subsequent chapter answering specific questions.

1. **Can a methodology be designed to discover which alias resolution methods can be used on a given set of alias candidates ? How ?** Before even trying an alias resolution method, the alias candidates should be analyzed to determine which methods are exploitable and which one is the most likely to provide accurate results. These questions will be answered with a novel alias resolution framework presented in Chapter 3.

2. **How can the probing cost of this new methodology be reduced ?** Minimizing the number of probes sent per alias candidate is critical to achieve efficient alias resolution and to avoid triggering security mechanisms. How this can be achieved will also be covered in Chapter 3.

3. **How and when should the new methodology be used ? Is it accurate ?** The best way(s) to use the new methodology will also be discussed. The framework will also be assessed with the help of a groundtruth network. Both these questions will be answered in Chapter 4.

4. **To which extent can state-of-the-art methods be used in today's Internet ?** To answer this question, a study will be conducted by deploying the alias resolution scheme designed to answer questions 1 and 2 in the wild, i.e., from a measurement testbed. This study will be included in Chapter 4.

# CHAPTER 3

# A FINGERPRINT-BASED ALIAS RESOLUTION FRAMEWORK

This chapter introduces an alias resolution framework designed to address the two challenges discussed in Sec. 2.5. This framework consists of fingerprinting alias candidates in order to select the most suitable state-of-the-art alias resolution method, and therefore constitutes an answer to research question 1 from Sec. 2.5. Sec. 3.1 first introduces IP fingerprinting by reviewing previous work and presents the components of fingerprints for alias resolution. Sec. 3.2 describes a probing scheme designed to collect the necessary data for fingerprints with as few probes as possible, therefore answering research question 2 from Sec. 2.5. Sec. 3.3 presents how alias resolution can be carried out after the fingerprinting and without additional probing. Finally, Sec. 3.4 concludes this chapter by summarizing the advantages and perspectives of the new framework.

## 3.1 IP fingerprinting

### 3.1.1 Previous work

The idea of fingerprinting IP interfaces (or addresses) was first explored by Vanaubel et al. in an attempt to recognize different router brands through probing [121]. For this purpose, they studied the typical use of the *Time To Live* (or TTL), an 8-bit field found in the header of IP datagrams [102].

The IP protocol TTL field consists of an 8-bit unsigned integer value that is decremented each time an IP packet crosses a router. Once a packet reaches a TTL value of 1, the router receiving the packet drops it and then usually replies to the sender with an ICMP message featuring the `time-exceeded` code [101]. Such a mechanism prevents a packet from being forwarded indefinitely if it ends up cycling in a forwarding loop. In topology discovery, the TTL field is typically used to reveal router interfaces on the way to some destination IP address, as each `time-exceeded` reply will provide, as a source IP, one of the interfaces of the router that generated the reply. The classical tool `traceroute` [119] sends a sequence of UDP probes towards a target IP address, starting with a TTL value of 1 and incrementing the TTL after each new probe until it obtains a reply from the destination IP address, therefore revealing router interfaces on the way to the destination.

Though RFC 1700 recommends using 64 as the default initial value for the TTL field of any IP packet [104], Vanaubel et al. discovered that this convention is not strictly followed by equipment vendors. Indeed, the initial TTL values 32, 128, and 255 are also used, with 255 being the most common, and the initial TTL value of a packet also varying in some cases depending on its nature. For instance, at the time of their research (2013), Juniper routers running with the *Junos* OS typically used 255 as the initial TTL value of `time-exceeded` replies but 64 as the initial TTL value of ICMP `echo-reply` messages, i.e., replies to `echo-request` used to verify an IP interface is online [101].

To take advantage of this observation, ***fingerprints*** (or ***router signatures***) were conceptualized as $n$-tuples of values characterizing the behaviour of a router. The aforementioned research used an even more specific definition: *fingerprints* are $n$-tuples of initial TTL values used by $n$ different types of packets. The notation $<i_1, \; i_2, \; \ldots \; i_n>$ was introduced to denote a fingerprint of $n$ values. Consequently, the typical signature of Juniper routers running with *Junos* is `<255,64>`.

In addition to providing an insight on the vendor of a router, fingerprinting is easy to implement, as it requires a minimal number of probes to compute the values for a given IP interface once the interface has been detected. Vanaubel et al. used Paris `traceroute` [11] to discover router interfaces and collect the final TTL values of the corresponding `time-exceeded` replies to assume the associated initial TTL value. Several ICMP `echo-request` messages were subsequently sent (6 in total) to each router interface to infer the second initial TTL value of the fingerprints. This inference is easy to achieve in practice: as the vast majority of routes are less than 30 hops (i.e., more than 99%) [121], the initial value can simply consists of the smallest value among 32, 64, 128, and 255 which is just above the final TTL values of the `time-exceeded` and `echo-reply` messages. Using several `echo-request` messages guarantees that the inferred initial value is consistent across multiple probes. When the inferred initial value is inconsistent, it is deemed as unknown, and replaced in a signature by `*`. For instance, the signature of an interface which the initial TTL value for `time-exceeded` messages is 64 while the initial TTL value for `echo-reply` is unknown will be `<64,*>`.

Using more than 10 millions of paths discovered via Paris `traceroute` by probing around one million of target IP addresses, Vanaubel et al. studied the signatures distribution [121]. In particular, they showed that the signature `<255,255>`, accounting for Cisco routers, represented more than 50% of all signatures, while `<255,*>`, `<255,64>` and `<64,64>` each accounted for slightly more than 10%. This trend still holds true today: among router signatures collected by Maréchal et al. in 2019, almost 60% matched Cisco devices [88]. Though fingerprinting was later used by the same research group to detect particular MPLS tunnels [120, 121], it also has potential for alias resolution. Since IP fingerprints can be linked to equipment vendors, identical fingerprints can suggest alias pairs and also what methods can be used. For example, if the behaviour of the IP ID counter tied to an interface can be fingerprinted, a fingerprint can tell whether methods based on the IP identification field (cf. Sec. 2.3.2 and Sec. 2.3.3) are viable or not for a given set of alias candidates.

### 3.1.2 Fingerprints for alias resolution

As already established in Sec. 2.3, the research community has provided many different efficient alias resolution methods, but these can be difficult to apply to all router interfaces in the Internet. The classical tool `iffinder` (cf. Sec. 2.3.1) is a good example: despite being a very clear way to alias two interfaces with each other, the various security policies regarding `Port-unreachable` ICMP messages applied across the entire Internet drastically hampers it. Nowadays, it is recommended that the `iffinder` approach should only be used to complement other alias resolution methods, another approach that has been already put forward by other researchers (cf. Sec. 2.3.6).

IP fingerprinting can prove useful for alias resolution in two ways. First, fingerprints may be used to reduce the space search even before deciding which method to use. Indeed, as fingerprints will be derived from the behaviour of router interfaces, two interfaces exhibiting completely different behaviour patterns cannot belong to the same device. For instance, it is very unlikely that the source interface of an `echo-reply` ICMP message using the initial TTL value 64 will be found on the same router as another interface that uses the initial value 255 for the same type of message. Second, the characteristics of a fingerprint can suggest the most suitable alias resolution method for a set of alias candidates. For example, when fingerprints advertise `Port-unreachable` ICMP messages could be obtained, using the `iffinder` method is possible and should even be prioritized over other methods because of how `Port-unreachable` ICMP messages directly suggest alias pairs.

In the best way to model the behaviour and compatibility with various alias resolution methods of a router interface, a fingerprint can consist of the following components:

- the **initial TTL value of the `echo-reply` ICMP message**,
- the **source IP of the `Port-unreachable` ICMP message** (if any),
- the ***IP ID counter class***,
- the **existence of a DNS record**,
- the **compliance to the ICMP timestamp request** [101].

The first component simply re-uses one of the two initial TTL values of the fingerprints introduced by Vanaubel et al. [121]. The second component accounts for the possibility of getting a `Port-unreachable` ICMP message from the interface. When receiving such a message is possible, this component will appear in the fingerprint as the source IP found in the message. Otherwise, it will be replaced with `*` to demonstrate that such a reply cannot be obtained from the vantage point in use, therefore preventing the use of the `iffinder` method.

The ***IP ID counter class*** is a unique property specifically designed for IP fingerprinting. It consists of a label that describes the observed behaviour of the IP ID counter tied to the interface. Given a sequence of $k$ IP identifiers (or IP IDs) collected in $k$ packets emitted by the router interface, the counter class will consist of one of the five following labels.

- The ***healthy*** class will be attributed if the $k$ IP IDs sequence is steadily increasing and features no more than one rollover, i.e., reaching the maximum value of the IP identification field and starting again at 0. The limit of one rollover makes it possible to take account of sequences encompassing a counter overlap. Such a label means all alias resolution methods based on IP IDs are viable.

- The ***fast*** class is attributed if the $k$ IP IDs sequence does not consist of strictly increasing values as a result of multiple rollovers, but such that it remains possible to estimate the velocity (or *IP ID rate*) of the counter, e.g., as a range of values. It accounts for routers that process large amounts of traffic and for which only velocity-based methods are viable.

- The ***random*** class refers to an IP ID counter that generates an IP IDs sequence for which evaluating the velocity is impossible or requires modeling too many rollovers, suggesting that IP IDs are randomly generated.

- The ***echo*** class denotes the case of an IP IDs sequence that simply repeats the IDs contained in the IP datagrams sent to the interface in order to collect said sequence. E.g., if the values 2413, 2421, 2434, and 2455 ($k = 4$) are used in the probes sent to an interface labelled with the *echo* class, the sequence obtained will also be 2413, 2421, 2434, 2455.

- Finally, if an IP IDs sequence could not be collected (due to connectivity issues, for instance), the class is replaced with `*`.

The last two components reply with values of either "*Yes*" or "*No*" depending on whether the interface has a DNS record and whether it complies with the ICMP timestamp request mechanism, respectively. The timestamp request mechanism is an additional feature of the ICMP protocol [101] created for devices to synchronize their clock. The inclusion of this mechanism in the fingerprints is purely exploratory, as it does not disclose whether a specific alias resolution method is viable or not. The relevance of this component will be commented on in Chapter 4 (Sec. 4.3.4).

An example of a fingerprint using the proposed components would be `<64,*,Healthy,Yes,Yes>`. An interface with such a fingerprint uses 64 as the initial TTL in an `echo-reply` ICMP message, and does not or cannot send a `Port-unreachable` ICMP message. It is fit for alias resolution methods based on IP IDs or based on the DNS record, and replies to ICMP timestamp requests.

Listing 3.1: Example of fingerprints (with associated IP addresses) for alias resolution

```
10.0.1.1 - <255,10.0.1.1,Echo,Yes,No>
10.0.2.1 - <255,10.0.1.1,Echo,Yes,No>
10.0.2.65 - <64,*,Healthy,No,Yes>
10.0.2.129 - <64,*,Healthy,No,Yes>
10.0.2.193 - <64,*,Healthy,No,Yes>
10.0.2.225 - <64,*,Healthy,Yes,Yes>
```

By sorting the fingerprints collected for a given set of alias candidates and putting them side by side, it quickly becomes clear which candidates could result in alias pairs/lists and which methods should be used to test an alias pair. Example 3.1 shows a toy scenario where six alias candidates are fingerprinted prior to performing alias resolution. After fingerprinting, the candidates are sorted according to their fingerprints. In this example, `10.0.1.1` and `10.0.2.1` can only be aliased with each other, as their fingerprints show these IP interfaces uses a different initial TTL value for `echo-reply` ICMP packets than other candidates. Moreover, the source IP of their respective `Port-unreachable` ICMP reply is `10.0.1.1` in both cases, therefore strongly suggesting an alias pair. Likewise, the four remaining candidates (`10.0.2.65`, `10.0.2.129`, `10.0.2.193`, `10.0.2.225`) might result in an alias list, as they have similar fingerprints suggesting that a method based on IP IDs is applicable to elicit whether these candidates belong to the same router or not.

It should be noted that fingerprints do not necessarily need to be strictly identical. For instance, it is possible that only a few specific interfaces of a router possess a DNS record. This is also demonstrated in Example 3.1: among the alias candidates with the *healthy* IP ID counter class, `10.0.2.225` is the only one to feature a DNS record. Nevertheless, it can still be aliased with other IP interfaces with the same IP ID counter class as long as an ID-based resolution method associates them. In fact, only two components of a fingerprint can be used to exclude an alias pair: the (inferred) initial TTL value of the `echo-reply` ICMP messages and the IP ID counter class. With extended fingerprints, presented in the next section (Sec. 3.1.3), a third component that can exclude alias pairs can be used.

### 3.1.3 Extended fingerprints

The fingerprints introduced in Sec. 3.1.2 can be used on any IP interface, regardless of how it was detected. Indeed, the fingerpriting process itself, as will be detailed in Sec. 3.2, requires very few direct probes (typically ICMP and UDP probes) with the addition of one indirect probe (collecting the DNS record, if any). However, in order to include the (inferred) initial TTL value of `time-exceeded` ICMP messages (introduced in Sec. 3.1.1), such a message must first be obtained from the IP interface. In previous work, these messages were easily obtained due to the discovery of router interfaces via Paris `traceroute`. [11, 121] In other words, if the alias candidates were discovered through means other than `traceroute`-like probing (e.g., by scanning IP prefixes with ICMP probes featuring large TTL values), it might not be possible to obtain such messages, as it would require targeting an interface found one hop further such that said candidates can reply with `time-exceeded` ICMP messages.

Hopefully, when the alias candidates are first discovered via `traceroute`-like probing, getting `time-exceeded` ICMP messages from them becomes trivial, and the initial TTL value of such messages can be added to fingerprints. In order to stay true to the notation introduced by Vanaubel et al. (see Sec. 3.1.1), this initial TTL becomes the new first component of a fingerprint. For instance, if the router interface that corresponds to the example fingerprint `<64,*,Healthy,Yes,Yes>` presented in Sec. 3.1.2 is first discovered via Paris `traceroute` [11], it can also be observed that it uses the initial TTL value 255 for `time-exceeded` ICMP messages. As a consequence, the extended fingerprint for this interface will be `<255,64,*,Healthy,Yes,Yes>`. Extended fingerprints allow the exclusion of additional alias pairs while reinforcing the idea that IP interfaces with similar fingerprints are from the same device thanks to the additional component.

## 3.2 A probing scheme to collect fingerprints

This section describes in detail a probing scheme to both fingerprint alias candidates and perform alias resolution on them. Sec. 3.2.1 first describes the typical uses of the framework. Secondly, Sec. 3.2.2 presents the design of the probing scheme. Finally, Sec. 3.2.3 provides the practical algorithms. The time complexity of these algorithms can be found in Appendix A (Sec. A.1).

### 3.2.1 Use cases

On paper, fingerprinting can be easily used on both a small and a large scale. In fact, it can be used on specific small networks as well as on the scale of an entire autonomous system (or AS) or more. Fingerprinting only requires a small constant number of probes on each alias candidate, as the number of probes for a specific candidate is completely independent from which other alias candidates it could be eventually matched with.

However, fingerprinting itself is not alias resolution and is only meant to prepare the actual alias resolution by identifying subsets of alias candidates that are likely aliases of each others and selecting the most suitable method to actually resolve them. Subsequent sections of this chapter will nevertheless present a probing scheme that was specifically designed to allow simulatenous fingerprinting and alias resolution. I.e., the data collected for the fingerprinting itself can be used for the subsequent alias resolution to spare the need for sending additional probes. This feature is especially important to reduce the probing overhead of active alias resolution methods, particularly those based on the IP identification field of the IP protocol (cf. Sec. 2.3.2 and Sec. 2.3.3).

As a consequence, the probing scheme presented in subsequent sections will work best with small sets of alias candidates, i.e., scaled to a single alias resolution scenario rather than an entire network. If it was applied, for instance, to the entire Internet, the provided probing scheme would only be useful to produce the final fingerprints, as the collected data would be insufficient for actual alias resolution. Fingerprinting would therefore only be useful to reduce the number of alias candidates processed by Internet-scale solutions such as `MIDAR` [73] (see also Sec. 2.3.3).

The methodology introduced in this chapter should therefore be either applied on small-scale scenarios comparable to those in Sec. 2.4, or combined with ***space search reduction*** to apply it on the scale of whole networks. In the context of alias resolution, space search reduction consists of breaking down the initial set of alias candidates in many smaller sets by rejecting alias pairs that are deemed impossible. For instance, trying to alias two IP interfaces located several hops away from each other is difficult to justify. A method for space search reduction will be introduced in Chapter 4 in order to evaluate the entire alias resolution framework, but other practical methods for space search reduction will be presented throughout this entire thesis.

### 3.2.2 Designing and optimizing the probing scheme

Fingerprinting a single IP interface can be achieved in a few probes. Among the components presented in Sec. 3.1.2, the IP ID counter class represents the largest investment in terms of probes: indeed, it requires the collection of $k$ IP identifiers (or IP IDs) in order to attribute a class and later compute a range of values for the velocity (or *IP ID rate*). The value of $k$ can be tuned at will, but one should use at least $k = 4$ should be used in order to obtain at least three time intervals, i.e., given four IP IDs $a$, $b$, $c$, $d$ (collected in this order), there will be one interval for each pair among $(a, b)$, $(b, c)$ and $(c, d)$. In order to collect $k$ IP IDs, $k$ probes are required. These $k$ probes will be useful for collecting $k$ IP IDs, but they can be also useful to consistently infer the initial TTL value of the `echo-reply` ICMP packets. I.e., if the initial TTL value inferred for each reply packet [1] is the same at each probe, this initial TTL value will be the final value of the fingerprint, otherwise this value will be replaced by * (such an approach has already been mentioned in Sec. 3.1.1). Therefore, computing the IP ID counter class and the initial TTL value of `echo-reply` ICMP messages can be achieved with $k$ `echo-request` ICMP probes.

Given that collecting the three other components of a fingerprint requires a single probe for each, fingerprinting can be achieved with a total of 6 direct probes on the IP interface when using $k = 4$:

- $k = 4$ ICMP `echo-request` probes (IP ID counter class and initial TTL value of `echo-reply` ICMP messages),
- one UDP probe to obtain a `Port-unreachable` ICMP message (and the source IP of this message),
- one indirect probe for the reverse DNS look-up (existence of a DNS record),
- one additional ICMP probe with a timestamp request (responsitivity to such requests).

While being inexpensive in terms of probes (tools such as `MIDAR` can send dozens of probes to one IP address, see Sec. 2.3.3), fingerprinting is on paper not sufficient in itself for performing alias resolution, as it only reveals which IP interfaces could be aliases of each others and what method should be used. In practice, subsequent probing work is not required to use the `iffinder` method (the interesting IP interfaces are already given in the fingerprints) or the reverse DNS approach (since it is passive by design, cf. Sec. 2.3.5). Only methods based on the IP ID field (cf Sec. 2.3.2 and 2.3.3) would require additional probing after collecting the fingerprints, but carefully scheduling the probes used to pick the IP ID counter class can spare this additional effort.

An IP address and its IP IDs sequence can be expressed by the format "`IP - a,b,c,d`" where $k = 4$ and where $a$, $b$, $c$, $d$ are the collected IP IDs (in this order). Four IP interfaces featuring a *healthy* IP ID counter can be probed in sequence. If these interfaces belong to the same router, sequences comparable to those of Example 3.2 should be obtained. In this example, the IDs clearly form a strictly increasing sequence when reading the sequences consequentially (first the sequence for

[1] For reminders, it consists in picking the smallest value among 32, 64, 128 and 255 greater than the final TTL value.

`10.0.2.65`, then for `10.0.2.129`, etc.), with the IDs being given in the same order as the probes are used to collect them. However, such sequences are only useful for inferring the IP ID counter class or using a velocity-based approach, as long as timestamps are recorded on reception of each ID to later compute the rate at which IDs increase (for the sake of readability, such timestamps are omitted from Example 3.2). Where the same ideas as the `Ally` component from `Rocketfuel` [113] (see Sec. 2.3.2) need to be resued with the collected IDs, it must be possible to interleave IDs from separate sequences to see if the resulting sequence is still strictly increasing.

Listing 3.2: Sequences of IP IDs (collected one at a time)

```
10.0.2.65 − 45,48,51,55
10.0.2.129 − 56,58,59,62
10.0.2.193 − 65,67,69,72
10.0.2.225 − 73,74,78,81
```

One way to be able to interleave the collected IDs is by concurrently collecting the sequences of IDs. Not only will this increase the probing rate, i.e., the number of probes sent for a selected unit of time (per second, per minute, etc.), but this should also randomly create an interlaced order among the probes. To keep track of the order in which the probes were sent, each collected IP ID can be attributed a ***token***, i.e., a unique number which is emitted by a shared counter on sending a probe that is incremented after each request. [2] If a token is denoted by $t_i$, a sequence of token/ID pairs collected for a single IP can be expressed as "`IP -` $t_a ; a , t_b ; b , t_c ; c , t_d ; d$" where $k$ is still equal to 4 and where $t_a ; a$ denotes the first ***token/ID pair*** (again, the timestamps that would be useful to evaluate counter velocity are omitted for readability). An interleaving of two sequences is therefore possible if interleaving the tokens of both sequences results in a strictly increasing sequence.

Listing 3.3: Sequences of IP IDs with tokens (collected concurrently)

```
10.0.2.65 − 1;45,4;55,9;65,11;69
10.0.2.129 − 2;48,7;59,10;67,13;73
10.0.2.193 − 5;56,6;58,14;74,15;78
10.0.2.225 − 3;51,8;62,12;72,16;81
```

Example 3.3 shows sequences of token/ID pairs collected concurrently, with tokens ranging from 1 to 16. In this toy example, the order of the pairs is semi-interlaced: as a consequence, there are opportunities for interleaving sequences though this is not strictly possible for any pair of IP addresses. In the example, the third IP (`10.0.2.193`) only sent the probe to collect its first ID after the first IP of the set (`10.0.2.65`) sent a probe to get its second ID. While the sequences presented could be said to be sufficient to obtain alias pairs (for instance, it would be easy to interleave the sequences for `10.0.2.129` and `10.0.2.193`), random network delays and random thread scheduling at the

[2] Implementing such a counter in practice requires, of course, a mutual exclusion mechanism. The delay induced by such a mechanism should be negligible.

vantage point could result, with larger sets of alias candidates, in one sequence starting after another one has ended, preventing any chance of interleaving sequences for given pairs of alias candidates.

Therefore, to ensure that an interleaving of the token/ID pairs is always possible, the alias candidates can be probed in ***waves***. One ICMP probe is first sent to each of the target IP interfaces within a short timespan to ensure that the subsequent replies are received relatively quickly as well. As a consequence, the collected IP IDs should similar in value if the interfaces that sent them belong to the same router and use the same IP ID counter. Ideally, the probes of a wave should be sent in a pre-defined order with a short idle period (in milliseconds) between each, so that the replies coming from one device arrive in the same order as the probes were sent. The order in which the probes are sent is however up to the implementation: it can be randomized, static (i.e., always the same) or updated after each wave depending on the time taken to receive the replies. After (and only after) receiving a first IP ID from each alias candidate, a second wave of probes can be scheduled. The third wave is scheduled after all replies from probes from the second wave have been received, and so on for each wave, for a total of $k$ waves. As a result, the sequences of token/ID pairs can all be interleaved, so that it can be determined whether interleaving specific sequences of token/ID pairs results in strictly increasing tokens and IP IDs (with the exception of counter overlaps).

Listing 3.4: Sequences of IP IDs with tokens (collected in waves)

```
10.0.2.65 − 1;45,6;58,9;65,15;78
10.0.2.129 − 2;48,7;59,12;72,16;81
10.0.2.193 − 3;51,8;62,11;69,14;74
10.0.2.225 − 4;55,5;56,10;67,13;73
```

Example 3.4 shows the same toy scenario as in Examples 3.2 and 3.3 but using *waves* of probes to ensure the sequences of token/ID pairs can be interleaved at least in respect of the tokens. As a consequence, the first token of any sequence will be smaller than the second token, which will, in turn, be smaller than the third token, and so on. If the associated IP IDs also form an increasing sequence, then it can be assumed that the corresponding alias candidates belong to the same device.

Listing 3.5: Sequences of IP IDs with inconsistent tokens (waves scheduling)

```
10.0.2.65 − 1;45,7;58,9;65,15;78
10.0.2.129 − 2;48,6;59,11;72,16;81
10.0.2.193 − 3;51,8;62,12;69,14;74
10.0.2.225 − 4;55,5;56,10;67,13;73
```

Though scheduling probes in waves guaranteeing an interleaving of tokens is possible in theory, such a probing scheme can, in practice, suffer from network delays. Even if the scheduling includes small delays between probes, so replies from a same device come in the same order as the probes that were sent to it, random network delays can result in the probes not reaching their targets in the expected order. As a consequence, token/ID pairs can conflict with each other and prevent a consistent alias resolution. Example 3.5 shows another toy scenario where the IP IDs emitted by one device are not received in the same order as the initial probes due to network delays. In particular, tokens 6 and 7 in the second wave and tokens 11 and 12 in the third wave conflict with their IP IDs despite source interfaces belonging to the same router. As a result, it would be impossible to produce a single alias list with all four candidates.

This issue can be mitigated with a simple correction heuristic: after receiving all replies from a given wave, the corresponding token/ID pairs are sorted in ascending order in respect of the IP IDs. The sorted pairs are then reviewed, and if two consecutive pairs are found to have IP IDs with similar values but have tokens in the reverse order, then the tokens are rearranged to ensure the final tokens are ascending in the same way as the IP IDs. This is done in practice by moving backwards (in the sorted pairs) the token of a pair that appears to be in the wrong place.

### 3.2.3 Practical algorithms

The key to implementing the probing scheme of Sec. 3.2.2 in an elegant fashion is to carefully design the data structures that will store all the details tied to each probe and how they will be handled as the probing takes place. In particular, the data of a single ICMP probe should be collected as an ***IP ID tuple***, i.e., the gathering of

- the probed alias candidate,
- the token emitted prior to sending the `echo-request` message,
- the IP ID contained in the `echo-reply` message,
- the final TTL value contained in the `echo-reply` message,
- a timestamp value recorded at the reception of the `echo-reply` message.

The $k$ tuples collected for a specific alias candidate can be stored progressively in a data structure that abstracts them, e.g., in an object-oriented fashion. Before that, the tuples produced by each wave of probes as presented in Sec. 3.2.2 should be treated with the correction heuristic before storing them within the data tied to each alias candidate. After the $k$ waves, the complete list of tuples of each alias candidate can be post-processed to infer the initial TTL value of `echo-reply` messages (using the final TTL values of each of the $k$ `echo-reply` messages) and pick an IP ID counter class.

Algorithm 1 summarizes the probing strategy for fingerprinting, given a list of alias candidates $T$, the parameter $k$ previously introduced in Sec. 3.1.2 and the previously proposed data structures. Most of this algorithm formalizes how the IP IDs are collected for all alias candidates with the probing scheme elaborated at Sec. 3.2.2, therefore ensuring that the data is sufficient for both fingerprinting and state-of-the-art alias resolution methods based on IP IDs (lines 2–11). For simplicity's sake, the alias candidates are considered to be treated in the algorithm in an object-oriented manner, allowing fingerprint data to be stored progressively using a number of different methods, as illustrated by the `processIPIDs()` method at line 11 which reviews the previously collected IP ID tuples. Such

**Algorithm 1** Fingerprinting a set of alias candidates

```
Require: $T$, list of the target alias candidates
Require: $k$, the total of IP IDs collected per candidate
 1: procedure FINGERPRINT(List<IPEntry> $T$, Integer $k$)
 2:     i ← 0
 3:     for i < $k$ do
 4:         newTuples ← PROBEINWAVE($T$)
 5:         TOKENORDERHEURISTIC(newTuples)
 6:         for tuple ∈ newTuples do
 7:             candidate ← tuple.getTarget()
 8:             candidate.pushIPIDTuple(tuple)
 9:         i++
10:     for candidate ∈ $T$ do
11:         candidate.processIPIDs()
12:     PROBEFORPORTUNREACHABLE($T$)
13:     REVERSEDNSRESOLUTION($T$)
14:     PROBEWITHTSREQUEST($T$)
```

a method can compute, for instance, the initial TTL value of `echo-reply` ICMP messages, but can postpone the selection of the IP ID counter class to a preliminary step of the actual alias resolution (see Sec. 3.3). By doing so, the process of collecting the IP ID data can be completely separated from how it is interpreted for alias resolution. The moment when the IP ID counter class is selected depends on the implementation.

Methods for storing fingerprint data are also specifically touched on in the procedures mentioned at lines 12–14. Each of these procedures is meant to collect a single fingerprint component for each alias candidate found in $T$, using at worst one single direct probe for each. For instance, the procedure at line 12 replaces the process of sending a UDP probe with an unlikely high port number to each alias candidate and recording the source IP of the `Port-unreachable` ICMP reply (if any). By the end of the `fingerprint()` procedure, the objects initially listed in $T$ will have recorded all the fingerprint data as well as sequences of IP IDs for suitable alias resolution methods. In the case of an extended fingerprint (see Sec. 3.1.3), the procedure remains unchanged, as the initial TTL value of `time-exceeded` messages can be inferred from replies obtained at the time of discovering the alias candidates, i.e., prior to calling the `fingerprint()` procedure. The technical details of the probing, such as how the probes are sent in practice (e.g., with multithreading), are left to the implementation.

Algorithm 2 details the correction heuristic used to fix any potential out-of-order token/ID pairs. The general approach is simple: after sorting the token/ID pairs following the ascending order of the IDs (cf. line 2), a first loop reviews the pairs to identify successions of pairs where the tokens are decreasing instead of increasing along with the IDs (lines 5–29). If a mismatch is found, the algorithm proceeds with a second nested loop that will go through the sorted list in reverse order to move the problematic token until it matches the sequence of IDs (lines 17–27). This nested loop stops as soon as the tokens are increasing again, and the algorithm resumes the main loop right where it stopped. Of course, such a task only makes sense when reviewing token/ID pairs that do not correspond to echoed IDs (i.e., the IP ID in the reply is the same as in the probe), as shown at lines 10 and 19, and if the difference between two consecutive IDs is small enough to consider they are part of a sequence. This is emphasized in Algorithm 2 with an additional parameter $maxDiff$ (used at lines 14 and 22), which can be set with a relatively small value such as 100. Indeed, an implementation of the whole alias resolution framework should give the user the possibility to tune the maximum difference between two consecutive IP IDs via a parameter, as well as the possibility to tune the probing parameters that are not shown by Algorithm 1.

**Algorithm 2** Correction heuristic (fixing out-of-order token/ID pairs)

```
Require: $T$, a list of tuples collected during a single wave
Require: $maxDiff$, maximum difference between two consecutive IP IDs
 1: procedure TOKENORDERHEURISTIC(List<IPIDTuple> $T$, Integer $maxDiff$)
 2:     $T$.sort(IPIDTuple::compareByID)
 3:     iterator ← $T$.begin()
 4:     prevTuple ← ∅
 5:     for iterator ≠ $T$.end() do
 6:         tuple ← iterator.getContent()
 7:         if prevTuple == ∅ then
 8:             prevTuple ← tuple
 9:             continue
10:         if prevTuple.isEcho() or tuple.isEcho() then
11:             prevTuple ← tuple
12:             continue
13:         diff ← tuple.getID() − prevTuple.getID()
14:         if diff ≤ $maxDiff$ and tuple.getToken() < prevTuple.getToken() then
15:             reverseIterator ← iterator.previous()
16:             prevBis ← tuple
17:             for reverseIterator ≠ $T$.begin() do
18:                 tupleBis ← reverseIterator.getContent()
19:                 if tupleBis.isEcho() then
20:                     break
21:                 diffBis ← prevBis.getID() − tupleBis.getID()
22:                 if diffBis ≤ $maxDiff$ and tupleBis.getToken() > prevBis.getToken() then
23:                     SWAPTOKENS(tupleBis, prevBis)
24:                 else
25:                     break
26:                 prevBis ← tupleBis
27:                 reverseIterator ← reverseIterator.previous()
28:         prevTuple ← tuple
29:         iterator ← iterator.next()
```

The time complexity of the overall probing scheme should be $\mathcal{O}(N)$ where $N$ denotes the total number of alias candidates considered at once. The detailed analysis of this time complexity can be reviewed in Appendix A, with Sec. A.1.1 reviewing the time complexity of Algorithm 1 while Sec. A.1.2 discusses the practical impact of the correction heuristic detailed in Algorithm 2.

## 3.3 Practical alias resolution with fingerprints

As already discussed in Sec. 3.1.2, fingerprints allow hypothetical alias lists to be quickly highlighted as well as how they can be resolved. In particular, two alias candidates can be deemed as ***compatible*** if their fingerprints provide the same initial TTL value(s) (one for regular fingerprints, two for extended fingerprints as defined in Sec. 3.1.3) and the same IP ID counter class (as defined in Sec. 3.1.2), the other components being indicative of what methods could be used or exploratory. Alias candidates can therefore only be aliased if they are compatible with each other.

Due to the notion of compatibility implying that fingerprints must be equal to some extent, an easy way to detect compatible alias candidates consists of sorting the initial candidates in respect of their fingerprints. For instance, the fingerprints can be first sorted according to the initial TTL value of `echo-reply` ICMP packets (or `time-exceeded` replies in the case of extended fingerprints, cf. Sec. 3.1.3), then according to their IP ID counter class, and so on. The exact policy for sorting fingerprints is up to the implementation: what matters is that the sorted fingerprints naturally highlight contiguous sets of compatible alias candidates (much like Example 3.1).

Moreover, with the probing scheme presented in Sec. 3.2, the fingerprinting process normally collected enough data to replicate many of the methods already discussed in Sec. 2.3. No less than four methods are described in subsequent sections: an alias resolution method based on the source IP of `Port-unreachable` ICMP packets (Sec. 3.3.1.1), an `Ally`-like alias resolution scheme (Sec. 3.3.1.2), an IP ID velocity-based method (Sec. 3.3.1.3), and a DNS heuristic (Sec. 3.3.1.4). The systematic selection of a suitable alias resolution method for a set of compatible candidates is explained in detail in Sec. 3.3.2. Finally, a generic algorithm for building alias lists is briefly described in Sec. 3.3.3.

### 3.3.1 Individual methods

#### 3.3.1.1 Source IP of `Port-unreachable` messages

With fingerprints, reusing the alias resolution approach of `iffinder` (cf. Sec. 2.3.1) is immediate. Given a set of compatible alias candidates where fingerprints feature a value other than `*` as for the source IPs of the `Port-unreachable` replies, an alias list can be easily built by grouping the alias candidate with fingerprints displaying the same source IP. For the sake of completeness, an implementation of this method in the context of fingerprinting-based alias resolution should consider the two following scenarios.

- An implementation should make sure that the source IP of a `Port-unreachable` message appearing in a fingerprint is not simply among the other alias candidates, in case that candidate was configured not to reply to UDP probes.
- More broadly speaking, an implementation should also consider the possibility that not all interfaces of a router will be configured to reply to UDP probes, i.e., it should later verify

whether an alias list first obtained on the basis of `Port-unreachable` messages can be further aliased with other candidates using another method.

The second scenario means that replicating the `iffinder` method will more or less create premilinary alias lists which will be completed later with other methods. For the sake of accuracy, current implementations of the whole alias resolution framework presented in this chapter only consider completing such alias lists with methods based on IP IDs. In fact, these approaches are the most reliable next to replicating the `iffinder` approach, while other methods could be considered as a "best effort" way to make up for the impossibility of using more reliable methods. This way, alias lists built partially or completely on the source IPs of `Port-unreachable` replies are always the most likely to be true to the target domain.

#### 3.3.1.2 `Ally`-like method

One of the most well known alias resolution methods based on the IP identification field, the `Ally` method (cf. Sec. 2.3.2), consists of a sequence of three probes in its simplest form. Given two alias candidates, a first probe is sent towards each candidate within a short timespan, each reply coming with a separate IP ID, named respectively $x$ and $y$ ($x$ arriving first). A third probe is then sent to the IP interface that replied first (i.e., the nterface that emitted $x$), resulting in a third IP ID $z$. If $x$, $y$, and $z$ forms a strictly ascending sequence with a small delta between $x$ and $z$, then both candidates can be considered to belong to the same device.

With the probing scheme presented in Sec. 3.2, a sequence of at least four IP IDs is available for each alias candidate (except when said candidate is unresponsive). Moreover, these sequences can be easily interleaved thanks to the tokens and the probing in *waves*. As a result, it is possible to reuse the same ideas as `Ally` on two compatible candidates, as long as two conditions are met.

- Both alias candidates must feature the *healthy* IP ID counter class, i.e., they must have a strictly increasing sequence of IP IDs with at most one rollover.
- It must be possible to interlace the associated sequences with respect to the tokens.

Thanks to the collection of IP IDs with waves of probes (cf. Sec. 3.2.2), the sequences of IP IDs collected for each alias candidate are naturally easy to interlace, meaning the only remaining condition is that both candidates must have the *healthy* IP ID counter class. When these conditions are met, both candidates can be considered to be aliases of each other if a sequence created by interlacing the associated sequences is still consistently increasing (with some tolerance with respect to rollovers). In fact, to put a strong guarantee on the alias pair, two ways of interleaving are checked: e.g., given two sequences $a_1, b_1, c_1, d_1$ and $a_2, b_2, c_2, d_2$, one can check sequences $a_1, b_2, c_1, d_2$ and $a_2, b_1, c_2, d_1$. If both sequences are strictly ascending and features at most one rollover, then both alias candidates are alias of each other. As at least four IP IDs are collected per candidate, this means this approach works with the equivalent of eight probes, while `Ally` only uses three, making the former more thorough than the latter.

**Algorithm 3** `Ally`-like resolution of two candidates (*healthy* IP ID counters)

```
Require: $c_1$, the first alias candidate
Require: $c_2$, the second alias candidate
Require: $k$, the total of IP IDs collected per candidate
Require: $maxShift$, the maximum shift in the IP ID sequence upon a rollover
function AllyMethod(IPEntry $c_1$, IPEntry $c_2$, Integer $k$, Integer $maxShift$)
    $tokens_1 \leftarrow c_1$.getTokensArray()
    $IDs_1 \leftarrow c_1$.getIDsArray()
    $tokens_2 \leftarrow c_2$.getTokensArray()
    $IDs_2 \leftarrow c_2$.getIDsArray()
    i ← 0
    nbRollovers ← 0
    for i < $k$ do
        if $tokens_1[i] < tokens_2[i]$ then
            if $IDs_1[i] > IDs_2[i]$ then
                diff ← (65535 - $IDs_1[i]$) + $IDs_2[i]$
                if diff > $maxShift$ then
                    return $false$
                nbRollovers++
        else
            if $IDs_2[i] > IDs_1[i]$ then
                diff ← (65535 - $IDs_2[i]$) + $IDs_1[i]$
                if diff > $maxShift$ then
                    return $false$
                nbRollovers++
        if nbRollovers > 1 then
            return $false$
        i++
    i ← 0
    nbRollovers ← 0
    for i < $k - 1$ do
        if $IDs_1[i] > IDs_2[i+1]$ then
            diff ← (65535 - $IDs_1[i]$) + $IDs_2[i+1]$
            if diff > $maxShift$ then
                return $false$
            nbRollovers++
        if $IDs_2[i] > IDs_1[i+1]$ then
            diff ← (65535 - $IDs_2[i]$) + $IDs_1[i+1]$
            if diff > $maxShift$ then
                return $false$
            nbRollovers++
        if nbRollovers > 2 then
            return $false$
        i++
    return $true$
```

Algorithm 3 shows the `Ally`-like alias resolution of two alias candidates featuring *healthy* IP ID counters. The algorithm works in two major steps systematized in two loops. The first loop (lines 6–23) checks that the IDs are increasing along the tokens at each index, i.e., each *wave*, and makes sure at most one rollover happens. The second loop (lines 24–39) compares the sequence of IDs while interleaving them on the fly, i.e., lines 27–31 look for the first possibility for interlacing while lines 27–31 review the other scenario, with both being checked at the same loop iteration. Again, at most one rollover should happen here, though line 37 allows for up to two rollovers: this is due to the effects of the rollover being seen in both interleaved sequences, since the loop is checking two sequences at the same time. If all loops can be completed, then $c_1$ can be considered as an alias of $c_2$ and vice versa, as all *return false* instructions correspond to cases where problematic rollovers violate the consistency of a sequence (obtained with interlacing or not). Like Algorithm 2, the provided pseudo-code features an additional parameter $maxShift$ to fine-tune the resolution. This parameter expresses the largest accepted shift in a sequence upon a rollover, ensuring sequences with rollovers are handled correctly.

Though Algorithm 3 is only designed to resolve two alias candidates, it can be easily extended to sets of alias candidates thanks to the property of alias transitivity (cf. Sec. 2.2), i.e., a reference candidate can be picked from the compatible candidates, with each subsequent candidate being compared to it to build the final alias list. New candidates will also be aliases of previous candidates by transitivity. It is also possible to compare an unresolved candidate with each individual candidate that is already part of an alias list, to put a stronger guarantee on the result, but this means replicating the $\mathcal{O}(N^2)$ time complexity of `Ally` if $N$ denotes the number of IP addresses found in the final alias list. Hopefully, since the resolution itself does not require additional probing and only amounts to some additional computational work after the probing, the quadratic complexity would have only a minor effect as it would not change any probing overhead.

#### 3.3.1.3 IP ID velocity method

The sequences of IP ID values as collected with the probing scheme demonstrated in Sec. 3.2.2 can also be used to evaluate the velocity of the IP ID counters that emitted them, as long as a timestamp value is recorded when an IP ID value is collected (as suggested for the tuples described in Sec. 3.2.3). By computing the difference between the timestamps recorded for two successive IP IDs, delays between two successive IP IDs can be computed. Though it is unlikely these delays will correspond precisely to the delay experienced at the side of the target interface, as they measure wall clock time at the side of the vantage point, they can be considered as a best effort approximation of the actual delay. Using both the IP ID values and the delays makes it possible to estimate the *IP ID rate*, i.e., how fast the IP ID counter that emitted the IP IDs is increasing over time. Two (compatible) alias candidates who show comparable or identical rates can be considered to be the alias of each other. Several state-of-the-art alias resolution methods (see Sec. 2.3.3) rely on this approach rather than on IP ID sequence interleaving (i.e., `Ally`-like) as this simplifies the probing work: any alias candidate can be probed with a constant number of probes regardless of which candidate it will be aliased with. In other words, tools based on this approach can achieve a linear time complexity.

However, due to the typical use cases of the present alias resolution framework (cf. Sec. 3.2.1) and the probing scheme itself, alias candidates with fingerprints featuring the *healthy* IP ID counter class will already have been processed with the help of the `Ally`-like method (cf. Sec. 3.3.1.2). Therefore, a velocity-based method should be considered if and only if the alias candidates have neither the *echo*, nor the *healthy* IP ID counter class (cf. Sec. 3.1.2). As a reminder, this can occur when a sequence of IP IDs features no echoed IP IDs and more than one rollover at the same time. It is possible, however, to attempt to compute a range for the IP ID rate of each of these candidates, which allows to alias two compatible candidates if their respective ranges of IP ID rates overlap each other. The final IP ID counter class of an alias candidate will depend on whether a consistent range of IP ID rates can be discovered: if this range can be computed while modeling a fairly low number of rollovers, the candidate will get the *fast* IP ID counter class, otherwise it will get the *random* IP ID counter class. This alias resolution can be systematized in two steps: the estimation of the IP ID rate within a range for each candidate, which also results in picking the IP ID counter class between *fast* and *random*, and the subsequent comparison of these ranges if the candidates get the *fast* class.

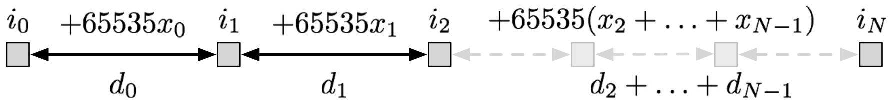


Figure 3.1: Schematic view of the variables used for estimating IP ID rates

Estimating an IP ID rate requires modeling potential additional rollovers not seen with the probes and is carried out as follows. For every pair of consecutive IP IDs (denoted as $i_i$ and $i_{i+1}$) in a sequence and the timelapse between those IP IDs $d_i$, a variable $x_i$ can be considered to model the number of additional rollovers that occurred between $i_i$ and $i_{i+1}$, $x_i = 0$ meaning that no rollover occurred if $i_{i+1} > i_i$. It also means that exactly one rollover happened if $i_{i+1} < i_i$. Fig. 3.1 illustrates these variables and unknowns for a sequence of $k$ IP IDs, with the value 65535 denoting one additional rollover (for reminders, 65535 is the largest possible value for an IP ID). To find all $x_i$, a fixed number of additional rollovers is first assigned to $x_0$, i.e., the number of additional rollovers for the two first IP IDs. Then, using this $x_0$ and the known IP IDs and delays, $k-2$ equations are solved to find the number of additional rollovers that occurred between the other pairs of successive IP IDs. Equation 3.1 formalizes the typical equation necessary to find the value of $x_i$, the variable denoting the number of additional rollovers that occurred between the IP IDs $i_{i+1}$ and $i_i$, assuming each pair of successive IP IDs resulted in a negative delta (i.e., $i_{i+1} < i_i$). The expression equates the rate observed for the first pair (left) and the other pair (right), as the rate should stay quasi-constant over short periods. When a pair results in a positive delta (i.e., $i_{i+1} > i_i$), $i_{i+1} + (65535 - i_i)$ should be replaced with $i_{i+1} - i_i$ so the number of times the IP ID counter was incremented is accurately expressed.

$$\frac{i_1 + (65535 - i_0) + 65535 \times x_0}{d_0} = \frac{i_{i+1} + (i_i - 65535) + 65535 \times x_i}{d_i} \tag{3.1}$$

One way to find solutions for these equations is to find a real solution for each $x_i$ (except $x_0$, previously set) and rounding them up to the closest integer in order to stay true to the real world, as additional rollovers must strictly be modeled by naturals multiplied by 65335 (i.e., 65535, 131070, 196605, etc.). To do so, a formula deriving from Equation 3.1 can be used to compute $x_i$. The development of Equation 3.1 to the resulting formula is shown at Equation 3.2. It should be noted, at this point, that this equation is written as if there were only negative deltas (i.e., $i_{i+1} < i_i$) in the sequence of IP IDs, but if one of the pairs of IP IDs results in a positive delta (i.e., $i_{i+1} > i_i$), any sum $i_{i+1} + (65535 - i_i)$ must be replaced with $i_{i+1} - i_i$ in the development. These sums are handled linearly, which means that there is no change the overall development. For reference, Equation 3.3 shows the formula obtained if there are only positive deltas.

$$\begin{aligned}
\frac{65535 \times x_i + i_{i+1} + (65535 - i_i)}{d_i} &= \frac{65535 \times x_0 + i_1 + (65535 - i_0)}{d_0} \\
\Leftrightarrow 65535 \times x_i + i_{i+1} + (65535 - i_i) &= \frac{d_i \times (65535 \times x_0 + i_1 + (65535 - i_0))}{d_0} \\
\Leftrightarrow x_i + \frac{i_{i+1} + (65535 - i_i)}{65535} &= \frac{d_i \times 65535 \times x_0}{65535 \times d_0} + \frac{d_i \times (i_1 + (65535 - i_0))}{65535 \times d_0} \\
\Leftrightarrow x_i &= \frac{d_i \times x_0}{d_0} + \frac{d_i \times i_1 + d_i \times (65535 - i_0)}{65535 \times d_0} - \frac{i_{i+1} + (65535 - i_i)}{65535}
\end{aligned} \tag{3.2}$$

$$x_i = \frac{d_i \times x_0}{d_0} + \frac{d_i \times i_1 - d_i \times i_0}{65535 \times d_0} - \frac{i_{i+1} - i_i}{65535} \tag{3.3}$$

For consistency's sake, it must be checked that each rounded $x_i$ is sound by checking it is positive (negative rollovers do not exist) and that the rounding error is reasonably low enough (e.g., below 0.25). Tolerating some rounding error is sound since the IP ID rate of the counter tied to some device will not be strictly constant in the real world, although this error should not be too large: for instance, having half a rollover as a result of an equation means the model does not fit well enough to allow for the observed sequence of IP IDs. Each time a rounded $x_i$ value either is negative or has a large rounding error, the value of $x_0$ is increased by one and all $k - 2$ equations are solved again. The equation solving ceases if all the rounded $x_i$ values are positive and only have small enough rounding errors or if $x_0$ reaches a pre-defined upper value for the number of additional rollovers (e.g., 5). If the former is the case, an IP ID rate $v_i$ is computed for each pair of successive IP IDs, and the minimum and maximum $v_i$ are kept as the boundaries for the range of the IP ID rates of the alias candidate. Equation 3.4 shows how each $v_i$ can be computed after obtaining the $x_i$ values, assuming $i_{i+1} < i_i$ (negative delta). However, if the upper limit for the number of additional rollovers is reached, the alias candidate receives the *random* IP ID counter class as a way to tell that the candidate is more likely to be generating random values than producing a sound sequence of IP IDs, even at a fast rate.

$$v_i = \frac{i_{i+1} + (65535 - i_i) + 65535 \times x_i}{d_i} \tag{3.4}$$

Algorithm 4 formalizes, in pseudo-code, the estimation of the IP ID rate for a given alias candidate $c$, provided it has a sequence of $k$ IP IDs, and two additional parameters acting as upper boundaries on the number of (additional) rollovers and the rounding error, respectively. After setting some variables, the main loop (lines 7–36) starts and sets a first value for $x_0$. The inner loop (lines 13–33) solves the $k-2$ equations, each iteration solving an equation (lines 14–24), ensuring each time whether it handles a positive or a negative delta, and computing the resulting $v_i$ as long the rounded $x_i$ value is consistent (lines 25–32). After exiting the main loop, as a result of successfully finding consistent $x_i$ or reaching too many additional rollovers, the algorithm ends by attributing the final IP ID counter class to the alias candidate and assigning the resulting $v_i$ (lines 37–41). In the pseudo-code, the creation of the range is assumed to have been carried out using the `assignIPIDRates()` method. The counter classes are also denoted by `RANDOM_COUNTER` and `FAST_COUNTER` constants for the sake of readability (an implementation should also uses constants for the same purpose).

After estimating the IP ID rate as a range for each alias candidate not falling into the *echo*/ *healthy* categories, not only can they receive their own final IP ID counter classes (*fast* or `random`), but also their additional data regarding IP ID rates can be used for actual alias resolution. Two alias candidates featuring the same initial TTL value(s) and the *fast* IP ID counter class can be considered to be compatible and can be eventually aliased if their respective ranges of IP ID rates overlap each other. An additional guarantee on an hypothetical alias pair can be added by ensuring it is possible to match IP ID values within a range from either sequence using the rates (or the sequence of tokens) from the other sequence. Whether such an heuristic should be used and how it can be implemented and fine-tuned are left to the implementation.

Finally, it is worth noting that estimating the IP ID rates of a set of alias candidates is achievable immediatly following probing (cf. Algorithm 1) in order to obtain the final IP ID counter classes as soon as possible. This is because the actual alias resolution, achieved by comparing the ranges, is, more often than not, the final step of the entire framework (i.e., after selecting a suitable method, cf. Sec. 3.3.2). Due to the parameters $k$ and $r$ being fixed and small, it can also be assumed the IP ID rate estimation is achieved in constant time ($\mathcal{O}(1)$) regardless of the scenario of alias resolution.

The case of two alias candidates with the *random* IP ID counter class leaves out few possibilities: either there is a source IP from a `Port-unreachable` ICMP packet (cf. Sec. 3.3.1.1) for each candidate implying the alias pair (or a single one, equal to the IP of the other candidate), or there is a DNS record for each candidate such that a reverse DNS heuristic (cf. Sec. 3.3.1.4) can be used to make sure the alias pair remains sound. Otherwise, there is no viable method for alias candidates with the *random* class, other than considering that having similar fingerprints is enough to alias them.

**Algorithm 4** Evaluating the IP ID rate of an alias candidate (neither *echo*, neither *healthy*)

```
Require: c, an alias candidate with neither the echo neither the healthy IP ID counter class
Require: k, the total of IP IDs collected per candidate
Require: maxRollovers, an upper boundary on the number of rollovers
Require: maxError, an upper boundary on the estimation error
 1: procedure ESTIMATEVELOCITY(IPEntry c, Integer k, Integer maxRollovers, Float maxError)
 2:     IDs ← c.getIDsArray()
 3:     delays ← c.getDelays()
 4:     x_0 ← 0
 5:     v ← new Float[k − 1]
 6:     success ← true
 7:     for x_0 < maxRollovers do
 8:         if IDs[1] > IDs[0] then
 9:             v[0] = (IDs[1] − IDs[0] + 65535 ∗ x_0) / delays[0]
10:         else
11:             v[0] = (IDs[1] + (65535 − IDs[0]) + 65535 ∗ x_0) / delays[0]
12:         i ← 1
13:         for i < k − 1 do
14:             x_i ← (delays[i] / delays[0]) ∗ x_0
15:             if IDs[i + 1] > IDs[i] then
16:                 x_i ← x_i − ((IDs[i + 1] − IDs[i]) / 65535)
17:             else
18:                 x_i ← x_i − ((IDs[i + 1] + (65535 − IDs[i])) / 65535)
19:             if IDs[1] > IDs[0] then
20:                 x_i ← x_i + (((delays[i] ∗ IDs[1]) − (delays[i] ∗ IDs[0])) / (65535 ∗ delays[0]))
21:             else
22:                 x_i ← x_i + (((delays[i] ∗ IDs[1]) + (delays[i] ∗ (65535 − IDs[0]))) / (65535 ∗ delays[0]))
23:             roundedX ← ROUNDVALUE(x_i)
24:             roundingError ← ROUNDINGERROR(x_i)
25:             if roundedX > 0 and roundingError ≤ maxError then
26:                 if IDs[i + 1] > IDs[i] then
27:                     v[i] = (IDs[i + 1] − IDs[i] + 65535 ∗ roundedX) / delays[i]
28:                 else
29:                     v[i] = (IDs[i + 1] + (65535 − IDs[i]) + 65535 ∗ roundedX) / delays[i]
30:             else
31:                 success ← false
32:                 break
33:             i ← i + 1
34:         if success ≠ false then
35:             break
36:         x_0 ← x_0 + 1
37:     if success ≠ true then
38:         c.setIPIDCounterType(RANDOM_COUNTER)
39:     else
40:         c.setIPIDCounterType(FAST_COUNTER)
41:         c.assignIPIDRates(v)
```

When there is no other option, such a choice can arguably constitute a best effort approach if the alias resolution scenario allows it. This specific question is discussed in Sec. 3.3.2.

#### 3.3.1.4 DNS heuristic

The last alias resolution method that can be implemented with the proposed framework relies on the available DNS records collected through reverse DNS look-up. The fact that an alias candidate has a DNS record does not prevent compatibility with other candidates that do not have such a record (cf. Sec. 3.1.2) and is rather indicative of whether a DNS-based heuristic can be used or not. Moreover, as already discussed in Sec. 2.3.5, alias resolution methods based on DNS records tend to be difficult to tune or may lead to false positives as a consequence of DNS misnaming. A recent study, however, explored the possibility of using machine learning to automatically tune a DNS-based scheme [80]. Due to these hurdles, reverse DNS is only considered as a last resort method in the context of this framework, i.e., when no other method can be used.

The method in this context simply consists of the following heuristic: given a set of compatible alias candidates all having a DNS record, the associated DNS records are split into substrings at the `.` symbol and compared with each others in respect of the substrings. For instance, the DNS record `if-ae-47-2.tcore1.fr0-frankfurt.as6453.net` is split into 5 substrings: `if-ae-47-2`, `tcore1`, `fr0-frankfurt`, `as6453` and `net`. If all chunks of two DNS records are equal, except the first one, it can be assumed that they belong to the same device. For example, the DNS record `if-ae-37-4.tcore1.fr0-frankfurt.as6453.net` can be considered as being on the same machine as the former record, since they share the same suffix `.tcore1.fr0-frankfurt.as6453.net`. This approach is further justified by the fact that a common convention among most network operators consists of writing details about a network interface in the very first part of the DNS record, while the rest of the record usually represents a device and/or a location, as illustrated by both example records. This heuristic should nevertheless be used with caution, because other naming conventions made by network operators can easily result in false positives.

### 3.3.2 Selection of an alias resolution method

As mentioned at the start of Sec. 3.3, the actual alias resolution starts by first sorting the alias candidates to highlight contiguous subsets of compatible candidates. As already mentioned, two candidates are compatible when their fingerprints bear the same initial TTL value(s) and IP ID counter class. In the next step, a suitable alias resolution method is selected for each subset according to the values found in the fingerprints so that the selected method is as accurate as possible.

As a consequence, the selection of an alias resolution method is, in practice, a prioritization of methods previously described in Sec. 3.3.1. Such a prioritization can of course be freely tuned to the needs of the implementer depending on whether a method has been modified to improve its accuracy (e.g., if the reverse DNS method reused regexes generated by previous research work [80]), but this chapter will present one that was used for the entire course of the research that led to this thesis. In its simplest form, the prioritization suggested in this chapter will consider the methods of Sec. 3.3.1 in the following order.

1. The method based on the source IPs of `Port-unreachable` packets (Sec. 3.3.1.1) will be considered first, due to the way in which it directly suggests alias pairs.

2. Next, the `Ally`-like approach (Sec. 3.3.1.2), which is usually very reliable with alias candidates whose fingerprints exhibit the *healthy* IP ID counter class.

3. When none of the above can be used, and if the IP IDs collected for each candidate can still be used for velocity estimation (Sec. 3.3.1.3), i.e. resulting in the *fast* IP ID counter class, alias resolution by associating ranges of IP ID rate that overlap each other can still be performed.

4. When none of the above can be used, and if DNS records are available for alias candidates, the DNS heuristic (Sec. 3.3.1.4) will be used.

5. If no method at all can be used, it will be up to the implementation to decide whether it should alias together candidates with compatible fingerprints as a best effort approach or not.

In addition to this prioritization, two main challenges must be addressed. The first challenge to address is how to handle the results of the `iffinder`-like method (i.e., item 1). Indeed, whether or not such a method produces alias pairs/lists, it is quite probable that the results will be incomplete: a candidate left out by the resolution might very well be an alias of other candidates through other methods, while an alias pair/list built through this method could also be incomplete (this was already shortly discussed in Sec. 3.3.2). Both cases are handled as follows.

First, when an alias candidate is not successfully aliased to any other candidate via the source IP found in its fingerprint, it is put back onto the list of remaining alias candidates but with the source IP of the `Port-unreachable` reply being *masked*. I.e., it is temporarily over-ridden with the `*` value so that the candidate can be put through other alias resolution methods (i.e., by skipping the `iffinder`-like approach). Second, when an alias pair/list is successfully produced by the `iffinder`-like method, it is later considered to be merged with another alias pair/list obtained by the `Ally`-like method. In this scenario, a merging only occurs if the `Ally`-like method confirms the alias pairs/lists can be merged. Indeed, the initial alias pairs/lists obtained with the `iffinder`-like approach may be incomplete, as not all interfaces of a device will reply to UDP probes. Technically, this means it could be merged with other candidates through any other method, but for the sake of accuracy, current implementations of the framework only attempt merging with alias pairs/lists obtained with the `Ally`-like method to guarantee that the final alias list(s) are as true as possible to the target domain.

The second challenge lies in deciding what to do when there is no suitable alias resolution method available. In this situation, aliasing candidates for which no method can be used can be avoided, or they can be aliased on the basis that they feature compatible fingerprints. The decision is ultimately up to the implementation, but can be discussed depending on the context. Though the overall

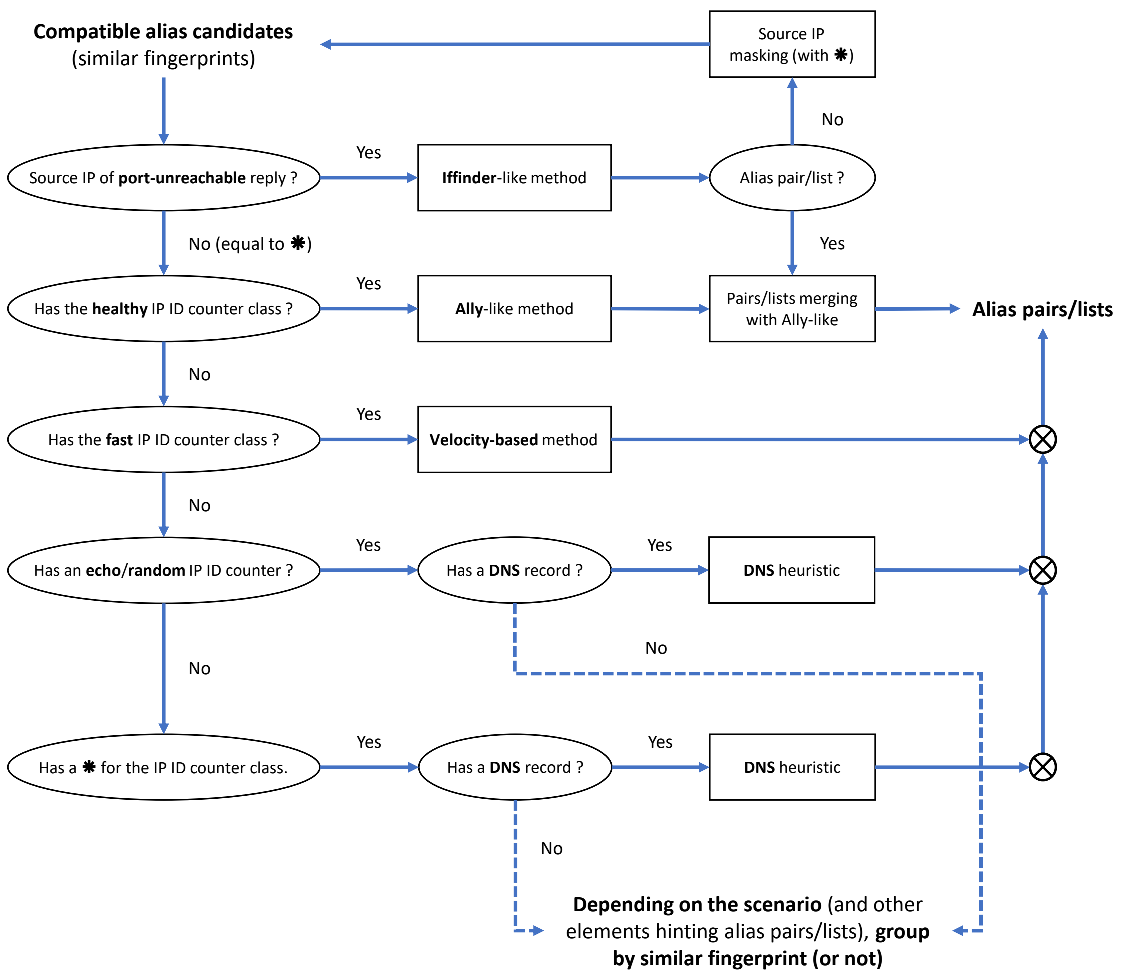

Compatible alias candidates (similar fingerprints)
Source IP masking (with ✱)
No
Source IP of port-unreachable reply ?
Yes
Iffinder-like method
Alias pair/list ?
No (equal to ✱)
Yes
Has the healthy IP ID counter class ?
Yes
Ally-like method
Pairs/lists merging with Ally-like
Alias pairs/lists
No
Has the fast IP ID counter class ?
Yes
Velocity-based method
No
Has an echo/random IP ID counter ?
Yes
Has a DNS record ?
Yes
DNS heuristic
No
No
Has a ✱ for the IP ID counter class.
Yes
Has a DNS record ?
Yes
DNS heuristic
No
Depending on the scenario (and other elements hinting alias pairs/lists), group by similar fingerprint (or not)

Figure 3.2: Practical selection of an alias resolution method as a flow-chart.

framework was designed to handle small scenarios of alias resolution (cf. Sec. 3.2.1), the former decision could be preferred to the latter if the resulting alias pairs/alias lists have an influence on subsequent algorithms. For instance, if alias pairs/lists play a role in mapping a network topology (cf. chapter 11), alias resolution results as accurate as possible could be used to avoid grouping together alias candidates when they feature compatible fingerprints but are not suited to any available method. On the contrary, grouping candidates unfit for any method can be acceptable in scenarios where there are already strong hints that the alias candidates might be aliases of each others anyway, such as in some scenarios of subnet inference (cf. Sec. 6.1.3 of Chapter 6). Whether aliasing candidates unfit for any available method should be considered will be discussed in all subsequent parts of this thesis where the present alias resolution framework is involved.

Figure 3.2 shows an exhaustive flow-chart of how alias resolution methods described in Sec. 3.3.1 are typically selected in the context of this thesis. This flow-chart naturally features the previously described prioritization, but also includes the aforementioned nuances regarding the alias pairs/listed built on the basis of source IPs of `Port-unreachable` packets and regarding the alias candidates for which no practical method can be used. It should be noted that the summing junctions in the flowchart are essentially there to connect the results to the overall output (a set of alias pairs/lists), though they are actually mutually exclusive: if a set of compatible candidates is handled with the `Ally`-like method, it will not be put through the velocity-based method or the DNS heuristic afterwards.

### 3.3.3 Building alias pairs and alias lists

Finally, one final algorithmic detail will be reviewed: how to build alias lists, given that the individual methods (Sec. 3.3.1) were described to resolve two alias candidates at most. The generic algorithm to build an alias list using these methods is formalized by Algorithm 5. Given a set of compatible alias candidates, best handled as a list (named *candidates* in this instance), and a selected method (denoted by *method* in this case), the first candidate on the list can be picked as a *reference candidate* to which any subsequent candidate will be compared (line 5). Then, a loop will process the remaining candidates (lines 7–25), with each iteration attempting to alias the reference with the current candidate (line 10). If the method concludes the reference is an alias of the second candidate, the latter will be put in a side list `aliased`, which will be used later to build the final alias list (line 11). If the *method* concludes that the alias pair does not make sense, the second candidate will be moved to a second side list with candidates on which the method should be tried again to find more alias pairs/lists (line 13). When the *candidates* list becomes empty, the loop will build an alias with the *reference* candidate and the *aliased* list (if not empty), relying on the property of alias transitivity (cf. Sec. 2.2), as shown by lines 15–20. Afterwards, it will refresh the *candidates* list and the *reference* candidate if there is more than one excluded candidate in order to allow the loop to start again and build more alias pairs/lists, cf. lines 21–25. It is worth noting that, due to the generic nature of Algorithm 5, a practical implementation might need additions depending on the selected method (such as the pair/list merging mentioned in Fig. 3.2). Whether it should make a census of unaliased candidates (e.g., by allowing single item alias lists) is also up to the implementation.

## 3.4 Closing comments on the framework

In this chapter, an alias resolution framework based on IP fingerprinting was introduced. Using this framework makes it possible, not only to discover what are the most suitable alias resolution methods for a given set of alias candidates (cf. Sec. 3.3.2), but also to already collect enough data to perform actual alias resolution, as detailed in Sec. 3.2. Such a methodology was designed for small sets of candidates, principally because a single *wave* of probes while collecting IP ID values (cf. Sec. 3.2.2) can take more time with more candidates and be more often subject to network delays or counter rollovers. As such, to deploy the framework at the scale of a whole network, the alias candidates must first be properly identified. Chapter 4 will present a topology discovery tool with a mapping methodology that acts as a form of space search reduction (as defined in Sec. 3.2.1) for alias resolution, a context in which the framework presented in this chapter is a competitive solution, as

**Algorithm 5** Building alias pairs/lists

```
Require: candidates, a list of compatible alias candidates
Require: method, an object modeling the selected alias resolution method
 1: function BUILDALIASLISTS(List<IPEntry> candidates, AliasResolutionMethod method)
 2:     aliasLists ← new List<Alias>
 3:     aliased ← new List<IPEntry>
 4:     excluded ← new List<IPEntry>
 5:     reference ← candidates.front()
 6:     candidates.pop_front()
 7:     for candidates.size() > 0 do
 8:         next ← candidates.front()
 9:         candidates.pop_front()
10:         if method.resolveAliasPair(reference, next) then
11:             aliased.push_back(next)
12:         else
13:             excluded.push_back(next)
14:         if candidates.size() == 0 then
15:             if aliased.size() > 0 then
16:                 newAlias ← new Alias()
17:                 newAlias.add(reference)
18:                 newAlias.add(aliased)
19:                 aliased.clear()
20:                 aliasLists.push_back(newAlias)
21:             if excluded.size() > 1 then
22:                 candidates.copy(excluded)
23:                 excluded.clear()
24:                 reference ← candidates.front()
25:                 candidates.pop_front()
26:     return aliasLists
```

will be demonstrated with a groundtruth network (in Chapter 4). Finally, Chapter 4 will also quantify fingerprints collected in the wild from the PlanetLab testbed and compare the results with previous observations (cf. Sec. 3.1.1) to further demonstrate the potential of IP fingerprinting.

# Chapter 4

# An evaluation of fingerprint-based Alias Resolution

This chapter comprehensively evaluates the fingerprint-based alias resolution framework previously introduced in Chapter 3. A first implementation of the framework, combined with space search reduction (as defined in Sec. 3.2.1), is first presented in Sec. 4.1 and is subsequently used to assess the framework in Sec. 4.2, where the implementation is validated on a groundtruth network while being compared with other state-of-the-art tools. These two sections answer research question 3 from Sec. 2.5. Sec. 4.3 subsequently answers research question 4 by discussing observations with the fingerprints introduced in Sec. 3.1.2, using data collected from the PlanetLab testbed on numerous Autonomous Systems during the course of 2017. Finally, Sec. 4.4 concludes both this chapter and this first part by reviewing the key contributions and discussing how the methodology of the framework could prove useful for the future of alias resolution.

## 4.1 The first implementation with space search reduction

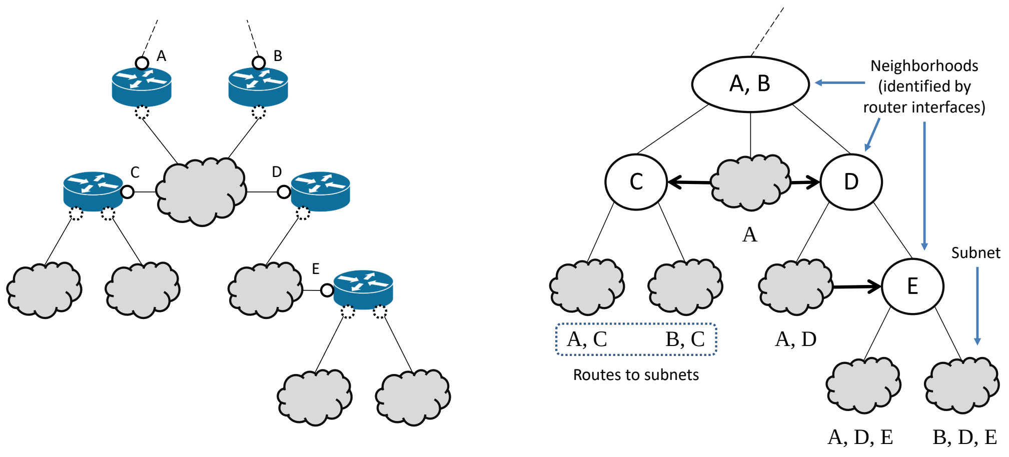
### 4.1.1 Space search reduction with `TreeNET`

The framework presented in Chapter 3 was first implemented in the IPv4 topology discovery tool `TreeNET` [55]. It was also assessed with the help of this first implementation, as will be discussed in Sec. 4.2 and Sec. 4.3, though it has also been included (with slight improvements over the years, cf. Sec. 4.1.2) in several other tools that will be discussed in subsequent parts of this thesis. `TreeNET` was used for the first implementation of the framework mainly because it was developed prior to the research work presented in this thesis [1] and provided an off-the-shelf space search reduction solution to apply and assess the whole methodology [51].

[1] `TreeNET` was the main contribution of my master thesis, which I presented during academic year 2014-2015. The first paper on `TreeNET` [55] presented an improved version of this work.

Thanks to its subnet-based network mapping methodology, `TreeNET` naturally highlights network places that could consist of a single router or a mesh of routers, with the latter potentially involving Layer-2 equipment. These places are called ***neighborhoods***, and are formally defined as network locations bordered by a set of subnets that are all located at most one hop away from each others. `TreeNET` identifies these locations within a target network by building a tree-like map using subnet-level data and (Paris) `traceroute` [11] paths collected towards each inferred subnet. In the tree, route paths are modeled by successions of branching out nodes, each denoted by the corresponding route hop, while subnets act as the leaves. Subnets that share a common route should therefore be inserted under the same internal node, which therefore amounts to a single neighborhood.

A collection of `traceroute` paths towards a target network are usually closer to a directed acyclic graph than to a tree, as a consequence of forwarding dynamics and load balancing [14, 98]. In other words, a same hop can feature several predecessors within `traceroute` paths, while the tree format implies there can be only one. To solve this issue, `TreeNET` allows the construction of *multi-label* nodes, i.e., nodes that model several router interfaces observed at the same hop count that can all act as the parent to subsequent nodes. Multi-label nodes account for parts of the network topology where several paths are leading to the same endpoints. They therefore ensure that subnets with differing routes but which are the same length and end with the same last hop(s) can be gathered under the same internal node, and therefore the same neighborhood. This construction also guarantees any IP address appears only once at a given depth (i.e., hop count).



(a) Toy network (circles depict interfaces) (b) As a tree (clouds depict subnets)

Figure 4.1: Mapping a network with `TreeNET`. Each subnet features a single `traceroute` path.

Figure 4.1 presents a toy network (Fig. 4.1a) and how it would be interpreted with the tree formalism implemented by `TreeNET` (Fig. 4.1b) [2]. In Fig. 4.1a, the plain-line circles denote interfaces that can be seen with (Paris) `traceroute` measurements, while the grey clouds represent subnets. The upmost subnet symbolizes a larger subnet that acts as a LAN between the routers identified by the interfaces $A$, $B$, $C$, and $D$. In Fig. 4.1b, the internal nodes labeled with router interfaces $C$, $D$, and $E$ depict neighborhoods that are representative of the target network thanks to the introduction of

[2]This figure is reminescent of Fig. 1.8 from Sec. 1.1.1, but accounts for the multi-label mechanism.

the multi-label node $\{A, B\}$. In particular, subnets under $E$ have different routes (starting with either $A$, either $B$) but share the same last hops and length, which is why they are found around the same neighborhood. The same goes for subnets from $C$. The black arrows in Fig. 4.1b show a subnet which was identified as encompassing another route hop (i.e., it is within the subnet prefix), suggesting that this subnet is the link between two successive neighborhoods. For example, the subnet with the route $A, D$ is identified as the link between the neighborhoods identified by $D$ and $E$.

A benefit from discovering neighborhoods lies in the fact that the surrounding subnets can encompass interfaces which can be identified as router interfaces as well. Indeed, a subnet should, in theory, always feature at least one interface located on the router that provides access to it from the perspective of the vantage point. Usually, these interfaces can be distinguished from others because they require one hop less to obtain an `echo-reply` message from them [3]. Therefore, for each neighborhood, both the router interfaces detected in the gathered subnets and the route hop(s) denoting the neighborhood can be combined to build a list of alias candidates. Because of how they were located by `TreeNET`, these interfaces should not be aliased with interfaces belonging to other neighborhoods. In other words, the discovery of neighborhoods acts as a space search reduction for alias resolution: rather than considering all router interfaces discovered via the `traceroute` measurements or all responsive IPs together, (much) smaller scenarii can be resolved separately. To do so, `TreeNET` implements the fingerprint-based framework presented in Chapter 3, as it was tailored to resolve small alias resolution scenarios while using only a few probes. Figure 4.2 pictures a neighborhood as discovered by `TreeNET` and the alias resolution scenario that results from it.

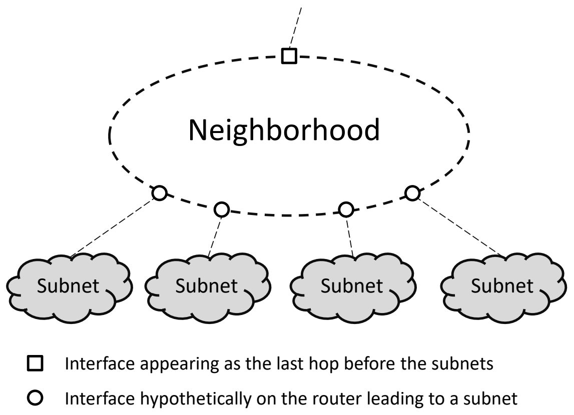

Neighborhood
Subnet
Subnet
Subnet
Subnet
□ Interface appearing as the last hop before the subnets
○ Interface hypothetically on the router leading to a subnet

Figure 4.2: Alias resolution scenario resulting from a neighborhood discovered by `TreeNET`.

The intricate details of the methodology of `TreeNET` regarding subnet inference and neighborhood discovery are out of the scope of this part of the thesis. However, `TreeNET` and related topics (in particular, subnet inference and neighborhood discovery) are discussed in more detail in subsequent chapters. The same chapters also introduce other practical alias resolution scenarios (some being reminiscent of Sec. 2.4) and space search reduction methods. In other words, the fingerprint-based alias resolution framework acts as the generic tool for alias resolution in this entire thesis.

[3]These interfaces are also called *contra-pivot* interfaces. A thorough definition will be provided in Chapter 5.

### 4.1.2 Remarks on the implementation(s) of the framework

Before moving to the assessment of the framework on a groundtruth network (Sec. 4.2) and to the evaluation of fingerprints collected from the PlanetLab testbed (Sec. 4.3), there are several important points to make regarding the implementation(s). Indeed, the framework presented in Chapter 3 corresponds to its final version, i.e. its state at the time of writing this manuscript, but the data used to produce the results showed in the subsequent sections was collected with earlier versions.

The main differences between the final version (notably found in `SAGE` [54], cf. Chapter 11) and the implementation in `TreeNET` lie in the handling of alias resolution methods based on the IP identification field. Though the probing scheme introduced in Sec. 3.2 was already definitive in `TreeNET`, there were two main differences when it came to the actual methods.

- First, the `Ally`-like method in `TreeNET` differs from the lattest implementation: rather than taking advantage of the collection of IP IDs in *waves* (cf. Sec. 3.2.2), the method at the time still assumed that the sequences of IP IDs could not necessarily be interlaced in respect of the tokens. It therefore first looked for tokens that could lead to an interlaced sequence, then verified the interlaced sequence in a manner that was much closer to the original `Ally`. The improved method was designed in early 2018.

- Second, there was no *fast* IP ID counter class, though the velocity-based method was already available. In fact, the velocity estimation (cf. Sec. 3.3.1.3) was performed on all candidates labelled with the *healthy* IP ID counter class and resulted in replacing the *healthy* class with the *random* class when no satisfying estimation could be achieved. Then, when the former `Ally`-like method (see above) did not find an opportunity for interleaving sequences, the implementation simply replaced it with the velocity-based . The *fast* class was later created as a consequence of the refreshed `Ally`-like method (cf. 3.3.1.2), because the improved framework was tuned to not try this approach when alias candidates had sequences with more than one rollover. The *fast* class was therefore created to set apart *random* IP ID counters from counters for which IP ID rate estimation was still possible.

These two differences are still present in the last version of `TreeNET`, for which the source code can be browsed on the dedicated public GitHub repository. [4] Additionally, the latest version of `TreeNET` does not feature extended fingerprints (cf. Sec. 3.1.3), which were added in other tools in late 2019.

Finally, it should be noted that the framework as implemented in `TreeNET` considers the space search reduction sufficient to motivate aliasing candidates on the basis of having similar fingerprints when no available method can be used (see also Sec. 3.3.2). In particular, it aliases by default all (compatible) candidates for which the fingerprints feature the *echo* IP ID counter class and no source IP from `Port-unreachable` packets (i.e., `*`), with one small exception: when DNS records are available, it uses the DNS heuristic to exclude alias pairs.

[4] `https://github.com/JefGrailet/treenet/`

## 4.2 Validation on a groundtruth network

### 4.2.1 Groundtruth and measurement methodology

To assess the framework introduced by this thesis, `TreeNET` was deployed in April 2017 to measure the entire network of the University of Liège, consisting of a /16 IPv4 prefix combined with two additional /24 prefixes used by the backbone. The measurement was conducted from a single vantage point within the network [5]: indeed, firewalls completely shield the vast majority of the network from outside probing. With the friendly help of the SeGI (*Service Général d'Informatique*, i.e., IT services of the University), the alias lists and pairs discovered by `TreeNET` could be confronted with the actual routers of the network at the time, therefore a groundtruth [6], in order to quantify

- the **true positives**, i.e., how many discovered alias pairs were true to the groundtruth,
- the **false positives**, i.e., how many discovered alias pairs were not in the groundtruth,
- the **true negatives**, i.e., how many pairs of interfaces were not alias in both cases,
- and the **false negatives**, i.e., how many pairs of interfaces were not alias in the measurement but actually are in the groundtruth.

Using the responsive IP interfaces detected by `TreeNET` as input, `MIDAR` [73] (see also Sec 2.3.3) was also deployed from the same vantage point in order to collect alias pairs on its own to later compare the results of both tools. Finally, (Paris) `traceroute` paths were collected towards all responsive IP addresses previously detected by `TreeNET` in order to later feed them to `kapar` [72], an analytical alias resolution tool (see also Sec. 2.3.6). The comparison with `kapar` was motivated by the fact that `TreeNET` is partially an analytical solution, due to how it capitalizes on its network mapping methodology to reduce the search space for alias resolution.

An important point to make before presenting the results of the validation is that the network of the University of Liège was well suited for alias resolution methods based on the IP identification field (cf. Sec. 2.3.2 and Sec. 2.3.3) at the time (part of the equipment has been updated since then). As a consequence, the present validation assesses the efficiency and accuracy of the probing scheme used by the fingerprint-based framework (cf. Sec. 3.2) and the accuracy of the space search reduction method (cf. Sec. 4.1) rather than assessing the ability of the framework to make the most of each alias resolution method. This particular point will be however discussed in the next section (Sec. 4.3).

[5] From the former WiFi network *ULg-Secure*.

[6] Even if the topology of ULiège network has evolved since 2017, both in structure and equipment, the details of the groundtruth will not be provided for security reasons.

| | `TreeNET` | `MIDAR` | `kapar` |
|---|---|---|---|
| True positive rate | 81.78% | 98.14% | 0.19% |
| False positive rate | 0.22% | 0.29% | 0.12% |
| False discovery rate | 3.6% | 3.65% | 91.67% |
| Precision | 96.39% | 96.35% | 8.33% |
| Accuracy | 98.6% | 99.6% | 94.47% |
| Duration of the alias resolution | 3'45" | 1h47 | A few seconds |
| Duration including preliminary probing | 1h56 | 2h28 | 2h12 |
| Total of probes during alias resolution | 1,948 | 532,172 | 0 |
| Total of probes (with preliminary probing) | 146,015 | 659,985 | 176,786 |

Table 4.1: Validation of `TreeNET`, `MIDAR` and `kapar` on the ULiège network (April 2017).

### 4.2.2 Validation results

Table 4.1 provides the results of the validation performed in April 2017 with `TreeNET`, `MIDAR` and `kapar` on the network of the University of Liège. The first part of the table provides the ratios of true and false positives to the total of alias pairs found in the groundtruth, completed with the false discovery rate, the precision, and the accuracy. For reminders, if the number of true positives are denoted as $\#TP$ and the amount of false positives by $\#FP$, the number of true negatives as $\#TN$ and the number of false negatives by $\#FN$,

- the **false discovery rate** is equal to $\frac{\#FP}{\#TP+\#FP}$,
- the **precision** is equal to $\frac{\#TP}{\#TP+\#FP}$,
- and the **accuracy** is equal to $\frac{\#TP+\#TN}{\#TP+\#FP+\#TN+\#FN}$.

The second part of Table 4.1 provides some metrics in respect of the probing and the execution time. In particular, it shows how many probes where used during alias resolution (this includes the probing scheme of the fingerprint-based framework) and how many probes were sent in total. The first time metric provides the total execution time during alias resolution, while the second time metric provides the total execution time if all preliminary probing is taken into consideration. Therefore, the total time for `TreeNET` included scanning the network to find responsive IP interfaces, inferring and post-processing subnets, and building the tree-like map with the help of (Paris) `traceroute` – in addition to the alias resolution step. The total time for `MIDAR` was the sum of the execution time of the algorithmic step of `TreeNET` finding responsive IP interfaces and `MIDAR` execution time. Finally, the total execution time and total of probes for `kapar` amount to the discovery of responsive IP interfaces by `TreeNET` plus the collection of `traceroute` paths towards them.

The main highlight of Table 4.1 is that the combination of space search reduction and the fingerprint-based framework rivals `MIDAR` in terms of accuracy and is drastically more economic in terms of probes. Indeed, less than 2,000 probes were used by `TreeNET` for its own alias resolution step, for a total run time of less than four minutes, while `MIDAR` took almost two hours and sent more than half a million probes in total. Moreover, prior to its alias resolution step, `TreeNET` collected additional useful topological data, such as subnet prefixes and the tree-like mapping, i.e., data that `MIDAR` did not provide. The amount of probes used to collect this additional data was still way below the total amount used by `MIDAR`: to make a census of responsive IP addresses, discover subnets, and record `traceroute` paths towards them, `TreeNET` sent 144,067 probes, meaning the total was equal to 146,015 probes if the probes used for alias resolution are added. In a nutshell, `TreeNET` sent more than three times less probes than `MIDAR` whilst collecting more data. Moreover, `TreeNET` used a generous timeout value while probing for responsive IP interfaces, i.e., equal to two and an half seconds, meaning its total run time could have been even shorter. Finally, it is worth noting that `MIDAR` sent an average of 37 probes to each alias candidate during its second stage, i.e., during which time it estimated the speed of the IP ID counters tied to each candidate, while the probing scheme of the fingerprint-based framework only required 6 direct probes for each candidate (cf. Sec. 3.2.2) unless temporary network failures lead to additional probes. The alias resolution as implemented in `TreeNET` also featured a slightly better false positive rate, false discovery rate and precision.

However, the `TreeNET` column in Table 4.1 also shows a lower ratio of true positives than in the `MIDAR` column. A careful look into the alias pairs and lists discovered by `TreeNET` shows that, for some large alias lists, one or two interfaces were missing, therefore leading to many alias pairs being absent from the true positives. Further investigation showed that the few interfaces missing from the large alias lists were actually found at different hop counts than all other interfaces during the probing. Therefore, the problem here was not the alias resolution framework itself, but the space search reduction method. In fact, the method is prone to suffer from the effects of traffic engineering, such as asymmetrical paths in load balancers, or from the effects of network (mis)configurations. In the case of the ULiège network, the unexpected hop counts were the result of network redirections. By manually fixing one of the largest alias lists in the data produced by `TreeNET`, the true positive rate rose to 85.07%, clearly demonstrating that the accuracy of space search reduction is critical to achieve accurate alias resolution with the fingerprint-based framework.

Finally, it goes without saying that `kapar` is not well suited to this scenario. Indeed, most if not all its correct aliases were found in the backbone of the network, and moreover, our groundtruth data does not contain all of it, which is why the true positive rate is so low. Therefore, using `kapar` in roughly the same conditions as `TreeNET` (or a comparable tool) should be avoided. Instead, discovering alias pairs with `kapar` should at the very least involve measurements from multiple vantage points (as already suggested by their authors) or be combined with active methods, as suggested by similar research (cf. Sec. 2.3.6) and as already done in practice by `TreeNET`.

## 4.3 Study of fingerprints in the wild

This section studies fingerprints, and more broadly fingerprint-based alias resolution, using data collected from various networks from the PlanetLab testbed with `TreeNET` during the course of 2017. Sec. 4.3.1 presents the target networks as well as how they were selected and measured. Subsequent sections study the properties of the collected fingerprints and discovered alias pairs: Sec. 4.3.2 discusses the distribution and correlation of the initial TTL value of `echo-reply` packets and the IP ID counter class, Sec. 4.3.3 evaluates the distribution of alias resolution methods across discovered pairs and Sec. 4.3.4 comments on compliance with the ICMP timestamp request mechanism. Finally, Sec. 4.3.5 evaluates the space search reduction provided by `TreeNET`.

### 4.3.1 Deployment on the PlanetLab testbed

During the first semester of 2017, `TreeNET` was deployed from the PlanetLab testbed in order to measure 20 different Autonomous Systems (or ASes), each from a single unique vantage point. These 20 ASes were selected for their size, i.e., how many attributable IP addresses their IPv4 prefixes covered, and their classification by the research community. As a reminder, ASes are categorized into either *Tier-1*, either *Transit* or *Stub*, a classification that accounts for the economic relationships between ASes: *Tier-1* ASes only act as providers to other ASes, while *Stub* ASes are exclusively clients to other ASes, *Transit* ASes being both clients and providers to other ASes.

Due to their relationships with other ASes, the topology of a *Tier-1* can differ vastly from the topology of a *Stub* AS, even if the former has fewer attributable addresses than the latter. The criterion of the size is nevertheless important in the case of `TreeNET`, because a network with too many attributable IP addresses can prove too difficult to measure on a regular basis, especially if this network has a high responsitivity to probes (i.e., large amounts of subnets are detected). As a consequence, the largest ASes measured with `TreeNET` usually have less than 1.5 million of attributable IP addresses. Probing an entire network from a single vantage point facilitates the construction of a tree-like map (all `traceroute` paths will have the same *root*).

Table 4.2 lists the ASes probed with `TreeNET` from PlanetLab during the first semester of 2017. A dataset provided by CAIDA [7] was used at the time to learn the ASes classification, while the Hurricane Electric BGP toolkit [4] provided the IPv4 prefixes associated with each one. It goes without saying that the column providing the attributable IP addresses of each AS only gives the amount at the time, as the prefixes (and topology) may have changed since then. For instance, at the time of writing, the total amount of attributable IP interfaces of AS224 announced by the Hurricane Electric BGP toolkit was equal to 1,050,112, which is slightly less than the aforementioned 1,115,392. Note how Table 4.2 is also split in two: the top half corresponds to ASes that were measured once every few days, while the bottom half accounts for ASes that were measured on a daily basis. Finally, the first column is meant

[7] `https://data.caida.org/datasets/as-relationships/`

| Index | ASN | Name | Origin | Category | Attributable IPs |
|---|---|---|---|---|---|
| 1 | 109 | Cisco Systems | USA | Stub | 1,173,760 |
| 2 | 10010 | TOKAI Communications | Japan | Transit | 1,445,376 |
| 3 | 224 | UNINETT | Norway | Stub | 1,115,392 |
| 4 | 2764 | AAPT Limited | Australia | Transit | 993,536 |
| 5 | 5400 | British Telecom | UK | Transit | 1,385,216 |
| 6 | 5511 | Orange S.A. | France | Transit | 911,872 |
| 7 | 6453 | TATA Communications | USA | Tier-1 | 656,640 |
| 8 | 703 | Verizon Business | USA | Transit | 863,232 |
| 9 | 8220 | COLT Technology | UK | Transit | 1,342,720 |
| 10 | 8928 | Interoute Communications | USA | Transit | 827,904 |
| 11 | 12956 | Telefonica Global Solutions | Spain | Tier-1 | 209,920 |
| 12 | 13789 | Internap Holding LLC | USA | Transit | 96,256 |
| 13 | 14 | Columbia University | USA | Stub | 339,968 |
| 14 | 22652 | Fibrenoire, Inc. | Canada | Transit | 76,288 |
| 15 | 30781 | Jaguar Network | France | Transit | 45,312 |
| 16 | 37 | Navy Net. Info. Center (NNIC) | USA | Stub | 140,544 |
| 17 | 4711 | INTEC, Inc. | Japan | Stub | 17,408 |
| 18 | 50673 | Serverius Holding B.V. | Netherlands | Transit | 61,696 |
| 19 | 52 | University of California | USA | Stub | 328,960 |
| 20 | 802 | University of York | Canada | Stub | 71,936 |

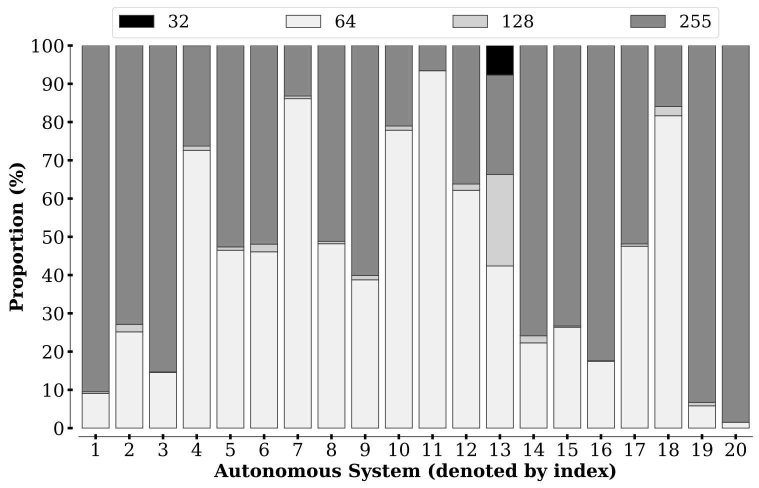
Table 4.2: Autonomous Systems probed with `TreeNET` during the first semester of 2017.

to attribute a unique integer to each AS in order to make it easier to identify them in subsequent figures, as each of these figures illustrates results for all 20 ASes at once.

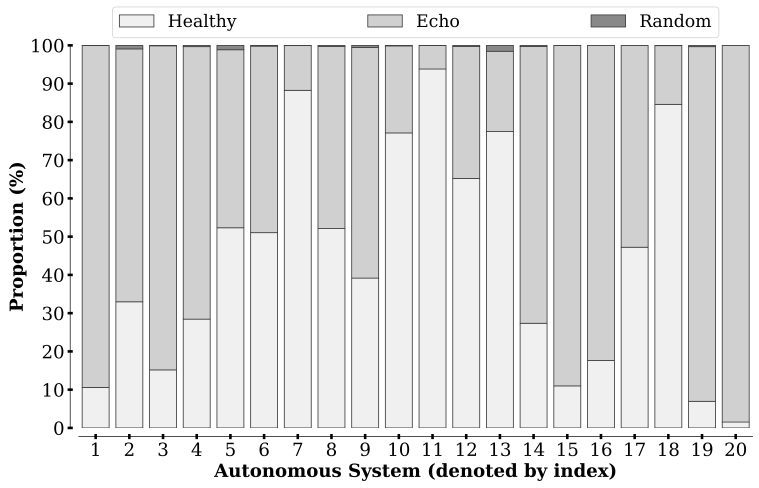
### 4.3.2 Distribution and correlation of initial TTL and IP ID counter class

First, the distribution of the typical values for the initial TTL value of `echo-reply` packets and the distribution of the IP ID counter classes were comprehensively studied. As previously explained, these values are considered as strictly discriminating in the alias resolution framework (cf. Sec. 3.1.2), i.e., they are used to build subsets of alias candidates that could be aliased with each others, therefore excluding alias pairs before even using a method.

Figure 4.3 shows the distributions of the initial TTL value and of the IP ID counter class among fingerprints collected for the 20 ASes from Table 4.2, as stacked bar charts, denoted in the figures by their respective index in the far left column. The data used to build the figure comes from measurements that were started on April 14, 2017. This data is also representative as the observed properties do not change significantly if the same AS is measured on a different date. It is worth noting that Fig. 4.3b does not mention the *fast* IP ID counter class, since it was created after the measurement campaign discussed in this chapter (cf. Sec. 4.1.2). Figure 4.3 also does not consider the fingerprints of alias candidates for which the probing was unsuccessful (i.e., they did not reply to probes while fingerprinting them).



(a) Initial TTL of `echo-reply` packets



(b) IP ID counter class

Figure 4.3: Distributions of fingerprint features among the target ASes (April 14th, 2017).

Several interesting observations can be made from Figure 4.3. The first is that, as already highlighted by Vanaubel et al. in 2013 [121], the initial TTL values 64 and 255 were by far the most common in Fig. 4.3a. There were noticeable shares of fingerprints with the initial value 128 in almost every AS, with the exception of AS14 (13; Columbia University) where roughy 20% of the fingerprints included this value. Interestingly, the initial value 32 also appeared frequently in this specific AS and accounted for slightly less than 10%. Another observation to make is that, in both Fig. 4.3a and Fig. 4.3b, the distributions of either component varied greatly from one AS to another. For instance, the *echo* IP ID counter class was either overwhelmingly common, evenly distributed with respect to the *healthy* class, or in the minority, with the *random* class being noticeable in all cases. What is however striking in both Fig. 4.3a and Fig. 4.3b is how the distributions have the same appearance, especially when looking at the initial TTL value 255 and the *echo* class. This would suggest the *echo* and *healthy* IP ID counter classes were correlated with the initial TTL values 255 and 64/128, respectively.

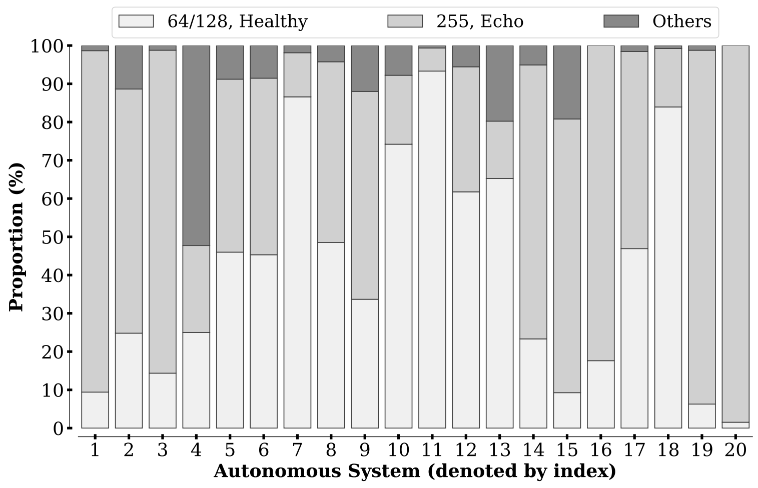


Figure 4.4: Correlation between the initial TTL value and the IP ID counter class (April 14th, 2017).

Figure 4.4 shows the distribution of fingerprints providing both the initial TTL value 255 along with the *echo* IP ID counter class and fingerprints featuring both the initial TTL values 64 or 128 along with the *healthy* counter class, respectively, using the same data and format (stacked bar charts) as Figure 4.3. Fingerprints that did not fall into these categories were accounted for as *others* in the figure. The resulting distribution strongly suggests there was indeed a correlation between both values, though this correlation was not as strong in all ASes: see for instance the case of AS2764 (4; AAPT Limited) where more than 50% of the fingerprints did not fall into either category. This particular case was however more of an outlier, as the correlation was observed for roughly 80% (sometimes more than 90%) of the fingerprints in all other ASes, which is an important result. On the one hand, given how specific profiles of fingerprints should correspond in the real world to different brands of equipment (according to Vanaubel et al. [121]), this would suggest that the suitability of a router to specific alias resolution methods (in this instance, methods based on the IP identification field of the IP protocol) could also be linked to its vendor. To some extent, this proves that using IP fingerprints in the context of alias resolution is sound. On the other hand, this could suggest probing heuristics that would reduce the total number of probes sent towards a target network. For instance, if the initial TTL value of `echo-reply` packets sent by some router interface is quickly identified as 255, it could be assumed there is little chance this router interface could be aliased through methods based on IP IDs (cf. Sec. 2.3.2 and Sec. 2.3.3), therefore avoiding subsequent probes to collect sequences of IP IDs.

### 4.3.3 Distribution of alias resolution methods

Figure 4.5 shows the distribution of the alias resolution methods that led to the discovered alias pairs/lists, reusing the same format and data as the previous figures. Again, the stacked bars have a similar shape to previous figures, but a closer look shows that there were actually clear differences: for instance, in Figure 4.5, 50% of the alias pairs/lists found in AS13789 (12; Internap Holding LLC) were discovered with the `Ally`-like approach, while Fig. 4.3b shows more than 60% of the fingerprints

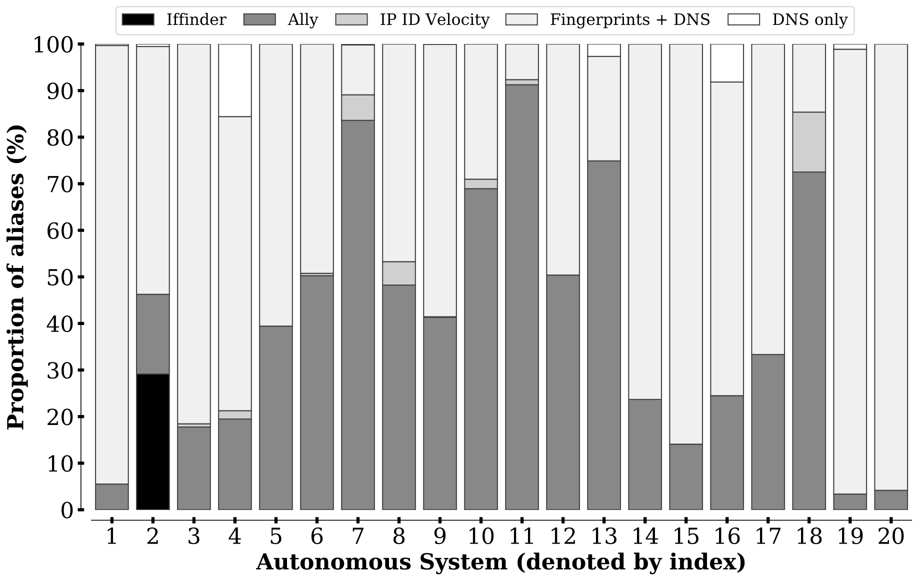


Figure 4.5: Distribution of alias resolution methods used to discover alias pairs (April 14th, 2017).

found in this AS featured the *healthy* counter class, suggesting not all candidates associated with these fingerprints could be aliased. Furthermore, several distributions only included small amounts of alias pairs/lists resulting from the DNS heuristic only: this corresponds to alias candidates that could not be properly fingerprinted due to unresponsivity to `echo-request` probes, and for which DNS records could be collected (for reminders, reverse DNS does not require direct probing, cf. Sec. 2.3.5). As a consequence, the framework could only attempt alias resolution with the DNS heuristic, resulting in some additional alias pairs/lists.

Overall, the methods based on IP IDs and the aliasing of candidates with similar fingerprints and comparable DNS records were the most commonly used methods, with one notable exception: in the case of AS10010 (2; TOKAI Communications), almost 30% of the alias pairs/lists could be inferred via the `iffinder`-like method (cf. Sec. 3.3.1.1). The same method was however completely unused in all other cases. This does not mean the `iffinder`-like method was only suitable to AS10010: indeed, there is a fair chance that UDP probes were filtered on their way from the PlanetLab testbed to the target networks, completely preventing the use of such a method. Interestingly, almost all the PlanetLab nodes used in 2017 were international nodes [8], with the exception of one European node [9] used to measure AS10010. From 2019, more European nodes could be used to conduct measurements of various kinds, resulting in more alias pairs/lists being discovered with the `iffinder`-like method in all kinds of networks. The same observation was possible with measurements performed from the EdgeNet cluster, which replaced PlanetLab starting from mid-2020 [99, 108]. All in all, these observations suggest that mimicking `iffinder` [71] remains a credible option for alias resolution. In fact, research carried out later than the work presented in this chapter still involved `iffinder` [87].

[8]denoted at the time as PLC, for PlanetLab Central. They became more and more difficult to use over time.

[9]denoted as PLE, for PlanetLab Europe. Most PLE nodes were difficult to use before a large update in 2019.

The main conclusion to draw from Figure 4.5 is that using a single alias resolution method has become highly unrealistic. As already discussed in the previous paragraph, `iffinder` (cf. Sec. 2.3.1) should only be used to complement other methods as it cannot be self-sufficient. Moreover, methods based on IP IDs, which were frequently researched up to the early 2010's (cf. Sec. 2.3.2 and Sec. 2.3.3), can now only detect a small amount of alias pairs/lists within specific target networks. This not only suggests combining multiple methods is nowadays mandatory for exhaustive alias resolution, but this also highlights the importance of space search reduction. For instance, as the DNS heuristic (cf. Sec. 3.3.1.4) has a limited accuracy, it is crucial to break down the initial set of alias candidates as much as possible to minimize both false positives and false negatives.

### 4.3.4 Compliance with the ICMP timestamp request

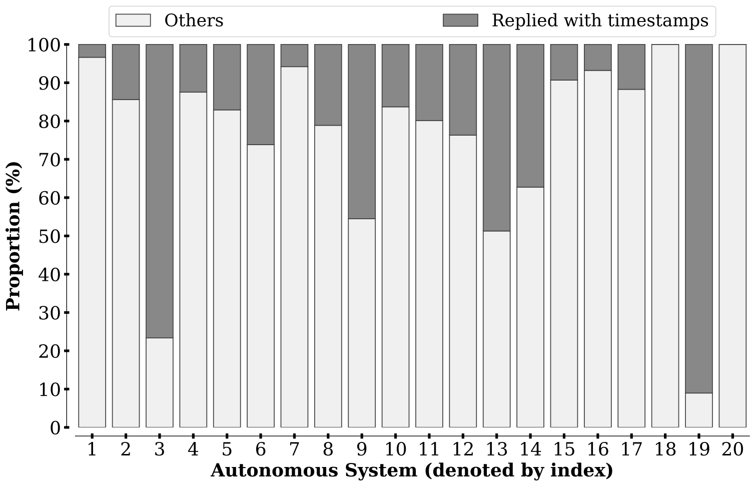


Figure 4.6: Distribution of alias candidates replying to ICMP timestamp request (April 14th, 2017).

Figure 4.6 shows the distribution of successfully fingerprinted alias candidates replying to the ICMP timestamp request, with the same format and data as in previous sections. As discussed in Sec. 3.1.2, the inclusion of this feature in the fingerprints was purely exploratory: the objective was to see if there were correlations between alias pairs/lists and compliance to the mechanism, or even correlations with other fingerprint components. The main finding from this exploration was that not all router interfaces replied to such requests, and that the distribution greatly changed from one AS to another. The mechanism was not implemented at all in the cases of AS50673 (18; Serverius Holding B.V.) and AS802 (20; University of York). However, it did not appear to correlate with other fingerprint components. Moreover, among well established alias pairs/lists (notably found within the ULiège network), not all aliased interfaces were displaying the same behaviour regarding this mechanism. Overall, there was no conclusive evidence the ICMP timestamp request could be used to validate or invalidate potential aliases. Therefore, the investigation of this feature was not continued in the context of this research work, though it could be further tested in the future to determine whether it

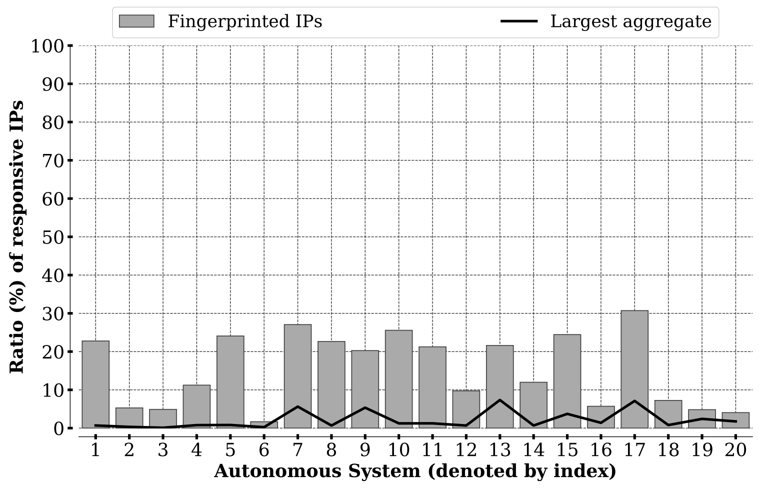


Figure 4.7: Effects of the space search reduction induced by `TreeNET` (April 14th, 2017).

could help in the same manner as `iffinder`, i.e., to build small alias lists to be completed with other methods (as suggested in Sec. 3.3.1.1).

## 4.3.5 Effects of space search reduction

Finally, the effects of the space search reduction resulting from the tree-like topology mapping of `TreeNET` were examined. Figure 4.7 illustrates the proportion of successfully fingerprinted alias candidates with respect to the total number of responsive IP interfaces for each target AS as a bar chart. The black curve illustrates the largest list of alias candidates that were considered simultaneously. Once again, the data is the same as in previous figures. Regardless of the target AS, there was a drastic reduction in the interfaces to consider for alias resolution with respect to the responsive (or *live*) IP interfaces: at worst, around 30% of the *live* interfaces were fingerprinted, and at best, significantly less than 10%. This result does not include the fact that the fingerprinted alias candidates were themselves split into even smaller sets before being put through a suitable alias resolution method.

However, Figure 4.7 also shows that the method for space search reduction is crucial, as it shows that, in some ASes, there were very large aggregates of alias candidates to consider together. This was especially apparent for AS14 (13; Columbia University), where roughly one third of the fingerprinted candidates was found in the same alias resolution scenario, which seemed large even considering the total number of *live* interfaces.

In fact, the large aggregates of alias candidates suggested by Figure 4.7 often coincided with the multi-label nodes used by `TreeNET` to account for the parts of the topology that feature multiple paths (cf. Sec. 4.1.1). Due in particular to load balancing, some large neighborhoods located closer to the vantage point had to be superposed through the mechanism of multi-label nodes, which eventually resulted in large lists of alias candidates. In theory, alias resolution can reveal routers within these scenarios, some of them accounting for convergence points in load balancing architectures [50]. In practice, however, and because multi-label nodes fuse alias resolution scenarios instead of clearly distinguishing them, they reduce the benefits of the tree-like mapping regarding space search reduction. As a result, `TreeNET` perfectly handles neighborhoods appearing at the ends of a tree or in networks with little traffic engineering, but barely reduces the space search when load balancing architectures are involved.

Because of asymmetrical paths, load balancers can also induce inconsistencies in the distances observed between the vantage point and a remote location. For example, subnet interfaces that are around the same real-life devices can be found at the end of route paths with slightly different lengths. This has two consequences: first, as will be discussed in Chapter 5, subnets can be poorly identified and split into several chunks, and second, subnets from the same location can be found at the end of different branches of the tree-like map. This results in some neighborhoods being divided into several individual neighborhoods, resulting in overly reduced alias resolution scenarios. A similar issue was witnessed on the ULiège network while validating `TreeNET` (cf. Sec. 4.2.2), though it was caused by network redirections rather than load balancing. Figure 4.8 shows, using the same data and format as before, the ratio of aliased candidates versus the ratio of non-aliased candidates for each measured AS. While a majority of IP interfaces could be aliased in most cases, there were also large amounts of interfaces that could not be aliased at all. Although it would be perfectly reasonable to discover that an interface is not an alias of any other identified interface, the large number of unaliased interfaces in some instances, such as in the case of AS5511 (6; Orange S.A.), was also the result of overly reduced alias resolution scenarii.

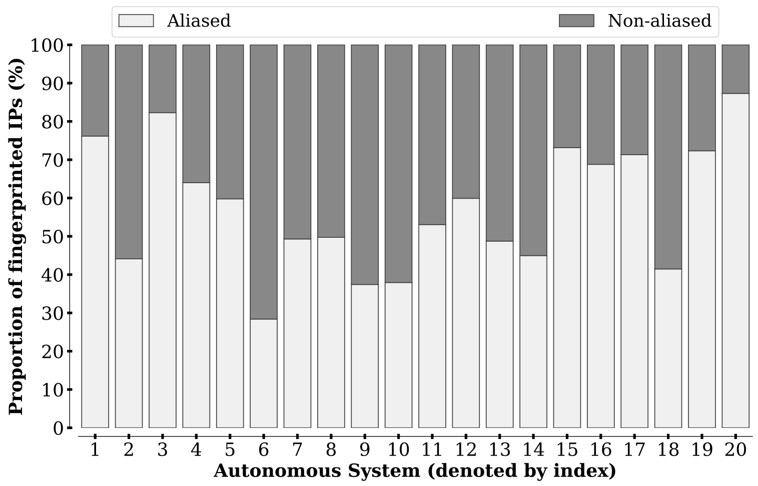


Figure 4.8: Proportions of alias candidates being successfully aliased (April 14th, 2017).

Two main conclusions can be drawn from Figures 4.7 and 4.8: first, space search reduction is a powerful tool, and second, its accuracy is critical to achieve thorough alias resolution. Having to concurrently consider large amounts of alias candidates requires thorough probing scheme and resolution, such as Internet-scale solutions like `MIDAR` [73], but creating such techniques for every possible alias resolution method is ultimately cumbersome. Reciprocally, a space search reduction scheme that (accidentally) reduces the space search too much can also result in poor alias resolution.

## 4.4 Closing comments on the framework and perspectives

This first part of the thesis provides the three following key contributions.

1. It provides a new application of IP fingerprinting [121] by designing a new format of fingerprints (cf. Sec. 3.1.2) used to detect which alias resolution methods are suited to a set of alias candidates. It also provides a form of space search reduction, i.e., alias candidates with completely different fingerprints cannot be an alias of each other. This also constitutes a first systematic methodology for combining several alias resolution methods (cf. Sec. 3.3.2), a research direction which has already been put forward by previous research [117] and further explored by posterior research [87]. Observations in the wild conducted from the PlanetLab testbed further showed individual methods are no longer self-sufficient (cf. Sec. 4.3.3).

2. By carefully scheduling the probing work (cf. Sec. 3.2) used by the framework and combining it with space search reduction (cf. Sec. 4.1.1), it has been demonstrated on a groundtruth network that the overall methodology features both an accuracy rivaling the state-of-the-art and a very cheap probing cost, much lower than that of other state-of-the-art tools (cf. Sec. 4.2.2).

3. With the help of fingerprints collected in the wild from the PlanetLab testbed, it was also proved that specific values of fingerprints tend to correlate (cf. Sec. 4.3.2). This suggests there are several profiles of routers in the Internet, and some of them might be linked to equipment vendors, much like the TTL-based router signatures of Vanaubel et al. [121].

As long as it is combined with preliminary space search reduction to get simple alias resolution scenarios, the framework can be both accurate and cheap when it comes to probing, as noted by contribution 2. Moreover, as contribution 1 showed, envisioning such a methodology becomes even more necessary: indeed, state-of-the-art methods are increasingly less self-sufficient, but combining them together can compensate for the shortcomings of each.

In fact, the combination of space search reduction and fingerprinting should be envisioned as a general approach to alias resolution and not as a new *ad hoc* solution that takes advantage of the specificities of network protocols. Such a method can only work for a time, in the same way as the methods based on the IP protocol timestamp option (cf. Sec. 2.3.4). On the contrary, as long as an implementation provides accurate individual alias resolution heuristics, the whole methodology could easily be re-used in future research, especially given how some fingerprint values tend to correlate, suggesting certain brands of equipment have specific properties to take into account upon performing alias resolution (as noted by contribution 3). While developed only for IPv4 in the context of this thesis, the proposed framework could also be adapted to the IPv6 protocol as well.

Because it was designed with medium- and small-sized alias resolution scenarios in mind (cf. Chapter 3), the framework will work best if involved in a holistic approach to topology discovery (see Sec. 1.1.1). In particular, neighborhood inference can constitute a powerful space search reduction scheme, as it can break down the entire set of alias candidates (e.g., all router interfaces found in `traceroute` data) into small scenarios. `TreeNET` [55] (cf. Sec. 4.1.1) provided such a scheme to test the framework, and led to the validation results highlighted by contribution 2. Not only the framework can benefit from hop-level topology discovery (as defined in Sec. 1.1.2), but it could also prove useful to this very task. As discussed in Sec. 2.4, alias resolution could also be used to decipher load balanced paths, which is a crucial challenge to achieve accurate hop-level mapping. `TreeNET` itself can be easily impaired by load balancing, resulting in inaccurate neighborhood discovery on occasions, as discussed in Sec. 4.3.5. In the third part of this thesis (see also Sec. 1.4), the framework presented in this first part will be used to detect convergence points in hop-level topologies (cf. Chapter 11) in order to achieve a more accurate neighborhood inference, which can benefit in turn to the framework to perform exhaustive alias resolution.

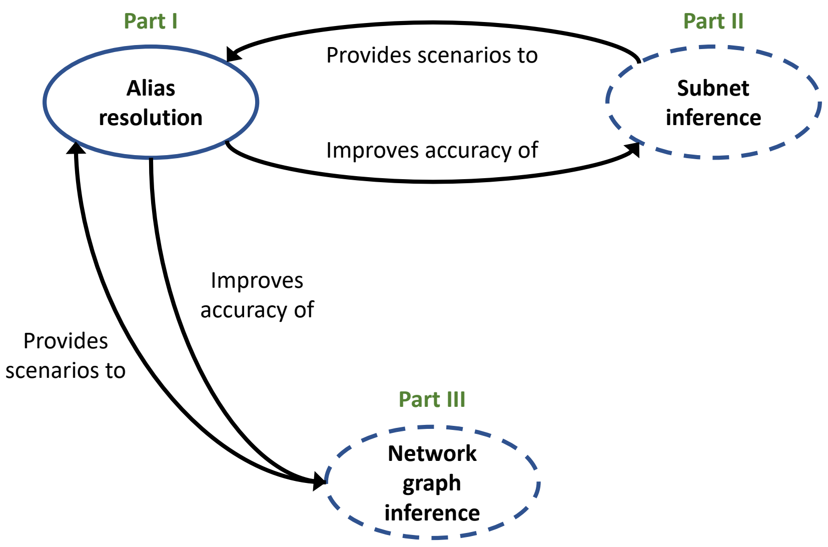


Figure 4.9: Interactions of the alias resolution framework in a holistic approach.

Not only the framework can benefit from network graph inference and vice versa, but it can also interact with subnet inference (as already hinted in Sec. 1.1.1). Indeed, as will be discussed in the second part of this thesis (notably in Chapter 6 and Chapter 7), a key challenge in subnet inference lies in the identification of the last router crossed before reaching a subnet (a.k.a. ingress router, as defined in Sec. 1.1.1 and in Chapter 5), and this router does not always appear as a single router interface in `traceroute` data as a consequence of load balancing. The framework can therefore be used to solve ambiguous subnet inference scenarios, therefore improving subnet inference itself. Figure 4.9 summarizes the interactions of the alias resolution framework with other topology discovery challenges explored by this thesis.

Finally, it should be noted that most of the research presented in this first part of this thesis was conducted during 2016 and 2017, and that implementations of the framework have only received minor improvements since then (cf. Sec. 4.1.2). Any new research reusing ideas from the fingerprint-based framework should therefore consider:

- whether it should refresh and/or expand the typical values of the fingerprints (cf. Sec. 3.1.2),
- whether there are other unexplored properties to investigate either for alias resolution itself or for space search reduction,
- including research that was conducted after this work (e.g., automatically creating regexes for reverse DNS with machine learning [80]).

These potential improvements have been left for future research.

# Part II

# Subnet Inference

# Chapter 5

# Introduction to Subnet Inference

This chapter presents the terminology and the state-of-the-art of subnet inference. First, Sec. 5.1 introduces the definitions and motivations relevant to subnets (short for *subnetworks*) and their discovery. Sec. 5.2 then reviews the state-of-the-art of subnet inference. Sec. 5.3 concludes this first chapter by summarizing the typical limits of state-of-the-art solutions.

## 5.1 Terminology

### 5.1.1 Main definitions and motivations

In a computer network, a ***subnet*** (short for *subnetwork*) is a set of devices, each identified by a unique IP address, that are connected together through the same connection medium and that can communicate directly with each others at the link layer [92]. As a result, ***subnet inference*** denotes the systematic discovery of subnets within a target network. In the Internet, a subnet can be a point-to-point link between two adjacent routers as well as a LAN (Local Area Network) within the topology of a network, e.g., a group of end systems, such as servers.

Typically, a subnet is denoted by the CIDR (Classless Inter-Domain Routing) notation [43] traditionally used to represent contiguous blocks of addresses managed by network operators. As a reminder, the CIDR notation for IPv4 is `a.b.c.d/e` where `a.b.c.d` is the *prefix address* and `e` the length of this prefix. The ***subnet prefix*** is therefore the prefix address of a given subnet and contains a fixed number of bits, starting from the most significant bit, that are constant for all IP addresses found within the subnet, with the length `e` denoting how many bits are constant. For example, `139.165.223.0/24` denotes a subnet within the ULiège network (within the `139.165.0.0/16` prefix) managed by the RUN [1] that hosts end systems and the work stations allocated to RUN members. For

[1] Research Unit in Networking, the computer networking research group of the Montefiore Institute

this subnet, the prefix length is equal to 24: this means the 3 first bytes (24 bits), i.e., `139.165.223.`, are identical for any IP address on the subnet (e.g.: `139.165.223.1`, `139.165.223.30`, etc.).

Though the field of topology discovery has historically focused on the interface-level, router-level (cf. Sec. 2.1) and AS-level [37, 61], notably with the development of alias resolution (cf. Chapter 2), the research community has been exploring alternative approaches since the late 2000's. This general effort encompassed the study and discovery of Internet eXchange Points (or IXPs) [13, 95], Points-of-Presence (or PoPs) [41, 109], and of course, subnets [58, 70, 116, 118]. Initially, the subnet-level was explored as a complementary view to the router-level: the objective was to characterize links better at the network layer to annotate or complete existing maps [58]. Subnet inference has also been investigated to highlight unique topological features of ISP networks [118]. As already mentioned in Sec. 4.1, studying the subnet-level can also act as a form of space search reduction for alias resolution [51, 55]. Not only the subnet-level can help discover the router-level, but it also opens new opportunities for network modeling, as will be explained in detail in Chapter 13.

### 5.1.2 Practical definitions

To describe the state-of-the-art of subnet inference as well as new subnet inference techniques introduced in subsequent chapters more easily, a few practical terms are hereby introduced. First of all, the ***TTL distance*** is the minimal TTL value a tool running at a given vantage point can use in a probe packet to get a reply (e.g., an `echo-reply` ICMP message) from a given target interface. For example, a TTL distance of 10 means that there are 10 hops between the target interface and the vantage point. In this scenario, using a TTL value one unit smaller in a probe packet would result in a `time-exceeded` ICMP message emitted by the router located just before the target interface.

A measurement-based definition of a ***subnet*** should also be provided. From the perspective of a single vantage point, and in the (quasi) absence of traffic engineering (e.g., load balancing), the interfaces of one subnet will appear as a set of IP addresses that are both consecutive with respect to the IP address space and (ideally) located at the same TTL distance. Moreover, they should be reached through similar (if not identical) routes: as a result, the last interface(s) appearing in the routes towards distinct subnet interfaces of the same subnet should be identical.

Ideally, a subnet should also contain at least one interface belonging to the last router crossed before entering the subnet, therefore observed one hop closer to the measurement vantage point than other subnet interfaces. The last router crossed before entering the subnet can be referred to as the ***ingress router***. Any subnet interface that is not located on the ingress router is called a ***pivot interface*** (occasionally shortened to *pivot*). The immediate consequence of this definition is that all pivot interfaces of a subnet should be observed at the same TTL distance. The (presumably) single interface located on the ingress router is called a ***contra-pivot interface*** (occasionally shortened to *contra-pivot*). In practice, routers can have back-up interfaces for critical subnets, meaning a subnet with several contra-pivot interfaces is possible. [2] It is important to note that, measurement-wise, the notions of (contra-)pivot interface depend on comparison: a contra-pivot interface usually stands

[2]Such subnets were observed on the ULiège network and verified with the friendly help of the SeGI.

out in a subnet because its observed properties differ from those of the pivot interface(s) (e.g., it has a shorter TTL distance). This implies that, if only one interface within a given subnet is reponsive to probes, it becomes difficult to elucidate whether or not it is located on the ingress router, and therefore, what kind of subnet interface it is.

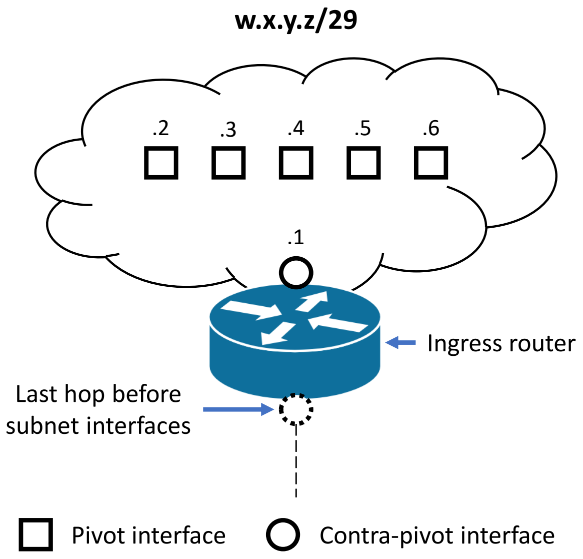


Figure 5.1: Definition of a subnet from the perspective of a measurement.

Figure 5.1 depicts the measurement-based definition of a subnet, completed with the concepts of: ingress router, pivot interface and contra-pivot interface. In this figure, the squares depict the pivot interfaces, i.e. any subnet interface not located on the ingress router, while the circles represent router interfaces. The plain-line circle, in particular, stands for the contra-pivot interface of the depicted toy subnet. Note that the entire terminology is common to the subnet inference tools `TraceNET` [116], `ExploreNET` [118] and `TreeNET` [55], and will be re-used as is in subsequent chapters.

## 5.2 State-of-the-art subnet inference solutions

### 5.2.1 `traceroute`-based subnet inference (`TraceNET`)

In 2007, M. H. Gunes and K. Sarac presented an early subnet inference methodology that processed `traceroute` paths to identify subnets [58]. Their methodology relied on IP assignment practices, i.e., the fact that subnet interfaces from the same subnet will feature the same first bits, also known as the same subnet prefix. In particular, they used an iterative method to enumerate all potential subnet prefixes and subsequently used several heuristics to exclude potential subnets which could not be considered as accurate. Among others, their approach excluded any subnet that encompassed two IP addresses that had a predecessor/successor relationship in a `traceroute` path, and also excluded hypothetical subnets where the TTL distances were inconsistent. One of their heuristics also excluded small subnets when larger subnets encompassing them were possible, i.e., not filtered out by the previous heuristics. At the time, Gunes' and Sarac's methodology provided promising results, as more than half of their inferred subnets exactly matched a groundtruth.

In 2010, M. Engin Tozal and K. Sarac expanded this early approach with the subnet inference tool `TraceNET` [116]. As its name implies, `TraceNET` is an elaborated variant of `traceroute` that not only discovers route paths but also discovers the subnets crossed between each hop. To do so, for each IP address discovered with the usual `traceroute`, `TraceNET` builds a /31 subnet encompassing it and probes the other IP address encompassed by this early prefix. If the corresponding interface is responsive, it applies a series of heuristics to guarantee the subnet consistency. These heuristics (nine in total) are all based on IP assignment practices and typical router behaviour observed at the time. If the resulting subnet is sound, it is expanded by decreasing its prefix length by 1, i.e., it becomes a /30 subnet. `TraceNET` further probes the IP addresses newly encompassed by the expanded prefix and applies its heuristics for the second time to ensure the subnet remains consistent, then expands again, and so on. Whenever an heuristic fails, the subnet is shrunk by incrementing its prefix length once and `TraceNET` moves on to other potential subnets. Figure 5.2 illustrates the typical data discovered by `TraceNET`, based on a figure from the paper that introduced it [116]. In this figure, arrows show the analyzed route path while grey circles correspond to interfaces usually discovered with `traceroute`.

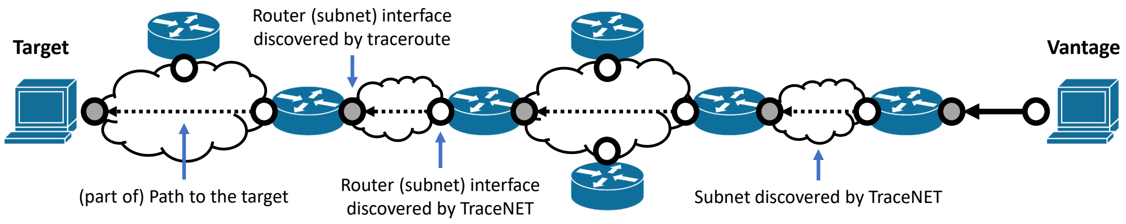

Target
Router (subnet) interface discovered by traceroute
Vantage
(part of) Path to the target
Router (subnet) interface discovered by TraceNET
Subnet discovered by TraceNET

Figure 5.2: Topological data as discovered by `TraceNET`.

In addition to providing a more thorough subnet inference scheme, `TraceNET` is also the first subnet inference work to rely on the concepts of pivot interface, contra-pivot interface and ingress router previously defined in Sec. 5.1.2. It should be noted, though, that its definition of pivot interface specifically denoted the first IP address of a growing subnet while this thesis broadens the definition as any subnet interface not located on the ingress router. Tozal and Sarac however noted that `TraceNET` was not well suited for situations featuring symmetric or asymmetric paths induced by load balancers [14], in addition to underestimating large subnets (e.g., /24). To mitigate the former issue, `TraceNET` only used ICMP probes, as ICMP probes were discovered to be less subject to load balancing at the time [81]. Another issue with `TraceNET` is that routing dynamics in the Internet tend to use the same sub-paths, meaning a lot of paths (such as back-up links) will be missed while collecting `traceroute` paths. As a consequence, `TraceNET` cannot discover the subnet-level exhaustively. Such an issue has already been noticed with research work involving *RocketFuel* [113] and *AROMA* [74], two topology discovery tools previously mentioned in Chapter 2.

### 5.2.2 An active probing method: `ExploreNET`

To make up for `TraceNET`'s inability to discover subnets exhaustively, their authors introduced an alternative in 2011: `ExploreNET` [118]. The methodology of `ExploreNET` consists of actively probing a given target IP address in order to discover the subnet that best accommodates it. In the process, `ExploreNET` will, of course, probe the IP addresses that are closest to the initial target with respect to the IPv4 address space. In fact, its methodology is very similar to that of `TraceNET`: `ExploreNET` starts by building a /31 subnet that includes the target IP address (if responsive), then probes the second IP interface of this prefix and determines whether or not both IP addresses form a consistent subnet with the help of a custom set of heuristics. If this first subnet is sound, it will be expanded by decrementing the prefix length. For each new prefix length, `ExploreNET` probes the newly encompassed IP addresses to confirm they satisfy the heuristics to guarantee the new subnet remains consistent. The heuristics however differ from `TraceNET`, as they mostly ensure that the probes reaching the subnet interfaces are passing through the same ingress router, and not through another router observed at the same distance, which may be the ingress router to another subnet. When a freshly expanded subnet no longer satisfies all heuristics, it is shrunk by incrementing its prefix length once and kept as is.

`ExploreNET` provides a number of benefits with respect to `TraceNET`: first, it can discover subnets corresponding to links that are difficult to observe with `traceroute` probing, and second, it can discover subnets regardless of the existence of symmetric paths induced by load balancing architectures. Indeed, `ExploreNET` only requires the probes to continually reach the same ingress router to consistently discover subnets, meaning there can be some load balancing on the way. The design of `ExploreNET` allowed Tozal and Sarac to study more exhaustively the subnet-level of various ISPs. In particular, they highlighted that the typical distribution of subnet prefix length tends to follow a power law shape, i.e., small subnets (/30 and /31) are the most common in the Internet, but a few large subnets (/20, /21, etc.) could also be discovered within specific ISP networks. Interestingly, they also discovered that the /24 prefix is far more common than comparable prefix lengths (e.g., /23 and /25), which they explained by the fact that /24 is a popular choice for creating subnets [118].

Unfortunately, `ExploreNET` also has limitations. For one thing, it has no built-in mechanism to mitigate the effects of asymmetric paths in load balancing architectures. Typically, these paths can induce inconsistencies in the observed TTL distances for subnet interfaces found in the same subnet, therefore preventing consistent inference by either `TraceNET` either `ExploreNET`. Moreover, the open source implementation of `ExploreNET` [3] puts a condition on growing a subnet: at least half of the attributable subnet IP addresses must be populated with responsive IP addresses before expanding it, with the exception of small subnets (i.e., /29, /30, and /31). Having to satisfy such a threshold is sound, as it is preferable to avoid inferring overgrown subnets. In practice, this threshold prevents correctly inferring subnets when their responsive IP addresses are either few, or not spread uniformly in their address space. For example, it is fairly possible a large subnet (e.g., a /24) features

[3] available at `http://nsrg.louisiana.edu/project/ntmaps/output/explorenet.html`

a block of contiguous IP addresses in the third quarter of its address space, but no responsive IP address elsewhere with the exception of one contra-pivot interface found in the beginning of the same space [4]. In such a scenario, `ExploreNET` will partition the actual subnet into several chunks, resulting in inaccurate subnet inference.

### 5.2.3 Subnet refinement with `TreeNET`

`TreeNET` [55] (briefly introduced in Sec. 4.1) implements an heuristic to mitigate `ExploreNET`'s partitioning issue. As already explained, `TreeNET` does not implement its own subnet inference scheme but instead builds itself upon `ExploreNET`, as the initial focus of `TreeNET` is subnet-based topology mapping. The underlying idea of the correction heuristic is simple: if a subnet only features pivot interfaces, it is probably incomplete, as any subnet should contain at least one interface found on the ingress router. The heuristic therefore consists of selecting an inferred subnet without a contra-pivot interface and expanding it repeatedly until it overlaps other inferred subnets that could contain such an interface. Contra-pivot interfaces are identified by the fact that they are located exactly one hop closer to the vantage point than pivot interfaces. If an overlapped subnet contains a sound contra-pivot interface, all overlapped subnets are merged under a new subnet prefix. Of course, this correction heuristic puts several conditions before the merger to guarantee sound results: for instance, if the expanded subnet overlaps too many subnets whose interfaces would become outliers (i.e., TTL distances become inconsistent), then the heuristic stops. This is also the case if the resulting merged subnet features too many contra-pivot interfaces – `TreeNET` allowing more than one, as multiple contra-pivot interfaces are possible in practice (cf. Sec. 5.1.2).

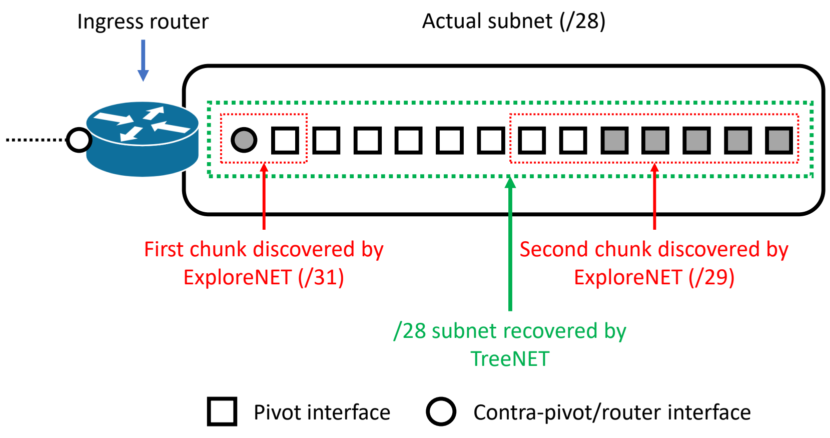

Ingress router
Actual subnet (/28)
First chunk discovered by ExploreNET (/31)
Second chunk discovered by ExploreNET (/29)
/28 subnet recovered by TreeNET
Pivot interface
Contra-pivot/router interface

Figure 5.3: A toy case of subnet partitioning. Grey squares/bullet are responsive interfaces.

Figure 5.3 depicts a toy case of subnet partitioning. An actual /28 subnet, containing up to 14 attributable addresses ($2^4 - 2$ because the subnet prefix and broadcast address should not be used), is first discovered as a /31 and a /29 by `ExploreNET`, the former containing the contra-pivot interface

[4]Several confirmed /24 subnets of the ULiège network are like this.

while the latter only contains pivot interfaces. This is due to the responsive IP addresses not being uniformly spread, as pivot interfaces are only visible in the second half of the subnet. By expanding the discovered /29 and correctly identifying the /31 as a contra-pivot interface, `TreeNET` can recover the full /28 subnet. As will be discussed in Sec. 8.1, this heuristic considerably improves the inference of larger subnets and allow `TreeNET` to be more representative of the subnet-level of a target domain overall, though it can also produce overgrown subnets (especially when TTL distances vary).

### 5.2.4 Subnet inference with multiple vantage points (*Cheleby*)

While previous state-of-the-art solutions are typically run from a single vantage point, it should be noted that subnet inference involving multiple vantage points has been explored too. In early 2012, H. Kardes et al. introduced the *Cheleby* system [70], which runs Paris `traceroute` [11] towards target networks from hundreds of vantage points and processes the collected paths in order to reliably discover subnets. The underlying idea is the same as before: a subnet can be recognized as a set of IP addresses, consecutive with respect to the IPv4 address space, where all interfaces (except contra-pivot interfaces) are observed at the same TTL distance. By combining multiple vantage points, it becomes possible to verify whether a sequence is consistently observed across several measurements to guarantee a discovered subnet is not a false positive (i.e., a subnet that does not match the real topology) resulting from occasionnal coincidences in the observed TTL distances.

The *Cheleby* system is an ambitious but expensive solution to deploy. In their paper, Kardes et al. mention having used around 500 PlanetLab nodes out of 600 available nodes (from a total of 1,100 advertised nodes), the difference being explained by the fact that around 100 of the 600 usable nodes did not produce satisfying measurements. Such a large scale deployment is therefore difficult to achieve in practice, with the main benefit being the mitigation of the consequences of traffic engineering, thanks to the multiple vantage points. Designing subnet inference methods that can address these challenges from a single vantage point can result in much simpler deployments.

## 5.3 Limits of state-of-the-art solutions

Besides the *Cheleby* system (cf. Sec. 5.2.4), that sets itself apart from other solutions by using many vantage points, most state-of-the-art tools can achieve reliable subnet inference from a single vantage point in different contexts. Moreover, measurements in the wild with these tools allowed the research community to have a first glance at typical properties of the subnet-level, such as the power law shape of the distribution of the prefix length (with the notable exception of /24 subnets) [118].

However, `TraceNET`, `ExploreNET`, and `TreeNET` all suffer from the same problem: all three rely on strong hypotheses regarding what a subnet will look like from the perspective of a single vantage point. Indeed, all three require constant TTL distances and a consistent ingress router for a given subnet to be inferred accurately. Both properties can be easily thwarted by load balancing (a common form of traffic engineering): for instance, asymmetrical paths in load balancers [14] will typically induce variations in the observed TTL distances, preventing the correct inference of specific subnets.

In addition to the problems induced by load balancing, each tool also comes with its own issues: for instance, `TraceNET` cannot exhaustively map the subnet-level due to routing dynamics. However, they also share another major flaw: all three rely heavily on the IPv4 scope. It was previously mentioned that IPv4 subnets are very small when compared to IPv6 subnets: in IPv6, it is not uncommon to use /64 prefixes (that can even be used as point-to-point links), each such a prefix providing up to $2^{64}$ possible addresses. Subnet inference as performed by `TraceNET` and `ExploreNET`, on the other hand, exhaustively explores the address space of subnets that can at most contain $2^{12} - 2 = 4,094$ attributable addresses (corresponding to a fully populated /20 subnet [5]). While testing a few thousands IP addresses remains computationally reasonable, doing the same task with billions of potential IP addresses per subnet would be prohibitively complicated.

## 5.4 Research questions

The second part of this thesis aims to provide a new subnet inference solution for IPv4 that can overcome the limits of the previously discussed tools, mainly `TraceNET` (cf. Sec. 5.2.1), `ExploreNET` (cf. Sec. 5.2.2) and `TreeNET` (cf. Sec. 5.2.3). Not only will this second part address the challenges induced by load balancing discussed in Sec. 5.3, but it will also describe a methodology that was designed with efficiency in mind.

To achieve both aforementioned goals, this part will center around the following research questions, with each subsequent chapter answering specific questions.

1. **Is it possible to precisely define the consequences of traffic engineering for subnet inference ? How frequent are they ?** Before designing a subnet inference methodology that will remain accurate despite load balancing, the induced phenomena first have to be characterized to identify them during subnet inference and take their effects into account. Moreover, quantifying them also will show the extent to which state-of-the-art tools are outdated and further justify the need for a novel approach. These questions will be answered in Chapter 6.

2. **Can a systematic subnet inference methodology that can live with traffic engineering be designed ? Can it be in linear time ?** Once the effects of traffic engineering have been properly identified, it should be possible to design algorithms that can discover subnets even when these consequences change how subnets appear to probes. Chapter 7 will present a methodology to achieve this. It will also shortly discuss the time complexity of each algorithm to show all `WISE` algorithms scale linearly with their respective input. The detailed time complexity analyses can be found in the Appendix A (Sec. A.2).

[5] /19 subnets (or larger) have not been observed with previous research nor the research presented in this thesis.

3. **How accurate is this methodology ? Can it outperform state-of-the-art tools ?** The subnet inference solution presented in Chapter 7 will also be thoroughly assessed on two groundtruth networks in Chapter 8. Through the process, the new tool will also be compared with the state-of-the-art in terms of accuracy (i.e., how true the discovered subnets are) and performance (i.e., how fast the tools can run).

4. **Do changes in effects of traffic engineering affect the discovered subnets ? To what extent ?** The ability of the new tool to consistently discover the same subnets despite variations in terms of traffic engineering should also be assessed. Chapter 8 will also cover these questions by introducing some custom metrics and measuring them with large amounts of data collected from the PlanetLab testbed.

5. **Can the new methodology be adapted to IPv6 ?** Finally, this part of the thesis will also briefly discuss whether the new methodology is only bound to work with IPv4 (like previous solutions, cf. Sec. 5.3) or if it can be adapted to IPv6. Answers to this question will be provided at the end of Chapter 7.

# Chapter 6

# Subnet Inference Challenges

This chapter discusses in detail the modern challenges of subnet inference, thereby answering research question 1 from Sec. 5.4. To recap, this research question consists of learning whether it is possible to precisely define and quantify the typical consequences of traffic engineering regarding subnet inference. Sec. 6.1 defines the phenomena induced by traffic engineering or specific network equipment that can impair state-of-the-art subnet inference methodologies. Sec. 6.2 presents a quantification of these phenomena in the wild, using data collected from the PlanetLab testbed. Sec. 6.3 concludes this chapter by summarizing how a modern subnet inference tool should handle the discussed challenges.

## 6.1 Characterization of the challenges

This section defines and characterizes several routing phenomena that can thwart the traditional hypotheses of subnet inference (as reviewed in Chapter 5). Sec. 6.1.1 first precisely defines the notion of *trail*, a concept used to broadly denote the last hop(s) prior to a subnet. Then, Sec. 6.1.2 and Sec. 6.1.3 respectively define *warping* and *flickering* trails, both being consequences of traffic engineering, and Sec. 6.1.4 describes *echoing* trails, which result from specific equipment.

### 6.1.1 A broader definition of the last hop: the *trail*

As previously discussed in Sec. 5.1, one of the key elements that can be used to infer subnets is to ensure that a set of interfaces (consecutive with respect to the IPv4 scope) are reached through the same ingress router. In particular, `ExploreNET` implements various heuristics to guarantee that the ingress router of a newly discovered interface is the same as it was for previously probed subnet interfaces, which guarantees that all interfaces are on the same subnet (cf. Sec. 5.2.2). An ingress router can be notably identified by a router interface discovered through `traceroute` probing.

However, the last hop observed prior to a set of subnet interfaces is not always visible: some routers are indeed configured not to reply to `traceroute` probes, and therefore appear as ***anonymous hops*** [59, 68, 125] in the paths, i.e., hops without a well identified IP interface (typically denoted by the wildcard symbol `*` in the output of the classical `traceroute` [119]). Moreover, the last hop can also be a ***cycling hop***, i.e., the router interface identified at this hop has already been identified at previous hop counts in the route. Cycling hops can either follow each other in a path or appear several hops away from each other, and should therefore be handled with caution [85]. Cycling hops can be the result of a misconfiguration as well as a consequence of temporary network failures.

In order to simplify the terminology used in subsequent sections and chapters, the last non-cycling and non-anonymous route hop observed prior to the target IP address of a `traceroute` measurement will be defined as a ***trail***. When this well identified hop is one or several hops away from the target IP address (e.g., as a result of intermediate anonymous hops), an integer value is also associated with the trail to count the number of intermediate problematic hops, also called ***anomalies***. A trail is said to be ***anomaly-free*** if there is no intermediate hop between the trail and the target IP address. An anomaly-free trail is typically denoted only by the associated IP address, while a trail with anomalies is denoted by the concatenation of the associated IP address with the anomaly count. E.g., `10.0.2.128` denotes an anomaly-free trail while `10.0.2.128 | 1` denotes a trail with one problematic hop (e.g., anonymous) observed before reaching the target IP address.

In addition to simplifying the terminology, the concept of trail also constitutes a best effort strategy for subnet inference in itself: IP addresses that are consecutive in the IPv4 scope and whose last hops are not visible to `traceroute` probes can still be approximated as subnets when they exhibit the same trail. In this particular scenario, the identical trails amount to a common end route shared by all subnet interfaces, which may not strictly identify a same ingress router but nevertheless suggests these interfaces are found at the same place. A subnet built on the basis of such trails can therefore be overgrown, but eventually interpreted as an IPv4 prefix delimiting actual subnets.

Listing 6.1: Subnet interfaces with problematic trails

```
10.0.7.1 - 4 [10.0.2.128]
10.0.7.3 - 5 [10.0.2.128 | 1]
10.0.7.4 - 5 [10.0.2.128 | 1]
10.0.7.6 - 5 [10.0.2.128 | 1]
10.0.7.8 - 5 [10.0.2.128 | 1]
```

A subnet interface with the IP address $IP$ and the estimated TTL distance $TTL$ can be expressed as `IP - TTL [trail]`. The trail is put into brackets for clarity. Example 6.1 shows a toy set of subnet interfaces, consecutive with respect to the IPv4 scope, four of which feature a trail with an anomaly. Though the ingress router of these four interfaces cannot be clearly identified, they still might be on the same subnet since they exhibit identical trails. Indeed, this implies they are at the very least located around the same place in the network topology. This conclusion is further supported, in this instance, by the fact that they were all located at the same TTL distance, i.e., they likely constitute the pivot interfaces of a same subnet. Finally, since `10.0.7.1` is located exactly one hop earlier than all the other interfaces, it could very well be the contra-pivot interface of a `10.0.7.0/28` subnet (or greater) that encompasses all listed interfaces.

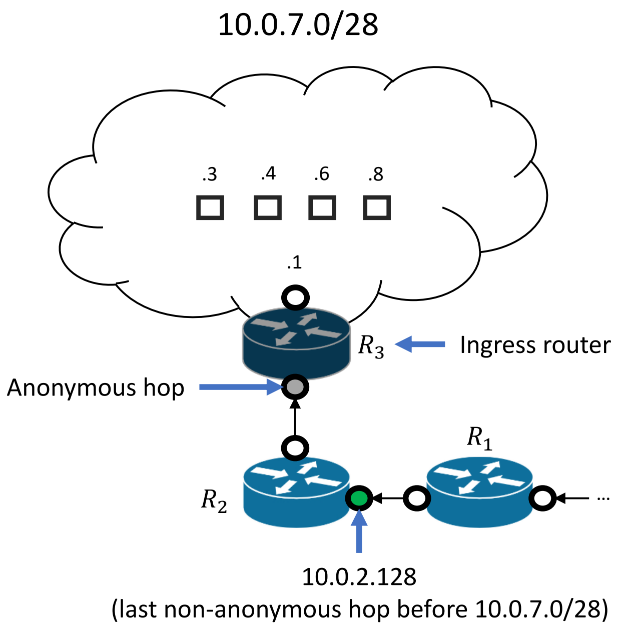


Figure 6.1: Toy topology inducing the trails from Example 6.1. Squares are pivot interfaces.

Figure 6.1 shows a toy topology that could induce the trails from Example 6.1. In this figure, the ingress router is darkened to emphasize that it does not respond to `traceroute` probes, i.e., it induces anonymous hops when it is crossed by `traceroute` probes. The interface on this router that cannot be seen at all is darkened as well. The green circle depicts the last non-anonymous router hop, `10.0.2.128` in this instance, which appears in the trails shown in Example 6.1. Because of the consistency of TTL distances and anonymous hops (i.e., they do not result from rate-limiting in this instance), it may be that the listed interfaces are on the same subnet. This simple example shows that reconsidering the last hop(s) towards an IP interface as a trail could offer a satisfying approximation of a subnet when `traceroute` issues prevent the discovery of the ingress router interface(s).

### 6.1.2 Warping trails

***Warping trails*** constitute the first challenge that needs to be tackled to achieve accurate subnet inference in the current Internet. As the term *warping* suggests, warping trails denote trails that are not consistently observed at the same TTL distances, i.e., they are "warping" within the `traceroute` records. While any trail can be warping, it is especially important to identify warping anomaly-free trails as these trails clearly identify interfaces of routers that are observed at different TTL distances depending on the target IP address of the analyzed `traceroute` records.

Having properly identified router interfaces appearing at varying TTL distances strongly suggests that warping trails are a direct consequence of asymmetrical paths in networks that implement traffic engineering policies such as load balancing [14]. The consequence for subnet inference is that the same ingress router can be observed for a set of subnet interfaces that are not located at the same TTL distance from the vantage point. The former observation would suggest the interfaces are on the same subnet, but classical subnet inference tools (cf. Sec. 5.2) usually assume that the (pivot) interfaces of a same subnet should be observed at a constant TTL distance. A modern subnet inference methodology should therefore be designed so that the trail has priority over the TTL distances to mitigate the effects of asymmetrical paths in network architectures. Figure 6.2 depicts a toy topology that can result in the observation of warping trails among interfaces of a same subnet, with the warping trails being associated to the green interface of $R_6$ in the figure. The dashed circles depict the alternative (and asymmetrical) path in the toy load balancer.

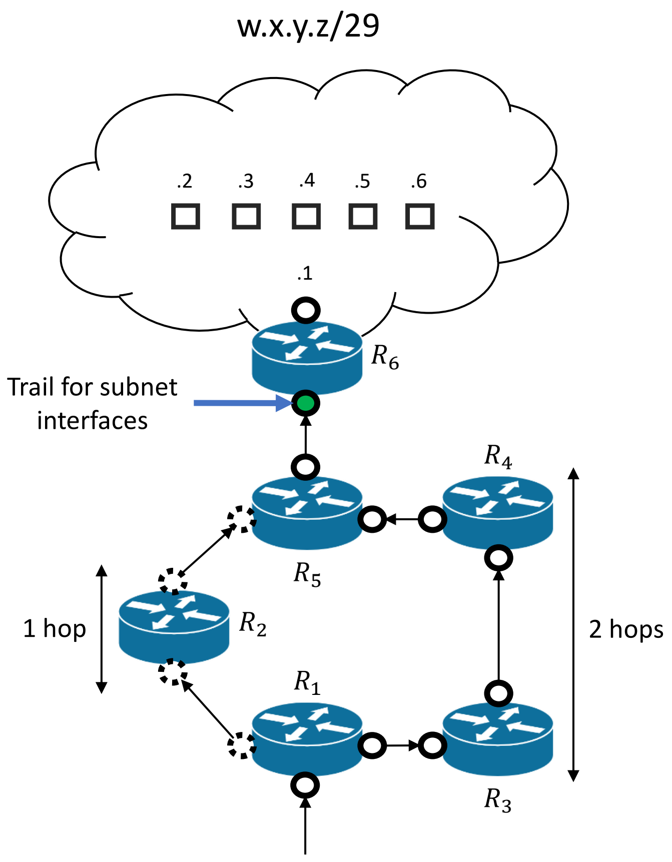


Figure 6.2: Toy topology inducing *warping trails*. Squares are pivot interfaces.

It should be noted that (Paris) `traceroute` [11], which was designed to stabilize paths in load balancing architectures, cannot prevent this phenomenon: indeed, (Paris) `traceroute` was designed to stabilize paths in *per-flow* load balancers to avoid false link discovery, and therefore does not prevent varying TTL distances being possible for one router interface.

### 6.1.3 Flickering trails

***Flickering trails*** are another consequence of traffic engineering. Given a set of subnet interfaces that are close in respect of the address space and located at the same TTL distances, the trails of these interfaces are said to be *flickering* if they are anomaly-free but appear one after the other in sequence. Example 6.2 shows a toy scenario of subnet interfaces exhibiting flickering trails, reusing the same notation as in Example 6.1. In this example, `10.0.3.1` and `10.0.3.64` are *flickering with each other*. The "flickering" term was picked to refer to how the expected result, i.e., seeing a specific interface of the ingress router, is changing in a quasi random manner.

Listing 6.2: Flickering trails

```
10.0.5.35 - 7 [10.0.3.1]
10.0.5.37 - 7 [10.0.3.64]
10.0.5.38 - 7 [10.0.3.1]
10.0.5.42 - 7 [10.0.3.1]
10.0.5.44 - 7 [10.0.3.64]
```

While warping trails are most likely a consequence of asymmetrical paths in load balancers, flickering trails are more likely to be a consequence of symmetrical paths in similar architectures. A subnet is reached through different paths of the same length that converge towards the same ingress router, and as a result, different interfaces from this ingress router are revealed by the `traceroute` probes. By definition, flickering trails can only be anomaly-free trails: observing different possible router interfaces one or more hops prior to a subnet does not mean the paths converge at the ingress router, as they may converge sooner.

Figure 6.3 depicts another toy topology that could induce the flickering trails observed in Example 6.2. The two green circles of $R_4$ depict the router interfaces that can appear as the flickering trails of the subnet interfaces from `10.0.5.0/25` due to the presence of symmetrical paths in the topology. Again, the dashed circles highlight the alternative path. It is worth noting that the phenomenon of flickering was actually first discovered on the ULiège network by reviewing the trails of subnet interfaces belonging to well-known subnets, and turned out to be induced by parts of the topology that were very close to the toy topology depicted in Figure 6.3.

In fact, Figure 6.3 is reminiscent of Figure 2.3 from Chapter 2. Indeed, the detection of flickering trails immediately raises the question of whether or not the associated router interfaces belong to the same device or not, therefore motivating the use of alias resolution. If an alias resolution method demonstrates that they belong to the same device, then subnet interfaces with aliased flickering trails should be considered as being on the same subnet. Ignoring this hypothetical scenario can result in subnet partitioning, i.e., the actual subnet is inferred as several smaller subnets due to the flickering trails. Because there are few flickering trails per subnet in practice, the framework from Chapter 3 is an ideal solution for resolving them.

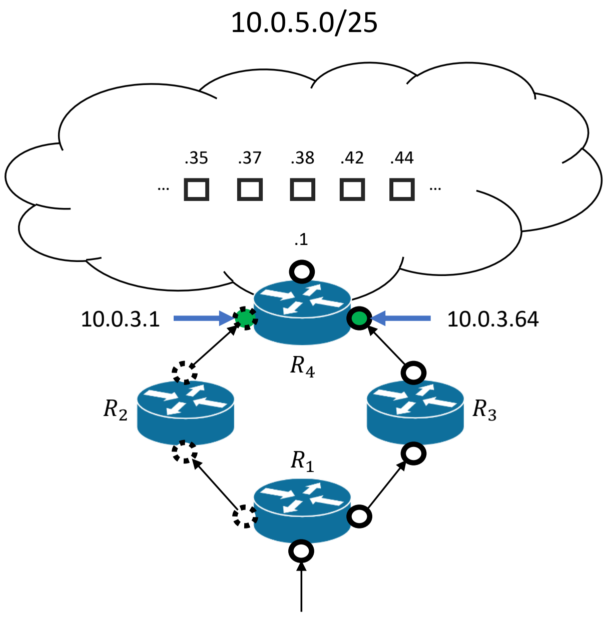


Figure 6.3: Toy topology inducing the *flickering* trails from Example 6.2. Squares are pivot interfaces.

It should be noted that, while spotting warping (anomaly-free) trails only requires a census of all TTL distances observed for each router interface associated to an anomaly-free trail to be done, detecting flickering trails is a bit more complicated. First of all, the largest possible ***flickering delta*** needs to be defined: it expresses the difference between the 32-bit integer equivalent of the IP addresses of two consecutive subnet interfaces (with respect to the address space). Then, if a sequence of three consecutive subnet interfaces $a$, $b$ and $c$ such that

- all three are observed at the same TTL distance,
- the deltas between $a$ and $b$ and $b$ and $c$ are smaller than or equal to the largest allowed delta,
- $a$ and $c$ have identical trails, that differ from the trail of $b$,

can be identified, then the trails associated to $a$, $b$ and $c$ can be considered to be flickering with each other. For example, the tree first subnet interfaces listed in Example 6.2 are enough to discover that `10.0.3.1` and `10.0.3.64` are flickering with each other if using a delta smaller than or equal to 4.

Obviously, the largest possible flickering delta should be tuned so that small gaps in the address space between consecutive IP addresses are permitted. But these gaps should also be small enough to avoid false positives, i.e., detecting flickering trails when the associated subnet interfaces are far enough from each other in the address space to infer they may be on distinct subnets. A practical algorithm for detecting flickering trails will be presented in Chapter 7, which will also briefly explain how the alias resolution framework of Chapter 3 should be tuned for resolving them.

### 6.1.4 Echoing trails

Finally, a modern subnet inference methodology should also take into account a peculiar phenomenon: ***echoing trails***. An echoing trail features the same IP address as the target interface of the `traceroute` probes that led to its discovery, i.e., the trail is "echoing" the target IP address. This observation might initially be assimilated as a cycle induced by a temporary network failure or some misconfiguration, but an echoing trail has two defining characteristics:

- the target will never reply to probes with a TTL value equal to the TTL distance of the trail,
- there is no other occurrence of the IP address of the target in the path leading to it.

In fact, echoing trails are a systematic observation for given ranges of IP addresses, rather than the result of occasional network failures or misconfiguration. This issue was first discovered while reviewing `traceroute` data collected on a Belgian ISP. After reviewing the data with the help of a network operator of the ISP, it transpired that the routers identified at the last hops of the collected routes were configured to use the target IP address as the source IP of their own `time-exceeded` replies as long as the source IP was in a neighboring subnet.

While this may initially suggest that subnet inference is impossible when the ingress router to some subnet(s) behaves as aforementioned, a best effort strategy is possible as long as the TTL distances are consistent. Indeed, if a set of subnet interfaces, consecutive with respect to the address space, are all located at the same TTL distances while exhibiting echoing trails, this suggests at the very least that the last hop router before each interface has the same approximate location and behaviour. Therefore, the subnet interfaces likely share the same ingress router, and could be considered as being on the same subnet. This heuristic will of course only work if the TTL distances are consistent: if asymmetrical paths are crossed on the way, it becomes difficult to assume that consecutive subnet interfaces with echoing trails but varying TTL distances are still on the same subnet. Example 6.3 shows a toy sequence of subnet inferences with echoing trails, reusing the same format as in Examples 6.1 and 6.2. Due to the constant TTL distances and consistent ingress router behavior, gathering these interfaces under the same prefix can still result in a sound subnet.

Listing 6.3: Echoing trails

```
10.0.8.128 − 4 [10.0.8.128]
10.0.8.130 − 4 [10.0.8.130]
10.0.8.131 − 4 [10.0.8.131]
10.0.8.133 − 4 [10.0.8.133]
10.0.8.135 − 4 [10.0.8.135]
```

Contrary to warping trails (cf. Sec. 6.1.2) and flickering trails (cf. Sec. 6.1.3), echoing trails are not the result of traffic engineering policies and are actually caused by specific network equipment. As discussed in the next section (cf. Sec. 6.2), they are an uncommon but noticeable occurrence in the wild. Due to their nature, echoing trails are always unique.

## 6.2 Subnet inference challenges in the wild

This section presents a study of warping trails (cf. Sec. 6.1.2), flickering trails (cf. Sec. 6.1.3) and echoing trails (cf. Sec. 6.1.4) in the wild, using data collected from the PlanetLab testbed during the fall of 2018. Sec. 6.2.1 first explains how the data was collected. Sec. 6.2.2 then presents the observations and discusses their implications for subnet inference.

### 6.2.1 Measuring problematic trails

To study trails in the wild, a new measurement tool was designed during the second half of 2018. This measurement tool was actually an early build of `WISE` [52, 53], a subnet inference tool created in the context of this thesis that will be presented in detail by Chapter 7. At the time, `WISE` was only meant to make a census of responsive IP addresses and conduct (Paris) `traceroute` measurements towards each one to be able to identify their respective trail. It was also designed so that it could minimize overhead probing while performing the `traceroute` measurements. With a few modifications, this early build was also able to detect all types of problematic trails introduced in Sec. 6.1 (using a delta value of 4 to detect flickering trails, cf. Sec. 6.1.3). The only task not performed by this early build, apart from subnet inference, was the alias resolution of flickering trails.

Because the detection and subsequent analysis of responsive IP addresses did not change to any great degree between the early build and the final version of `WISE` (with the obvious exception of resolving flickering trails), the algorithmic details of these operations will be provided in Chapter 7 rather than in this section. Example 6.4 nevertheless provides a small sample of the typical data that the early `WISE` collected, coming from a measurement performed on AS6453 (TATA Communications) from the PlanetLab testbed in December 2018 [1]. The format of the data is the same as in the Examples 6.1, 6.2 and 6.3. Note how the IP addresses starting in `116.0.70.` have varying TTL distances but similar trails: this suggests `180.87.12.45` is subject to warping.

Listing 6.4: Sample of data collected by the early build of `WISE` (December 2018)

```
116.0.68.150 - 15 [116.0.82.62]
116.0.68.151 - 14 [116.0.93.149]
116.0.68.201 - 13 [116.0.93.141]
116.0.68.206 - 13 [120.29.216.38]
116.0.70.1 - 12 [180.87.12.45]
116.0.70.2 - 10 [80.231.217.91]
116.0.70.3 - 12 [180.87.12.45]
116.0.70.4 - 11 [180.87.12.45]
116.0.70.5 - 10 [180.87.12.45]
```

[1] `https://github.com/JefGrailet/WISE/blob/master/Dataset/AS6453/2018/12/10/AS6453_10-12.ips`

The early WISE collected data, not only to design the subnet inference it would later provide, but also to study problematic trails. Moreover, since warping trails and flickering trails both result from load balancing, the measurements were scheduled to renew the vantage points at each run. Indeed, traffic engineering can vary depending from where a target network is being measured.

For convenience, the data collected on a single target network by any measurement tool from a single vantage point and on a given date is defined as a ***snapshot***. By collecting snapshots from the same target network but from different vantage points (and on different dates), it becomes possible to quantify how problematic trails depend on the vantage point.

Due to the limited amount of working PlanetLab nodes at the time [2], measurements with WISE were scheduled with ***vantage point rotation***, i.e., on each date, each target network would be measured from a specific vantage point. After collecting the new snapshots for all target networks, a new measurement was scheduled but only after shifting the vantage points. By shifting all vantage points between each measurement of all target networks once, it became possible to collect a snapshot of the same target network from each available vantage point. The whole measurement campaign was said to be finished when a complete rotation was achieved, i.e., for each target network, there was a snapshot from each vantage point (or an attempt at collecting one).

| **ASN** | **Name** | **Origin** | **Category** |
|---|---|---|---|
| 12956 | Telefonica Global Solutions | Spain | Tier-1 |
| 3257 | GTT Communications Inc. | USA | Tier-1 |
| 6453 | TATA Communications | USA | Tier-1 |
| 6762 | Sparkle | Italy | Tier-1 |
| 10010 | TOKAI Communications | Japan | Transit |
| 13789 | Internap Corporation | USA | Transit |
| 22652 | Fibrenoire Inc. | Canada | Transit |
| 286 | KPN B.V. | Netherlands | Transit |
| 30781 | Jaguar Network | France | Transit |
| 50673 | Serverius Holding B.V. | Netherlands | Transit |
| 5400 | British Telecom | UK | Transit |
| 6939 | Hurricane Electric LLC | USA | Transit |
| 703 | Verizon Business/UUnet ASPAC | USA | Transit |
| 8220 | COLT Technology | UK | Transit |
| 8928 | Interoute Communications | UK | Transit |
| 109 | Cisco Systems | USA | Stub |
| 14 | Columbia University | USA | Stub |
| 224 | UNINETT | Norway | Stub |
| 37 | Navy Network Information Center (NNIC) | USA | Stub |
| 4711 | INTEC Inc. | Japan | Stub |
| 52 | University of California | USA | Stub |
| 802 | York University | Canada | Stub |

Table 6.1: Summary of the 22 Autonomous Systems probed with WISE (early build) in late 2018.

[2] Only a few dozens of nodes could be used in late 2018.

The data discussed in Sec. 6.2.2 was collected between November 29th and December 21st, 2018. During this campaign, 22 different Autonomous Systems (or ASes) were measured by the early build of `WISE` from 22 distinct PlanetLab nodes. By the end of the campaign, and since each AS had been measured once from each PlanetLab node, almost 484 snapshots had been collected (i.e., minus a few cases where the corresponding PlanetLab nodes had poor reachability). Table 6.1 shows a summary of the ASes that were measured during this campaign. Most of the ASes found in this selection are the same as those measured with `TreeNET` in 2017 (cf. Sec. 4.3 from Chapter 4), but with updated prefixes and some additions, such as AS3257 (GTT Communications, Inc.) and AS6762 (Sparkle), two Tier-1 ASes selected for their sizes (slightly more than two millions of attributable IP addresses for AS3257, and less than 200,000 for AS6762) and high ranking in CAIDA's AS ranking [2] at the time.

### 6.2.2 Observations from the PlanetLab testbed

One way to visualize the frequency of problematic trails within a given target network is to look at the occurrences of trails affected by each issue. For instance, if a trail is found to be warping, all occurrences of this trail in a snapshot can be counted and compared to the total number of discovered trails in the same snapshot to evaluate the extent of the issue. Evaluating problematic trails by occurrences rather than uniqueness is motivated by the fact that the latter would distort interpretations in at least two ways. On the one hand, warping/flickering trails can be observed with a few specific interfaces while these interfaces continue to appear as the trails for many (sometimes most of) responsive IP addresses, so quantifying them with respect to unique trails could make them look less important than they actually are. On the other hand, echoing trails would be over-represented: echoing trails are unique by definition, therefore making up a large share of the unique trails found in a snapshot, while few subnet interfaces are actually concerned by this issue.

Given a snapshot, the total trails affected by each kind of problem can be divided by the total number of collected trails to compute a ratio. Then, for each target network, the ratios obtained for each snapshot can be plotted depending on the snapshot. With this approach, it then becomes possible to visualize, in a single figure, the extent of each issue for each target network, depending on the vantage point. Due to the considerable amount of ASes probed between November 29th and December 21st, 2018, only a few representative figures will be discussed in this section, but figures for any campaign and any target AS can be browsed online in the public GitHub repository of `WISE` [3].

The first conclusion that can be drawn from Figure 6.4, which illustrates AS6453 (TATA Communications), a Tier-1 AS, is that all issues appeared in every snapshot, with the obvious exception of December 17th, 2018, which was the result of the corresponding vantage point having poor reachability, as very few trails could be obtained in practice. There is also no mention of a snapshot for December 18th, 2018 (contrary to subsequent figures) as the corresponding PlanetLab node did not find any responsive IP.

[3] `https://github.com/JefGrailet/WISE/tree/master/Evaluation/Obstacles`

The second conclusion is important regarding subnet inference: the extent of each issue varies considerably from one snapshot to another. Among the very first snapshots of the collection, problematic trails were in the minority, but almost 80% of the trails were affected by consequences of load balancing (i.e., warping and flickering) in other snapshots. In particular, three *hills* can be seen in the figure: December 6th and 7th, December 10th and 11th, and finally December 15th and 16th. In all three cases, warping was the most common issue, but flickering was also well represented. Another very interesting result is how flickering and warping seemed decorrelated: for instance, on December 6th and 7th, flickering affected less than 20% of trails while almost 80% of them were affected by warping. There are however similarities between December 15th and 16th, but only one snapshot showed any degree of correlation for December 10th and 11th. Finally, it is also worth noting that flickering is sometimes the most common issue, as seen with the snapshots collected during the early days of December 2018. All in all, these observations clearly suggest warping and flickering were not caused by the same type of traffic engineering policy, as discussed in Sec. 6.1.

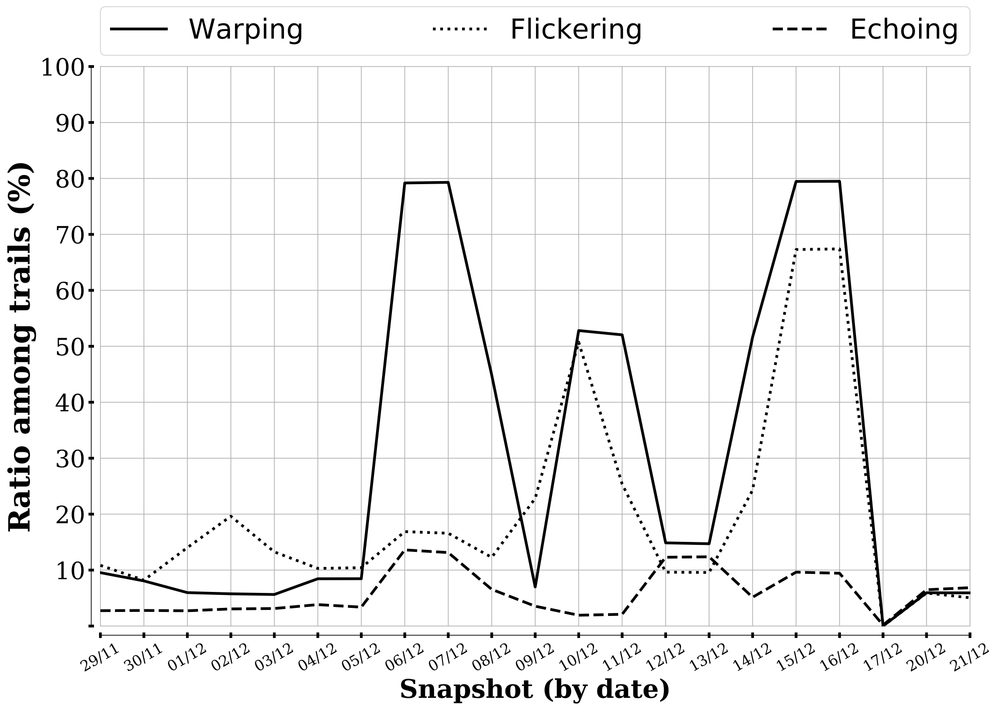


Figure 6.4: Problematic trails in AS6453 (snapshots from November 29th and December 21st, 2018).

While the early build of `WISE` did not perform alias resolution on flickering trails (cf. Sec. 6.2.1), it should be noted that a considerable amount of IP interfaces subject to flickering could be aliased in multiple snapshots collected with the final `WISE` (presented in Chapter 7). In particular, when it comes to snapshots of AS6453, many alias pairs or (small) alias lists could be detected thanks to the `iffinder`-like (cf. Sec. 3.3.1.1) and `Ally`-like (cf. Sec. 3.3.1.2) methods used in the framework outlined in Chapter 3. For instance, on February 19th, 2019, 37 out of the 62 IP interfaces associated with flickering trails ended up in alias pairs or lists, and there were as many as 84 out of 119 IP addresses that appeared on alias pairs/lists the next day (from another vantage point) [4]. In both snapshots, flickering trails were first detected using a delta equal to 4 (cf. Sec. 6.1.3). Because the aforementioned alias resolution methods are reliable when working with small sets of alias candidates, these results further support the initial hypothesis that flickering results from reaching an ingress router through different paths (cf. Sec. 6.1.3).

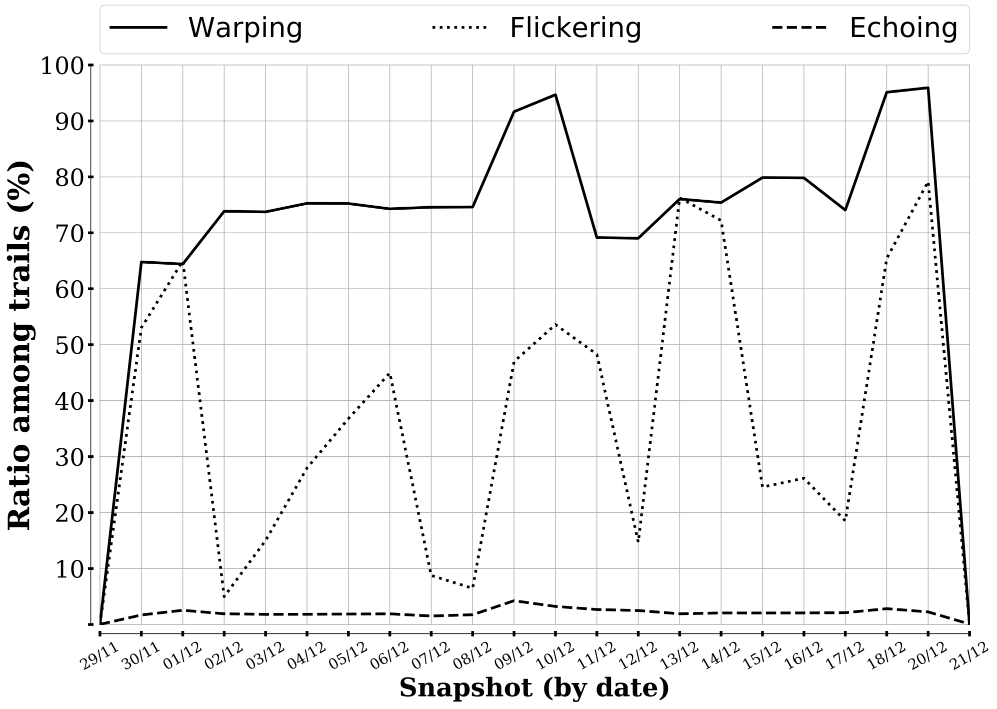


Figure 6.5: Problematic trails in AS3257 (snapshots from November 29th and December 21st, 2018).

Figure 6.5 shows results for another Tier-1 AS, AS3257 (GTT Communications). Similar conclusions can be drawn when it is compared to Figure 6.4: the presence of all three issues, the varying amount of trails affected by each, and the decorrelation between warping and flickering. Figure 6.5 however differs from Figure 6.4 in two ways. First, more than 60% of trails are subject to warping in all snapshots with the exception of the first and last snapshots, captured from PlanetLab nodes with poor reachability. The ratios vary much more with flickering, as affected trails range from less than 10% to almost 70% of all trails depending on the snapshot. Second, the echoing trails are quasi constant. Such an observation should not be too surprising, since echoing trails are consequences of specific equipment (cf. Sec. 6.1.4). It is, in fact, more surprising that the ratios of echoing trails vary slightly with AS6453. Several hypotheses can be formulated: the IP addresses with echoing trails might be reached through different equipment depending on the vantage point, small variations in the amount of responsive IP addresses might change the ratios, and finally, some network misconfiguration or failure might induce an actual cycle misinterpreted as echoing.

For completeness' sake, previous figures will be now compared with the observations made on

[4] `https://github.com/JefGrailet/WISE/tree/master/Dataset/AS6453/2019/02`

ASes featuring different classifications. First, Figure 6.6 shows the observations made on AS14, a small AS managed by the Columbia University (located in New York City) usually characterized as a Stub AS. The ratios of echoing and warping trails appear to be quasi constant and fairly low (i.e., below 10% of the total of trails), with the exception of two large *hills* of warping trails resulting from snapshots collected on December 4th, 5th, 17th and 18th, respectively.

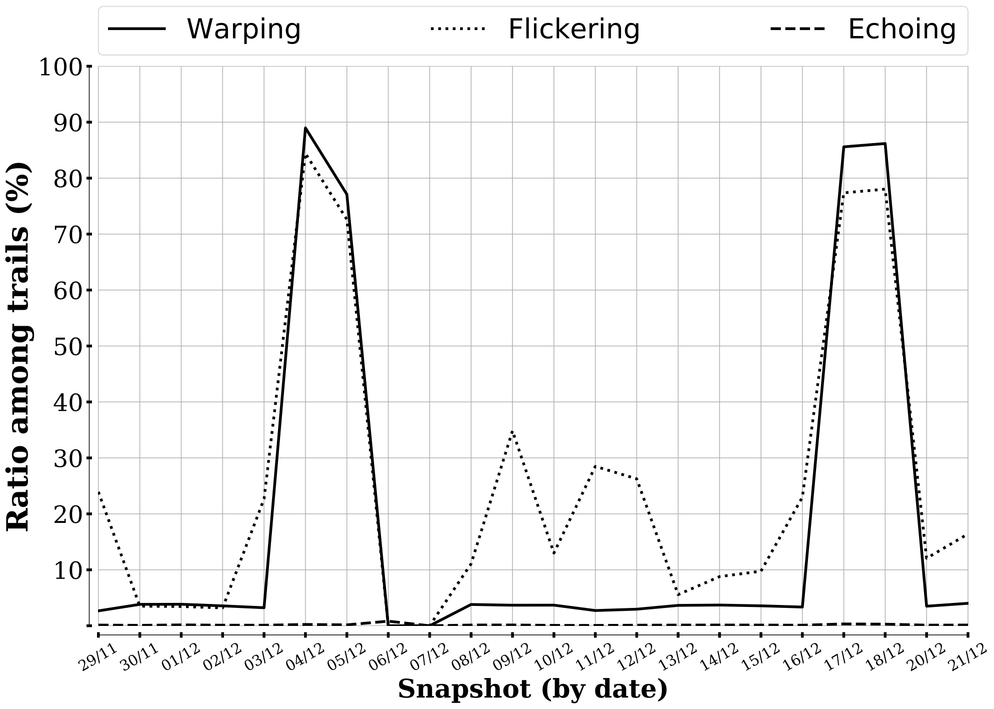


Figure 6.6: Problematic trails in AS14 (snapshots from November 29th and December 21st, 2018).

Interestingly, the vantage points used for December 4th and December 5th were the same as those used to capture the snapshots of December 15th and December 16th for AS6453, resulting in a comparable *hill*. Moreover, those vantage points were PlanetLab nodes located in Sweden, while both AS14 and AS6453 were located in the US, suggesting that the traffic engineering issues could be correlated with the large geographical distances. However, similar phenomenons were observed with the same ASes despite using PlanetLab nodes located in the US, on December 17th and December 18th for AS14 and December 6th and December 7th for AS6453. If it is also considered that flickering trails occurred more frequently than warping trails with AS14, it can be concluded that the subnet inference challenges discussed in this chapter are not exclusive to networks as important as Tier-1 ASes. This also further reinforces the conclusion that no issue cannot be completely avoided unless a very close vantage point is used, i.e., from which traffic engineering will be minimized.

Finally, Figure 6.7 shows the observations for AS286, a Transit AS managed by KPN B.V., a Dutch ISP. This time, the observations are somewhat similar to those illustrated on Figure 6.4, with one exception: two of the three *hills* span over the equivalent of four snapshots instead of two, unlike in previous figures. This suggests that, regardless of whether the vantage points were similar to those

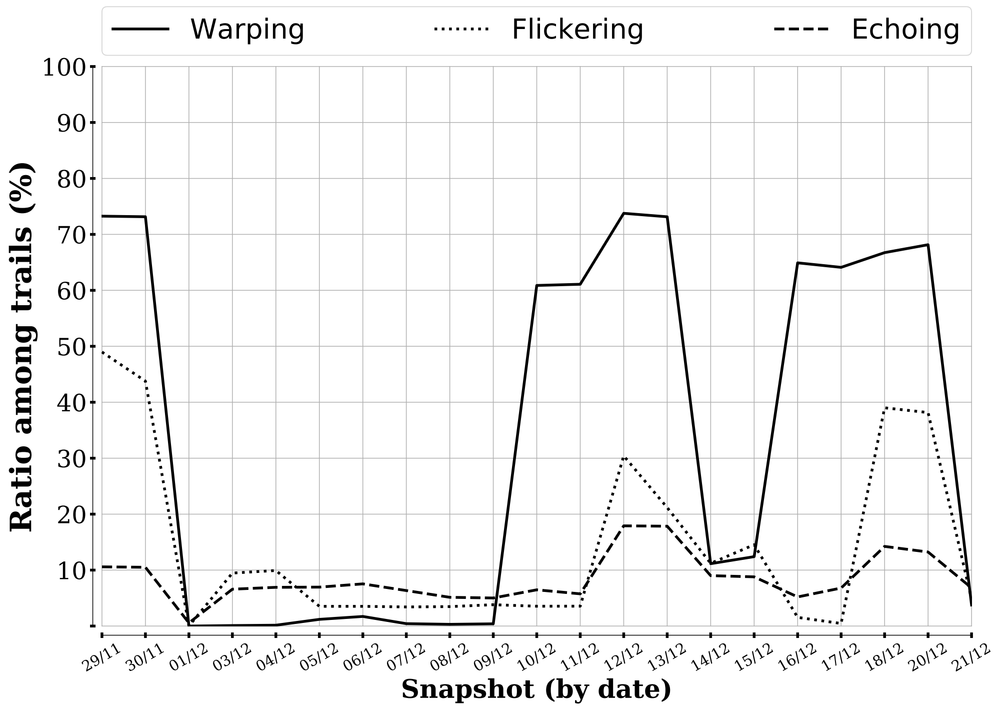


Figure 6.7: Problematic trails in AS286 (snapshots from November 29th and December 21st, 2018).

used to probe previous ASes and induced comparable results, the traffic engineering issues cannot be completely linked to the use of specific vantage points. In this instance, at least four PlanetLab nodes did not lead to similar observations for AS286 (or AS6453) and AS14.

## 6.3 Closing comments

As highlighted by the figures shown in Sec. 6.2.2, traffic engineering and its consequences should be considered as unavoidable in the Internet today, and the traditional hypotheses of subnet inference (cf. Sec. 5.1) should be re-evaluated. In particular, the TTL distance of subnet interfaces should no longer be considered as a requirement, but rather an indication: the trail (as defined in Sec. 6.1.1) should be the main criteria for verifying whether or not two subnet interfaces are reached through the same ingress router. Trails themselves are subject to the consequences of load balancing, and because of flickering trails (cf. Sec. 6.1.3), the trails of IP interfaces from the same subnet can vary. Such an issue can only be solved with the help of alias resolution, e.g., with the fingerprint-based framework of Chapter 3. Moreover, with echoing trails (cf. Sec. 6.1.4), the trails cannot be used to identify one or several interface(s) of the ingress router, which means only a best effort approach can be attempted. `WISE` [52, 53], a new subnet inference tool introduced in Chapter 7, implements a methodology to handle trails carefully and make the most of the data to accurately discover subnets.

# Chapter 7

# WISE: Wide and lInear Subnet inferencE

This chapter introduces a new tool that provides a linear time subnet inference scheme able to deal with traffic engineering: WISE [52, 53] (for **W**ide and l**I**near **S**ubnet inferenc**E**). It therefore answers research question 2 from Sec. 5.4. Section 7.1 begins with an overview of the tool, notably detailing its purpose, its features and its workflow. The subsequent sections provide the implementation and algorithmic details of WISE: Sec. 7.2 introduces some recommended data structures, Sec. 7.3 presents the probing work and Sec. 7.4 explains the subnet inference itself.

## 7.1 WISE overview

WISE is a subnet inference tool capable of discovering the subnets of a target network, given a set of input (IPv4) prefixes managed by the network. Its initial purpose is to refresh state-of-the-art subnet inference practices by addressing the challenges previously discussed in Chapter 6, i.e., it independently detects warping trails (cf. Sec. 6.1.2), flickering trails (cf. Sec. 6.1.3) and echoing trails (cf. Sec. 6.1.4) to take them into account while inferring subnets. In particular, it includes the alias resolution framework introduced in Chapter 3 in order to resolve flickering trails (cf. Sec. 7.3), therefore allowing the discovery of subnets whose ingress router is a convergence point (cf. Sec. 7.3.3).

Not only is WISE better armed against traffic engineering (and specifically load balancing) than its state-of-the-art predecessors, but it is also designed so that all its main algorithms feature a linear time complexity. These two features result from its overall methodology, as WISE proceeds in three major steps: discovery of the responsive IP interfaces, also known as ***target pre-scanning***, estimation of the TTL distance and trail of each discovered interface, also known as ***target scanning***, and offline ***subnet inference***. This workflow is summarized in Figure 7.1 as a flow-chart. The chart demonostrates how the two first steps, i.e. target pre-scanning and target scanning, are the steps that were already implemented in the early build of WISE mentioned in Sec. 6.2.1.

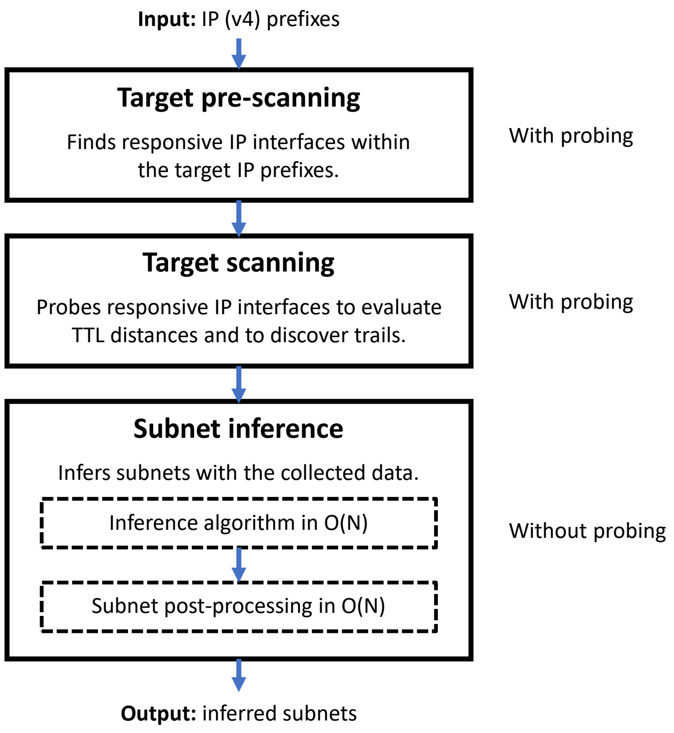

**Input:** IP (v4) prefixes
Target pre-scanning
Finds responsive IP interfaces within the target IP prefixes.
With probing
Target scanning
Probes responsive IP interfaces to evaluate TTL distances and to discover trails.
With probing
Subnet inference
Infers subnets with the collected data.
Inference algorithm in O(N)
Subnet post-processing in O(N)
Without probing
**Output:** inferred subnets

Figure 7.1: `WISE` workflow.

Rather than probing subnet interfaces while discovering subnets (like `ExploreNET`, cf. Sec. 5.2.2), `WISE` first collects the data it needs for subnet inference such that the inference can be carried out without additional probing. This data collection first starts with the target pre-scanning, which simply consists of probing every attributable IP address within the prefixes of a target network once to make a census of responsive IP interfaces. In practice, `WISE` performs this operation two or more times (as the user determines necessary) to avoid missing IP interfaces because of delays (cf. Sec. 7.3.1).

The main goal of target pre-scanning is to reduce the total of IP interfaces the target scanning should consider, as a way to simplify the probe scheduling. Indeed, this second step sends several probes towards each responsive IP interface to estimate its TTL distance and discovers its trail, both being required for subnet inference. Because the target scanning analyzes each responsive IP interface regardless of what subnet it is on, the time complexity of the operation can easily scale the total number of responsive IP interfaces. This step is made even more efficient in practice thanks to probing heuristics and multithreading, as will be explained in Sec. 7.3.

Once `WISE` is familiar with all responsive IP interfaces as well as their respective TTL distances and trails, it infers subnets by finding the IP prefixes that best accommodates sets of consecutive (and responsive) IP interfaces. This can be achieved by reviewing the list of discovered IP interfaces after sorting it with respect to the address space, i.e., interfaces which the IP addresses have the lowest 32-bit integer values come first (e.g., `10.0.1.255` comes before `10.0.2.4`). `WISE` can then pick the first IP interface on the list, grow a subnet prefix from it and extract the subsequent IP interfaces encompassed by the expanding subnet from the list. As the subnet grows, it checks a set of rules to guarantee the encompassed IP interfaces are part of the same subnet, and halts the growth if the subnet no longer fits the criteria. Once a subnet reaches its final size, `WISE` can proceed by finding the next subnets encompassing the remaining IP interfaces. As explained in Sec. A.2.4 of Appendix A, each discovered IP interface is evaluated a bounded number of times, resulting in an $\mathcal{O}(N)$ inference where $N$ denotes the number of discovered IP interfaces. The subnet inference is completed with a short post-processing, also in linear time, that aims at improving the overall accuracy (cf. Sec. 7.4.2).

Thanks to its design, `WISE` is a convenient tool able to scan the subnets within a set of target IPv4 prefixes in a short amount of time. As a result, it can be easily deployed from a single vantage point to discover the subnet-level of a target network encompassing hundreds of thousands of responsive IP interfaces. Finally, it is worth noting that `WISE` is an open source tool with a source code and dataset that can be browsed online via a GitHub repository [1].

The rest of this chapter presents `WISE` in detail: the recommended data structures for any implementation (Sec. 7.2), the probing stage (Sec. 7.3) and the offline subnet inference (Sec. 7.4). Each major algorithm will be described with pseudo-code. Their respective time complexity, whose detailed analysis can be found in Appendix A (Sec. A.2), will be mentioned as well. A validation of `WISE` will be provided in Chapter 8, along with an analysis of snapshots (as defined in Sec. 6.2.1) of the subnet-level of various ASes collected from the PlanetLab testbed throughout 2019.

## 7.2 Recommended data structures

The algorithms detailed in later sections can be implemented in any programming language as long as probing utilities are available (usually as one or several side libraries). However, in order to make any implementation convenient and efficient, at least three *ad hoc* data structures are recommended: an ***IP dictionary*** (Sec. 7.2.1), an ***alias set*** (Sec. 7.2.2) and an ***IP aggregator*** (Sec. 7.2.3). These structures are not mandatory but can help to achieve linear complexity with some algorithms detailed in subsequent sections. Note that these data structures are not exclusive to `WISE` and will also be involved in some algorithms of `SAGE` [54] (**S**ubnet **AG**gr**E**gation), another topology discovery tool developed during this thesis that will be presented in detail in Chapter 11.

### 7.2.1 IP dictionary

In order to store and manage the IP interfaces `WISE` discovers during the course of its execution efficiently, an *IP dictionary* could be implemented. The purpose of this data structure is to map any IP address with a small data structure providing all the details recorded for the interface associated with this address, typically consisting of: the estimated TTL distance, the trail, a path obtained by probing this IP address with (Paris) `traceroute` (if any) and the alias resolution data collected for it

[1] `https://github.com/JefGrailet/WISE`

(if any). This data can be complemented with various flags advertising any special behaviour, such as being part of a trail, or being affected by warping, flickering or echoing (as defined in Sec. 6.1).

It is crucial that the access to the data tied to any IP address, also called (dictionnary) ***entry***, is $\mathcal{O}(1)$. This can be achieved by using a hash table or by implementing a time/memory trade-off with more simple structures. One option consists of using the $X$ first bits of an IP address as the index of a large array of lists, so that the size of the lists keeps a fairly low upper bound, achieving $\mathcal{O}(1)$. For instance, with the IPv4 address space, the 20 first bits of any IP address can be used as an index for the array of lists (therefore featuring $2^{20} = 1,048,576$ cells), resulting in lists having at most 4,096 entries (i.e., the capacity of a /20 prefix). This solution is actually used by the public implementation of WISE [49] (as well as SAGE [48]). In addition to storing data, an IP dictionary should also provide various operations such as listing the recorded IP addresses (with or without special behaviour), flagging special IP addresses, or writing all the entries to an output file. In later sections, such operations will be used to get the input(s) of various algorithms, sometimes implicitly.

### 7.2.2 Alias set

During the course of its execution, WISE will use alias resolution to resolve IP addresses previously identified as flickering trails (as defined in Sec. 6.1.3). After discovering alias pairs/lists, it should be possible to retrieve a pair or list easily on the basis of an IP address that is part of it. Such an operation is notably performed during the subnet inference step (cf. Sec. 7.4) to check whether two subnet interfaces with flickering trails are accessed through the same ingress router.

In order to manage alias pairs/lists in a convenient manner, an ***alias set*** can be created to look up one alias pair or list encompassing a provided IP address easily. One way to do this is to manage a list of the alias pairs/lists (as data structures/objects) and a structure mapping any previously aliased IP address to the associated alias pair/list (still as an object or data structure) at the same time. In doing so, the pairs/lists can both be easily enumerated, notably to write them in an output file, while making it easy to quickly look up a previously discovered alias pair or list. Depending on the nature of the map structure, i.e. hash table or (balanced) tree [2], the looking up can be done in $\mathcal{O}(1)$ or $\mathcal{O}(\log N)$ with $N$ denoting the total number of IP addresses being part of alias pairs/lists.

### 7.2.3 IP aggregator

To minimize probing while resolving flickering trails, WISE can take advantage of the alias transitivity property first mentioned in Sec. 2.2. As a reminder, this property goes as follows: given three IP addresses $A$, $B$, and $C$, if $A$ and $B$ are aliases of each other while $B$ and $C$ are aliases of each other too, then $A$ is also an alias of $C$. One way to take advantage of alias transitivity consists of building the largest hypothetical alias lists before running the framework of Chapter 3. By doing so, each alias candidate will be probed only once in the context of WISE, while testing separate alias pairs could result in some alias candidates being probed multiple times.

[2]The `map` from C++ is typically a red-black tree, cf. `https://en.cppreference.com/w/cpp/container/map`.

To achieve this, an ***IP aggregator***, i.e., a data structure that eases the construction of hypothetical alias lists, can be created. Such a structure can receive a possible alias pair and check whether or not it has already recorded an alias pair/list already encompassing one of the IP addresses found in the pair. If an overlap exists between the new pair and one of the previously recorded pairs/lists, the aggregator builds a merged alias list on the basis of alias transitivity (unless both IP addresses in the new pair are already known). In practice, an IP aggregator can be implemented in almost the same manner as an alias set (cf. Sec. 7.2.2), i.e., by mapping any alias candidate to the current alias pair/list it is found in. This can be implemented, again, with the help of a hash table or a balanced tree (e.g., like `std::map` in C++). As a consequence, a pre-recorded alias pair/list look-up can be achieved in constant or logarithmic time (with respect to the total of alias candidates).

## 7.3 `WISE` probing

This section describes step by step the probing work of `WISE`. Sec. 7.3.1 presents the target pre-scanning step while Sec. 7.3.2 explains the target scanning. Then, Sec. 7.3.3 details the final step of target scanning, i.e., post-processing the collected data to detect the issues described in Chapter 6, and in particular, *flickering* trails.

### 7.3.1 Target pre-scanning

The target pre-scanning begins with the list of IP addresses to probe. This list is typically obtained by enumerating all attributable IP addresses contained in the IPv4 prefixes provided as input to `WISE`. As already mentioned in Sec. 7.1, the pre-scanning consists of sending a single probe towards each target IP address to make a census of all responsive IP addresses within the target network. For this to be made possible, a high TTL value (e.g., 64) should be used in the probes to increase the chances of getting a reply. Though, in theory, probes could use any protocol, the public implementation of `WISE` uses an ICMP `echo-request` since network interfaces reply more often to direct ICMP probes [81]. Responsive target IP addresses are recorded in the IP dictionary (cf. Sec. 7.2.1).

In order to make pre-scanning efficient, the target IP addresses can be shared between multiple threads. The way to schedule the threads is up to the implementation, as well as the total number of threads, though it is recommended that each thread probes a comparable amount of target IP addresses. However, it is highly recommended that the target IP addresses are ordered randomly to avoid probing them sequentially. Indeed, such a probing schedule could typically be identified as an attack on the target network and trigger rate-limiting policies.

Algorithm 6 summarizes target pre-scanning. This algorithm should not be changed when using multiple threads: each thread will simply have its own list $L$. A probe with a large TTL value (e.g., 64, as shown at line 4) is sent to the target for each target IP address provided to the thread (via the list $L$ in the algorithm). The probe can consist of a simple ICMP `echo-request`, as suggested by line 2, but other approaches can be considered if thought more appropriate. The thread then waits for the reply

**Algorithm 6** Target pre-scanning

```
Require: $T$, the timeout delay for a single probe
Require: $L$, list of target IPs (for this thread)
Require: $D$, the IP dictionary
1: procedure TARGETPRESCANNING(Time $T$, List<IP> $L$, Dictionary $D$)
2:     prober ← new ICMPProber($T$)
3:     for target ∈ $L$ do
4:         reply ← prober.probe(target, 64)
5:         if reply.isValid() then
6:             $D$.record(target)
```

and makes sure the reply is valid, i.e., if using an ICMP `echo-request`, a valid reply will be a correctly formed ICMP `echo-reply` whose source IP address matches the target IP address. If the reply is valid, the target IP address will be recorded in the IP dictionary (denoted $D$ in the algorithm). When using multithreading, such an operation must be made *thread-safe*. This is done by implementing mutual exclusion between threads upon recording a new IP address in the dictionary, unless the implementation of the dictionary guarantees there cannot be data corruption on recording different IP addresses concurrently. It is worth noting that, in the case of the public implementation of `WISE`, a timeout to a probe is modeled by a placeholder object returned by the `prober` object. Implicitly, the `isValid()` method at line 5 returns false if `reply` is a placeholder. Finally, when a large number of IP addresses turn out to be responsive, a thread should ideally wait for a short time (a few milliseconds) before probing the next target, again to avoid triggering rate-limiting policies.

As briefly mentioned in Sec. 7.1, pre-scanning should be conducted more than once: a target IP address can be unresponsive as a result of a temporary network failure. Therefore, it is recommended to re-do pre-scanning at least once to maximize the number of discovered responsive IP addresses. The public implementation of `WISE` typically runs the pre-scanning twice, with the second pre-scanning round using twice the timeout value of the first round. This is why line 2 of Algorithm 6 uses a parameter $T$ giving the timeout delay for a probe. The public implementation of `WISE` offers an optional third pre-scanning round triggered by the user when deemed appropriate.

As detailed in Sec. A.2.1, the target pre-scanning scales linearly with the total number of target IP addresses. Finally, it is worth noting that `TreeNET` [55] already implemented a step equivalent to target pre-scanning, as can be seen in its source code [3] and as briefly mentioned in Sec. 4.2.1. The differences between `WISE` and `TreeNET` in that regard are inconsequential.

### 7.3.2 Target scanning

The target scanning starts by listing all the responsive IP addresses that were previously recorded in the IP dictionary. In order to conduct subnet inference, each recorded IP address needs to be probed individually in order to estimate its TTL distance and discover its trail (as defined in Sec. 6.1.1). This can be achieved by running Paris `traceroute` [11] towards each target IP address. The TTL distance

[3] `https://github.com/JefGrailet/treenet`

will be equal to the TTL value used by the last probe sent by Paris `traceroute`, i.e., the first probe whose reply features the target IP address as the source. The trail can be easily derived from the last hop(s) of the discovered route. The `traceroute` itself can be performed with differing network protocols, though the public implementation of `WISE` relies on ICMP `echo-request` packets. As a consequence, the first reply coming from the target should be an ICMP `echo-reply` packet.

However, because `WISE` only requires the final hop(s) towards each responsive IP address (and the resulting TTL distance), the target scanning uses a ***probing reduction heuristic*** to avoid performing a full `traceroute` measurement for each responsive IP address. Because interfaces from the same subnet should ideally appear at the same TTL distance, IP interfaces that are consecutive (and close) in the IPv4 address space are likely to appear at similar TTL distances. Moreover, subnets that are close to each other in the address space can also appear at similar distances. Therefore, on probing a succession of known IP interfaces, the target scanning can rely on the TTL distance discovered for one interface to reduce the number of probes used to scan the next interfaces.

Before using the heuristic, the target scanning picks the very first target on the list of responsive IP addresses to scan. This first target is probed with the classical (Paris) `traceroute`, the TTL value of the probes starting at 1 [4], and increasing with each probe until a reply (i.e., an ICMP `echo-reply` packet in the case of `WISE`) from the target IP address has been received. This operation can be referred to as ***forward probing*** [39]. The TTL value of the last probe is recorded as the TTL distance between the vantage point and the target IP address, while the last hops before the target are used to infer the trail. If these last hops are anonymous, additional probes are sent to make sure this is not the result of a temporary failure. These additional probes first use the TTL distance of the target, minus one, and decrease the distance with each new probe, until a non-anonymous IP interface replies. This process of sending probes with a decreasing TTL can be referred to as ***backward probing*** [39].

The probing reduction heuristic can be used right from this point. The TTL distance discovered for the first target is used as the starting TTL value for the forward probing towards the second target. If the reply to the first probe does not come from this second target, then the forward probing continues as normal. In this scenario, backward probing is eventually required if and only if the last hops prior to the target prove problematic for trail discovery. If the reply to the first probe comes directly from the new target, however, the forward probing ceases already to be replaced with backward probing. This probing heuristic is not entirely new: it is very close to ordinary backward probing [38].

In this scenario, backward probing is mandatory and has two goals. First, it minimizes the TTL distance of the new target by decreasing the TTL value of the probes until direct replies from this target are no longer received. Second, after adjusting the TTL distance, it collects just enough `traceroute` data to reliably discover the trail. If the new target happens to be on the same subnet as the previous target, then only two additional probes are necessary to confirm its TTL distance and find its trail, as long as this trail is anomaly-free and if there is no warping (cf. Sec. 6.1.2). When the trail is anomaly-free, the first additional probe both confirms the TTL distance and discovers the

[4] In practice, with `WISE`, another initial TTL value can be used by tuning the probing parameters.

trail itself, while the second additional probe guarantees the IP address associated to the trail is not cycling [5]. More broadly speaking, having discovered the last non-anonymous router interface in a path, at least one additional probe is sent to make sure it is not cycling.

Regardless of the heuristic outcome, unless the second target IP address is located exactly next to the vantage point (i.e., the TTL distance is equal to 1), there will be less probes in total than if a full `traceroute` measurement has been conducted, especially if the second target happens to be on the same subnet as the first (Sec. A.2.2 will assess this). The heuristic can be repeated with each subsequent target by re-using the TTL distance of the previous target, and target scanning as a whole can be further sped up with multithreading, but with one major difference in respect of the target pre-scanning (cf. Sec. 7.3.1): the probing reduction heuristic will only fully make sense if the targets scanned by a single thread are consecutive in the address space. This detail aside, multithreading and other probing details (such as conditions for interrupting forward/backward probing to avoid problematic scenarii) depend on the implementation used.

**Algorithm 7** Target scanning for a set of responsive (and consecutive) IP addresses

```
Require: $T$, list of reponsive (and consecutive) IP addresses (as objects)
 1: procedure TARGETSCANNING(List<IPEntry> $T$)
 2:     prevTTL ← 1
 3:     for target ∈ $T$ do
 4:         startTTL ← prevTTL
 5:         FORWARDPROBING(target, startTTL)
 6:         foundTTL = target.getTTL()
 7:         if foundTTL == startTTL or target.badRoute() then
 8:             BACKWARDPROBING(target)
 9:         target.setTrail()
10:         prevTTL ← target.getTTL()
```

Algorithm 7 summarizes target scanning, including the probing reduction heuristic. In the same way as Algorithm 6, this pseudo-code should not change when using multiple threads, as each thread can simply have its own list of targets $T$. However, this pseudo-code assumes the targets are directly provided as objects rather than as IP addresses, i.e., as their respective counterpart in the IP dictionary (cf. Sec. 7.2.1). This is mostly for convenience: indeed, handling the objects directly avoids going back and forth to the IP dictionary (with mutual exclusion in case of multithreading). Before scanning the targets, a `prevTTL` variable is created (line 2) in order to maintain the TTL distance of the previous target, used to perform the probing reduction heuristic. There is, of course, no such TTL at the start, so it is initially set to 1, which means that the `startTTL` variable used for the forward probing can, in all cases, be set with `prevTTL` (cf. line 4). Forward probing is then performed (line 5), storing the (partial) `traceroute` measurement within the provided object in mean time and setting the TTL distance at the end. A need for backward probing is then evaluated: if the distance discovered by forward probing equals the TTL value used by the first probe (as shown at line 7), then backward probing is performed to both minimize the TTL distance (and adjust it if needed) and collect enough data for

[5]Here, the definition of cycle is restricted to consecutive occurrences of the same IP address.

trail discovery. Backward probing is also performed when the end of the (partial) route towards the target leads to a trail with anomalies. Note that `backwardProbing()` only receives the target object because this object is assumed to already contain the (current) TTL distance this procedure requires. Once the probing has been completed, the object for the target has all the data it needs to set the associated trail (line 9) and the `prevTTL` value can be updated for the next target.

Finally, it should be noted it is still possible that temporary failures or rate-limiting mechanisms can impair target scanning, especially when it comes to trail discovery. This can happen, for instance, on probing interfaces found in large and well populated subnets (i.e., most attributable IP addresses on the subnet are responsive). Therefore, to maximize the amount of anomaly-free trails, a short period of time can be allowed after having scanned all IP addresses, before repeating backward probing on the target IP addresses whose trails still contain anomalies. The public implementation of `WISE` implements such a mechanism. While scheduling backward probing tasks (i.e., threads), `WISE` ensures targets are attributed to each thread by TTL distance: this increases the chances that the same part of the target network gets re-probed by only one thread, again to minimize the risks of triggering rate-limiting mechanisms. The inclusion of this mechanism depends on the implementation.

Like target pre-scanning, the target scanning scales linearly with the total number of target IP addresses. It is important to note, however, that the probing reduction heuristic is essential to make this step efficient in practice. Indeed, the heuristic can cut the total number of probes sent towards the target IP addresses by more than 70% compared to an implementation without it. Both the time complexity and the benefits of the probing reduction heuristic are discussed in Sec. A.2.2.

### 7.3.3 Detection of problematic trails

After the target scanning step has been completed, one quick post-processing must take place in order to detect warping trails (cf. Sec. 6.1.2), flickering trails (cf. Sec. 6.1.3), and echoing trails (cf. Sec. 6.1.4). In particular, flickering trails must go through alias resolution so that the subsequent subnet inference (cf. Sec. 7.4) can tell whether or not two subnet interfaces with flickering trails have indeed been reached through the same ingress router.

The warping trail detection is hopefully trivial: it just requires going through the list of scanned IP addresses to make a census of all TTL distances observed for a given IP address appearing in trails. In practice, to keep track of the IP addresses found in trails while browsing the scanned IP addresses means simply taking advantage of the IP dictionary (cf. Sec. 7.2.1) and its (recommended) access in $\mathcal{O}(1)$, and making sure the $N$ scanned IP addresses can be processed in proportional time. Finding echoing trails is even more simple: due to their nature, the scanned IP addresses where the IP address of the trail is strictly equal to the scanned IP address simply needs to be flagged. Therefore, identifying echoing trails does not require looking up other entries in the IP dictionary.

Flickering trails, as already discussed in Sec. 6.1.3, require more care, as shown by Algorithm 8. All objects corresponding to scanned IP addresses are first extracted from the IP dictionary $D$ (cf. line 2). The `listScannedIPs()` method should provide them in an ordered manner with respect to the IP

**Algorithm 8** Detection of flickering trails

```
Require: $D$, IP dictionary
Require: $\delta$, the largest allowed flickering delta
 1: procedure DETECTFLICKERING(Dictionary $D$, Integer $\delta$)
 2:     lsIPs ← $D$.listScannedIPs()
 3:     it ← lsIPs.begin()
 4:     for it $\neq$ lsIPs.end() do
 5:         IP ← it.getContent()
 6:         if IP.getTrail().hasAnomalies() then
 7:             continue
 8:         if it.getPrev() == $\emptyset$ or it.getNext() == $\emptyset$ then
 9:             continue
10:         prevIP ← it.getPrev().getContent()
11:         nextIP ← it.getNext().getContent()
12:         if IPDIFF(prevIP, IP) > $\delta$ or IPDIFF(nextIP, IP) > $\delta$ then
13:             continue
14:         if prevIP.getTTL() $\neq$ IP.getTTL() or nextIP.getTTL() $\neq$ IP.getTTL() then
15:             continue
16:         prevT ← prevIP.getTrail()
17:         nextT ← nextIP.getTrail()
18:         if prevT.hasAnomalies() or !prevT.equals(nextT) then
19:             continue
20:         if IP.getTrail().equals(prevT) then
21:             continue
22:         flickering1 ← $D$.get(IP.getTrail().getIP())
23:         flickering2 ← $D$.get(prevT.getIP())
24:         flickering1.setAsFlickeringWith(flickering2)
25:         flickering2.setAsFlickeringWith(flickering1)
26:         it ← it.getNext()
```

address space. Next, each object is compared with the previous and following objects on the list (line 10 and 11). To consider the presence of possible flickering, all three IP addresses should be close in respect of the address space (line 12), using $\delta$ as the maximum difference, and must appear at the same TTL distance (line 14). If the previous and subsequent objects have the same (anomaly-free) trail (line 18) while differing from the (anomaly-free) trail of the current object (line 20), then the IP addresses associated with both trails can be considered as flickering with each other. As discussed in Sec. A.2.3, the detection of any type of problematic trail can scale linearly with the total number of scanned IP addresses as long as the recommended IP dictionary data structure is used (cf. Sec. 7.2.1).

Once all flickering trails have been identified, an IP aggregator (cf. Sec. 7.2.3) can be combined with the alias resolution framework of Chapter 3 to check which IP addresses associated with flickering trails are actually alias of each others. Each IP address associated with a flickering trail can be added to the IP aggregator, along with the IP addresses it is flickering with in the collected data. As the aggregator enforces alias transitivity (cf. 2.2), a list of the largest hypothetical alias lists can be obtained. Since none of these lists overlap (i.e., they share no common alias candidate), each one can be fed to the fingerprint-based framework to quickly test it. Due to flickering trails already suggesting the associated IP addresses could be alias of each others, the fingerprint-based framework can alias them by default when no practical method can be applied (cf. Sec. 3.3.2), though the public implementation of WISE permits the program to be configured in such a way that it only allows alias resolution based on the iffinder-like method (cf. Sec. 3.3.1.1) and methods based on the IP protocol identification field (cf. Sec. 3.3.1.2 and Sec. 3.3.1.3). Regardless of how the framework is tuned, the resulting alias pairs or lists can be stored in an alias set (cf. Sec. 7.2.2) for subsequent use during subnet inference (cf. Sec. 7.4). Unaliased IP addresses are subsequently no longer considered as being subject to flickering.

## 7.4 WISE inference

This section describes in detail how WISE can discover subnets, given the data collected and post-processed at target scanning (cf. Sec. 7.3.2 and Sec. 7.3.3). Sec. 7.4.1 presents the main algorithms of subnet inference, i.e., how WISE accommodates subnet prefixes on contiguous blocks of previously scanned IP addresses while ensuring subnet soundness. Sec. 7.4.2 then introduces and discusses a potential remaining issue after the inference, and how it can be mitigated with the help of a heuristic and a post-processing step. Pseudo-code algorithm(s) are provided for each operation, with their time complexity being reviewed in details in Sec. A.2 of Appendix A.

### 7.4.1 Subnet inference

The subnet inference performed by WISE is entirely offline: all the data it requires has already been collected during the previous steps (cf. Sec. 7.3). The process of aggregating the previously scanned IP addresses into subnets can be presented in a convenient manner by breaking it down in several operations. The main algorithm, which creates and grows individual subnets, will be reviewed first.

The main algorithm starts with the list of the successfully scanned IP addresses, sorted on the basis of their respective value as a 32-bit integer (by ascending order). An initial interface is picked at the top of the list and a /32 subnet is built for it. This first IP interface will subsequently be known as the ***reference pivot***, i.e., the interface used to test whether subsequent interfaces are on the same subnet. After creating this initial basic subnet, the algorithm decreases the prefix length of the subnet and removes all IP addresses that are encompassed by the expanded subnet from the list. These IP addresses will be denoted subsequently as ***candidate interfaces*** (or more simply *candidates*). Using the reference pivot, these candidates are reviewed to see if they could correspond to pivot interfaces (as defined in Sec. 5.1.2), i.e., IP addresses that are on the same subnet as the reference pivot. When a candidate is not a potential pivot, it can either be considered as a potential contra-pivot interface (as defined, again, in Sec. 5.1.2) or as an ***outlier***, i.e., an interface which cannot be considered neither as a pivot interface nor as a contra-pivot interface. Whether a candidate that is not a potential pivot will be ruled as a contra-pivot interface or as an outlier will depend mostly on its TTL distance.

Depending on the number of pivot interfaces, contra-pivot interfaces and outliers discovered among the candidates, the subnet inference decides whether the current subnet can be further expanded or if it should be kept either *as is* or with an increased prefix length. In the two first scenarios, the reviewed candidates are inserted into the subnet. The last scenario, i.e. increasing the prefix length, is equivalent to shrinking the subnet because too many candidate interfaces (if not all) are not on the same subnet as the reference pivot, leading to an inconsistent subnet. When shrinking occurs, the candidate interfaces must be restored to the list of the scanned IP addresses so they can be involved in the inference of the next subnet(s). As will be seen later, the main algorithm needs to take account of the possibility that the reference pivot may have been updated during the review of the candidates, i.e., it must be able to remember the reference pivot used at the last successful expansion so it can restore it if the current expansion changed it and if shrinking occurs. On a side note, a subnet is never grown beyond a given prefix length. In WISE, /20 is the smallest possible prefix length (as mentioned in Sec. 5.3, /20 is the smallest subnet prefix length observed in the wild [118]).

**Algorithm 9** Main algorithm of subnet inference

```
Require: I, sorted list of scanned IP addresses
Require: A, the alias set resulting from flickering trails
function SUBNETINFERENCE(List<IPEntry> I, AliasSet A)
    subnets ← new List<Subnet>()
    for I.size() > 0 do
        reference ← I.pop()
        subnet ← new Subnet()
        subnet.addInterface(reference)
        for subnet.getPrefixLength() ≥ MIN_PREFIX do
            subnet.expand()
            if subnet.overlap(subnets.back()) then
                subnet.shrink()
                break
            candidates ← new List<IP>()
            for subnet.encompass(I.front()) do
                candidates.add(I.pop_front())
            oldReference ← subnet.getReferencePivot()
            reviewed ← REVIEW(subnet, candidates, A)
            stop ← DIAGNOSE(subnet, reviewed)
            if stop > 0 then
                if stop > 1 then
                    subnet.shrink()
                    subnet.updateReferencePivot(oldReference)
                    for reviewed.size() > 0 do
                        I.push_front(reviewed.pop_back())
                else
                    subnet.insertInterfaces(reviewed)
                break
            else
                subnet.insertInterfaces(reviewed)
        subnets.add(subnet)
    return subnets
```

Algorithm 9 shows the main algorithm of subnet inference in WISE, starting with a list of scanned IP addresses $I$ and an alias set $A$ (as described in Sec. 7.2.2). This alias set provides the alias pairs/lists resulting from flickering trails discovered at the end of target scanning. This set is used to decide whether or not two flickering trails can be considered as belonging to the same ingress router. Lines 4–6 create a new /32 subnet and inserts inside the first IP from the list $I$. This first IP is automatically considered as the reference pivot. The subnet growth loop can start following this (line 7). The prefix length of the current subnet is first decremented (line 8), and the code determines whether the expanded subnet overlaps the last inferred subnet (if any), stopping the growth if such overlap exists (line 9). This is essentially to prevent the production of inconsistent results: indeed, depending on how a subnet expands, it can, in some instances, cover one or more previously inferred subnets whose interfaces are already out of $I$. After this check, a second loop removes all potential candidate interfaces from $I$ that are now encompassed by the new subnet, shown at lines 12–14. These candidates are then compared to the reference pivot, using the alias set $A$ if flickering trails are involved. The reference pivot does not appear directly in the call to the `review()` function as it is considered to be stored within the subnet object (line 16). Immediately prior to that, the reference pivot used by the last successful expansion is copied into a side variable in case the subnet needs to be shrunk (line 15), so that this previous reference pivot can be restored if necessary (line 21). The reviewed candidate interfaces are subsequently fed to a diagnostic function along with the subnet itself. This function, which will be detailed later (see Algorithm 11) returns: 0 if the subnet can be grown further, 1 if the growth needs to stop without changing the prefix length and 2 (or more) if the subnet has to be shrunk (line 17). If shrinking is required, the reviewed interfaces are put back in $I$ (lines 19–23). When the growth just needs to stop, the reviewed candidates are inserted into the subnet before quitting the current loop (line 25). If the growth can continue, the reviewed candidates are also inserted before the next iteration of the loop (line 28).

The $review()$ function shown in Algorithm 9 at line 16 will next be elaborated. This function replaces the process of determining whether each of the candidate interfaces can be considered as a potential pivot interface, as a contra-pivot interface or as an outlier. Remember that a candidate interface is considered as a potential pivot interface if it is said to be on the same subnet as the reference pivot. To check this, the properties of each candidate (trail and TTL distance) are compared to those of the reference pivot by testing up to five ***inference rules***. All rules have the same underlying idea: two IP interfaces, close with respect to the IP address space, can only be on the same subnet if the IP addresses associated to their trails belong to the same ingress router, and, if such a relationship cannot be elucidated, they should at the very least be observed at the same TTL distance while their trails should exhibit a similar behaviour. In other words, if two candidates are at the same TTL distance but have distinct trails, they can be considered as being reached through the same ingress router if they exhibit echoing trails or if the anomalies (cf. Sec. 6.1.1) differ between both trails (e.g., as a result of occasionnal rate-limiting policies). The rules are tested in sequence: if rule 1 cannot be met, rule 2 is tested, then rule 3 if rule 2 cannot be met either, etc. A candidate is ruled as a potential pivot interface of the current subnet as soon as one rule has been satisfied.

The five inference rules are:

- **Rule 1:** both interfaces have the exact same trail.
- **Rule 2:** both interfaces do not share the same trail, but are located at the same TTL distance and their trails differ in terms of anomalies (as defined in Sec. 6.1.1).
- **Rule 3:** both interfaces are located at the same TTL distance and exhibit echoing trails.
- **Rule 4:** both interfaces are located at the same TTL distance and their trails are both flickering with each other and previously aliased.
- **Rule 5:** both interfaces are not located at the same TTL distance and do not share the same trail; however, the IP addresses found in the trails were aliased as a result of resolving flickering trails, meaning they belong to the same device and are reached through asymmetrical paths.

Rule 2 is a way to ensure that an interface located at the same TTL distance as all other interfaces in a subnet is not mistaken for an outlier because its trail could not be discovered accurately (i.e., the trail contains anomalies due to a temporary failure or a rate-limiting policy at the time of target scanning). While testing this rule, an implementation should consider replacing the reference pivot with the candidate interface if this candidate has a better trail (i.e., less or no anomalies), so that subsequent candidates with less or no anomalies can be compared more accurately.

When a candidate interface does not meet any rule, two options remain: it is either a potential contra-pivot interface or an outlier. A candidate is considered as the former if and only if its TTL distance is lower than the TTL distance of the reference pivot. Otherwise, the interface is considered to be an outlier, except in a unique scenario. If the current subnet only features a single interface (the reference pivot) and if the difference in distance between the potential outlier and this interface is equal to 1 (i.e., the outlier is located farther), the algorithm swaps the roles and re-considers the reference pivot as a contra-pivot while the outlier becomes the new reference pivot. This special case ensures a contra-pivot interface, initially mistaken for a pivot interface, is eventually correctly identified as a contra-pivot.

Algorithm 10 provides a pseudo-code detailing how the `review()` function from Algorithm 9 works. This algorithm is straightforward, as it consists of selecting one of the candidates then testing each rule of inference until one is met (starting from line 5). Whenever a rule has been met or any other scenario confirmed, a `SubnetInterface` object is created and inserted onto the list of the reviewed candidates, combining the candidate itself with a constant value denoting the type of interface it would constitute within the subnet (each $RULE\_X$ constant denotes the rule of inference with which the associated pivot was successfully tested). However, there are several particular cases to consider. Testing rules 4 or 5 notably requires the alias set $A$, and if rule 2 is verified, the algorithm must check whether the candidate features a better trail than the reference pivot and replace it when necessary (lines 9–12). As can be seen at lines 17–18 and at lines 22–23, rule 5 is only tested when the

**Algorithm 10** Review of the candidates for a subnet $S$

```
Require: S, the subnet under expansion
Require: C, list of candidate interfaces (current expansion)
Require: A, the alias set resulting from flickering trails
 1: function REVIEW(Subnet S, List<IPEntry> C, AliasSet A)
 2:     reference ← S.getReferencePivot()
 3:     reviewed ← new List<SubnetInterface>()
 4:     for candidate ∈ C do
 5:         if RULE1(reference, candidate) then
 6:             reviewed.push_back(new SubnetInterface(candidate, RULE_1))
 7:         else if RULE2(reference, candidate) then
 8:             reviewed.push_back(new SubnetInterface(candidate, RULE_2))
 9:             betterReference ← COMPARETRAILS(reference, candidate)
10:             if betterReference ≠ reference then
11:                 reference ← betterReference
12:                 S.updateReferencePivot(betterReference)
13:         else if RULE3(reference, candidate) then
14:             reviewed.push_back(new SubnetInterface(candidate, RULE_3))
15:         else if RULE4(reference, candidate, A) then
16:             reviewed.push_back(new SubnetInterface(candidate, RULE_4))
17:         else if ref.getTTL() > candidate.getTTL() then
18:             if RULE5(reference, candidate, A) then
19:                 reviewed.push_back(new SubnetInterface(candidate, RULE_5))
20:             else
21:                 reviewed.push_back(new SubnetInterface(candidate, CONTRAPIVOT))
22:         else if ref.getTTL() < candidate.getTTL() then
23:             if RULE5(reference, candidate, A) then
24:                 reviewed.push_back(new SubnetInterface(candidate, RULE_5))
25:             else
26:                 diff ← candidate.getTTL() - ref.getTTL()
27:                 if S.getNbInterfaces() == 1 and diff == 1 then
28:                     S.clear()
29:                     S.add(candidate)
30:                     reviewed.push_back(new SubnetInterface(reference, CONTRAPIVOT))
31:                     reference ← candidate
32:                 else
33:                     reviewed.push_back(new SubnetInterface(candidate, OUTLIER))
34:         else
35:             reviewed.push_back(new SubnetInterface(candidate, OUTLIER))
36:     return reviewed
```

candidate interface and the reference pivot have distinct TTL distances. When the TTL distance of the candidate is lower, either the candidate turns out to be a potential pivot via rule 5, or it is ruled to be a potential contra-pivot (lines 17–21). When the TTL distance of the candidate is higher and rule 5 is not satisfied, then either there is an outlier (line 33), or a scenario where the reference pivot might be a contra-pivot after all. This scenario is plausible if and only if the subnet $S$ contains only one interface (the reference pivot) while the difference in TTL between the reference pivot and the candidate is strictly equal to 1. If verified, the roles between these interfaces are exchanged (lines 26–31) and kept for the rest of the review. Finally, when no rule has been met while the candidate and the reference pivot feature equal TTL distances, the candidate is ruled as an outlier (line 35).

The $diagnose()$ function shown in Algorithm 9 at line 17 is the last main operation to review. This part of the algorithm consists of deciding whether or not to include the reviewed candidates in *S* results in a sound subnet or not, returning an integer value demonstrating if *S* can be expanded further (0), if its growth should stop (1) or if it needs to be shrunk (2). The diagnosis mostly depends on the presence of potential contra-pivot interfaces among the reviewed candidates. If there is no potential contra-pivot interface at all, the subnet either keeps growing or is shrunk, the decision being based on the ratio of outliers found within the candidate interfaces. If pivot interfaces are in the majority, then the growth can continue. In the public implementation of `WISE`, pivot interfaces are considered to be in the majority if outliers make up less than 1/3 of the number of candidates [6].

If the candidate interfaces contain one or several potential contra-pivot interfaces, the diagnosis must be more thorough. First of all, if there are more interfaces other than contra-pivot interfaces among the candidates, the outliers should not account for more than 1/3 of the candidates, otherwise the scenario is too risky to be kept *as is*. The current prefix length is also rejected if the number of contra-pivot interfaces is above a constant [7] or if there are more contra-pivots than pivots in total (i.e., including the pivots already stored in *S*). There is, however, a practical exception to the latter: `WISE` tolerates subnets featuring two contra-pivot interfaces and one pivot interface, as it has been observed on groundtruth networks (which will be discussed in Sec. 8.1) that such a scenario is possible with small subnets. Such subnets usually play a key role in the network topology and are therefore reached through several interfaces of the ingress router (i.e., there is at least one back-up interface), but do not not necessarily have more than one pivot interface.

If there is only one contra-pivot candidate at this point, then the subnet growth stops, because a subnet featuring mostly (if not only) pivot interfaces with the exception of one contra-pivot interface already corresponds to the ideal measurement-based definition of a subnet (cf. Sec. 5.1.2). Doing this also prevents the subnet inference from badly inferring groups of small subnets (i.e., /30 or /31) that are consecutive in the IP address space and which often consist of one pivot and one contra-pivot interface: as `WISE` is supposed to tolerate multiple contra-pivot interfaces, growing past that point can produce overgrown subnets. However, if the candidates include more than one possible contra-pivot interface, the diagnosis should perform additional tests to guarantee that the contra-pivot interfaces are consistent. Because there are several possible and arbitrary tests, this part of the subnet inference can be freely tuned depending on which guarantees the implementation should put on contra-pivot interfaces. A simple test consists of checking whether contra-pivots are located at the same TTL distance and share the same trail. In practice, while the public implementation of `WISE` does the first, it does not perform the second as traffic engineering can cause a subnet to be reached through multiple paths, which can result in having multiple contra-pivot interfaces with differing trails.

[6] As with other arbitrary values, this can be tuned upon starting `WISE`.

[7] In `WISE`, the maximum number of contra-pivots is set to 5.

However, WISE implements a custom heuristic named ***overgrowth test***. This test consists of trying to accommodate sub-prefixes on the reviewed candidates in order to check if it is possible to obtain sub-prefixes covering together all the candidates while each sub-prefix features only one contra-pivot interface. This test can be easily implemented by re-using the main principles of Algorithm 9 on a copy of the list of candidates, but using the contra-pivot interfaces as the initial pivot of each sub-prefix. Each sub-prefix is then expanded until it starts overlapping the other contra-pivot interface(s), at which time it is shrunk once. Other candidate interfaces overlapped by this sub-prefix are then removed from the copy of the list of candidates (along with the contra-pivot itself), and if no candidates remain after creating each sub-prefix, then this means ideal sub-prefixes can be discovered within the list of candidate interfaces. In such a scenario, WISE rejects the current subnet prefix because these sub-prefixes are closer to the ideal definition of a subnet, and assumes the current subnet may simply not have a *visible* contra-pivot interface. This test, which is performed, at worst, once per subnet, is designed to prevent subnet overgrowth and can be performed in linear time with respect to the number of candidates. Depending on how an implementation processes the subnets after their inference, this heuristic may or may not be implemented.

**Algorithm 11** Diagnosis for a group of reviewed candidates

```
Require: S, the subnet under expansion
Require: R, list of reviewed candidate IPs
 1: function DIAGNOSE(Subnet S, List<SubnetInterface> R)
 2:     nbPivots ← COUNTPIVOTS(R)
 3:     nbContrapivots ← COUNTCONTRAPIVOTS(R)
 4:     nbOutliers ← COUNTOUTLIERS(R)
 5:     maxOutliersRatio ← R.size() / 3
 6:     if nbContrapivots > 0 then
 7:         nonContrapivots ← nbPivots + nbOutliers
 8:         if nonContrapivots > nbContrapivots and nbOutliers ≥ maxOutliersRatio then
 9:             return 2
10:         else if nbContrapivots > MAX_CONTRAPIVOTS then
11:             return 2
12:         else if nbContrapivots > nbPivots + S.getNbPivots() then
13:             return 2
14:         if nbContrapivots > 1 then
15:             if CONSISTENTCONTRAPIVOTS(R) then
16:                 return 1
17:             else
18:                 return 2
19:         else
20:             return 1
21:     else
22:         if nbOutliers ≥ maxOutliersRatio then
23:             return 2
24:         else
25:             return 0
```

Algorithm 11 summarizes, in pseudo-code, the operations carried out by the `diagnose()` function mentioned in Algorithm 9. It starts by quantifying the different types of interfaces and establishing the maximum tolerated number of outliers, denoted maxOutliersRatio in the pseudo-code, as shown at lines 2–5. The subsequent operations depends on whether or not there are potential contra-pivot interfaces (line 6), and in particular, the diagnosis entirely depends on the ratio of outliers among candidates if there is no contra-pivot interface at all (lines 22–25). Otherwise, the diagnosis confirms that the quantities for each type of interface are still sound (lines 7–13). If there is more than one contra-pivot interface, diagnose() has to check the candidate interfaces further to verify whether the contra-pivot interfaces are consistent, notably by checking if all contra-pivot interfaces appear at the same TTL distance or by performing the overgrowth heuristic used by the public implementation of WISE (line 15). These additional tests, which can be customized, are symbolized here by the consistentContrapivots() function, which returns true when the contra-pivot interfaces satisfy all tests. In that case, the diagnose() function as a whole can return 1 (no shrinking), otherwise it will return 2. Finally, as none of these tests are performed when there is a single contra-pivot interface, the function will immediately return 1 in this scenario.

Despite featuring several sub-operations nested in its main algorithm (cf. Algorithm 9), the subnet inference as performed by WISE scales linearly with the number of IP interfaces it has to aggregate into subnets. This notably results from the small size of subnets in the IPv4 address space [92] as well as most sub-operations being carried on small subsets of the initial IP interfaces. A detailed analysis of the time complexity of the WISE subnet inference is provided in Sec. A.2.4.

### 7.4.2 Subnet post-processing

Despite being cautious, the subnet inference detailed in Sec. 7.4.1 can occasionally produce undergrown subnets. Indeed, a subnet stops growing as soon as it includes one or several contra-pivot interfaces (i.e., several during the same expansion), and subsequent subnets cannot overlap previously discovered subnets by design. Both design decisions aim at preventing subnet overgrowth, but can end up partitioning large subnets, much like with ExploreNET (cf. Sec. 5.2.2). This notably occurs when the contra-pivot interface(s) are found among the first IP addresses of a growing prefix. As a result, a large subnet can appear as one long prefix, matching the measurement-based definition of a subnet (cf. Sec. 5.1.2), followed by a succession of similar or smaller prefixes exclusively containing pivot interfaces. Figure 7.2 illustrates this partitioning issue for a /28 subnet where the four first IP addresses allowed to find a sound /30 prefix, resulting in the remaining pivot interfaces ending up in a /30 prefix and another /29 prefix, both being undergrown in respect of the actual subnet.

This partitioning issue is addressed in one of two ways. First, the subnet inference in WISE actually reviews the scanned IP addresses in reverse, i.e., it starts with the interfaces whose IP addresses are the highest in the IP address space when considering their equivalent 32-bit value. This heuristic is motivated by the fact that, in the vast majority of snapshots (as defined at Sec. 6.2.1) collected with either WISE or previous state-of-the-art tools (cf. Sec. 5.2), the contra-pivot interfaces tend to appear among the first IP addresses of large subnets rather than the opposite. As a result, processing the scanned IP addresses in reverse ensures the subnet inference will review a large number of

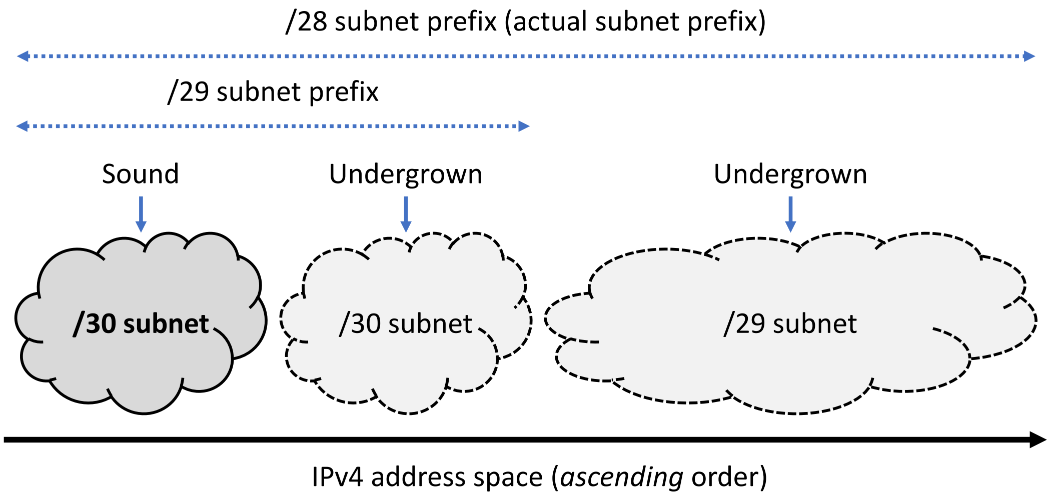

/28 subnet prefix (actual subnet prefix)
/29 subnet prefix
Sound
Undergrown
Undergrown
/30 subnet
/30 subnet
/29 subnet
IPv4 address space (*ascending* order)

Figure 7.2: A residual case of subnet partitioning after conducting the subnet inference of Sec. 7.4.1.

pivot interfaces before discovering contra-pivot interfaces, therefore maximizing the size of larger subnets. Unfortunately, this heuristic cannot completely prevent subnet partitioning, as contra-pivot interfaces can be found anywhere in the address space of a subnet in theory. This is why WISE also implements a post-processing step that detects undergrown subnets and attempts to merge them with subnets that are adjacent in the address space to recover larger subnet prefixes.

The post-processing as implemented by WISE works as follows. During the subnet inference (cf. Sec. 7.4.1), any subnet where the growth has had to be stopped because it started overlapping previously inferred subnet(s) (see line 9 of Algorithm 9) can be flagged so that it will be quickly identified by the post-processing. After the inference, the post-processing starts to review the list of inferred subnets until it comes across an undergrown subnet. Then, it decrements the prefix length of this undergrown subnet and examines the list to enumerate subnets overlapped by the new prefix. In practice, because decreasing once the prefix length only expands the subnet on one side (i.e., either the lower bound, either the upper bound of the prefix has changed, but not both at the same time), picking subnets from one side of the list is enough. If there is at least one overlapped subnet, the post-processing verifies if merging the overlapped subnet(s) with the current undergrown subnet produces a more interesting result. If so, the overlapped subnets are moved to a side list storing the subnets which can be merged for sure and the expansion of the initial undergrown subnet resumes in order to see if there are subnets to be merged. Otherwise, if the merging scenario produces an incoherent result, the newly overlapped subnets are put back onto the list where they came from and the expansion stops, with the undergrown subnet being shrunk once. If the list of subnets that can be merged for sure is not empty, then they are actually merged with the undergrown subnet and the result is added to the final list of subnets. Post-processing will proceed by searching for another undergrown subnet, until it has completely reviewed the inferred subnets.

Algorithm 12 presents the main algorithm for post-processing a list of inferred subnets $S$. As explained before, it begins by picking a subnet from the list of inferred subnets and checks whether or

**Algorithm 12** Subnet post-processing (main algorithm)

```
Require: S, the list of inferred subnets
 1: function POSTPROCESS(List<Subnet> S)
 2:     postProcessed = new List<Subnet>()
 3:     for S.size() > 0 do
 4:         currentSubnet ← S.pop_front()
 5:         if !currentSubnet.isUndergrown() then
 6:             postProcessed.push_back(currentSubnet)
 7:             continue
 8:         mergeable ← new List<Subnet>()
 9:         for currentSubnet.getPrefixLength() ≥ $MIN\_PREFIX$ do
10:             candidates ← new List<Subnet>()
11:             leftExpansion ← currentSubnet.expandsToTheLeft()
12:             currentSubnet.expand()
13:             if leftExpansion then
14:                 for currentSubnet.overlaps(postProcessed.back()) do
15:                     candidates.push_front(postProcessed.pop_back())
16:             else
17:                 for currentSubnet.overlaps(S.front() do
18:                     candidates.push_back(S.pop_front()))
19:             if candidates.size() == 0 then
20:                 continue
21:             if !CANMERGE(currentSubnet, mergeable, candidates) then
22:                 currentSubnet.shrink()
23:                 for candidates.size() > 0 do
24:                     if leftExpansion then
25:                         postProcessed.push_back(candidates.pop_front())
26:                     else
27:                         S.push_front(candidates.pop_back())
28:                 break
29:             else
30:                 for candidates.size() > 0 do
31:                     mergeable.push_back(candidates.pop_front())
32:         if mergeable.size() > 0 then
33:             mergeable.push_back(currentSubnet)
34:             postProcessed.push_back(new Subnet(mergeable))
35:         else
36:             postProcessed.push_back(currentSubnet)
37:     return postProc
```

not this subnet is undergrown. If this is not the case, the traversal of the list simply resumes (lines 4–7). Otherwise, a list to store the subnets which can be merged is created and the post-processing starts expanding the undergrown subnet (line 8). At each new prefix length, the algorithm checks the direction in which the subnet is expanding and picks the subnets which are now overlapped in either direction (lines 10–18). If the list of overlapped subnets (named `candidates` in this example) is empty, then the expansion continues (line 19). Otherwise, the post-processing tests the merging scenario (line 21), providing all subnets that needs to be merged (including those that can be merged for sure, found at a previous iteration and stored in `mergeable`). Depending on the outcome, the algorithm either stops the expansion, restoring the newly overlapped subnets to their respective list and shrinking once in the process (lines 22–28), or stores the mergeable subnets in the `mergeable` list (created at line 8) before resuming the expansion (lines 30–31). Once the expansion loop ends, the subnets stored for merging are actually merged with the initial undergrown subnet and the resulting subnet is added to the final list of subnets (lines 32–34). If no optimal merging scenario is to be found, then the initial undergrown subnet is simply moved to the final list without any change (line 36) and the main loop resumes until it has processed all inferred subnets.

The `canMerge()` function shown in Algorithm 12 at line 21 and what it can do to decide whether a merging scenario is sound or not is next examined. The most straightforward approach consists of testing whether or not the following rules are satisfied or not.

- **Merging rule 1:** no more than one subnet among the overlapped subnets can contain one or several contra-pivot interface(s).
- **Merging rule 2:** no subnet in the merging scenario can be the result of a previous post-processing.
- **Merging rule 3:** each subnet other than the initial undergrown subnet has a reference pivot that can satisfy a subnet inference rule (cf. Sec. 7.4.1) when compared to the reference pivot of the initial undergrown subnet.
- **Merging rule 4:** when all subnet interfaces are mixed together, pivots are still in the majority.

The last rule more or less replicates the behaviour of the `diagnose()` function previously described in Algorithm 11 to ensure the final subnet is not filled with outliers [8]. The aforementioned rules are usually enough to recover a large subnet that was chunked during the inference due to the positioning of its contra-pivot interface(s), as shown by Fig. 7.2, and are easy to implement in practice.

It is worth mentioning that the rules for merging subnets can be freely tuned by the implementation depending on how strict the subnet post-processing should be. In fact, the public implementation of `WISE` [49] adds some flexibility to the aforementioned rules in practice. Indeed, when the reference pivot of a subnet does not match an inference rule when compared to the reference pivot of the initial undergrown subnet, `WISE` also compares a contra-pivot interface (if any) or an

[8]By default, `WISE` considers that a majority amounts to 2/3 of the total of interfaces.

outlier (if any) of the former to the reference pivot of the latter. For instance, it is possible that a badly placed outlier fools the subnet inference into concluding the outlier is a pivot interface for a small subnet while a pivot interface of the encompassing actual subnet is mistaken for a contra-pivot interface for this smaller subnet. WISE also considers this kind of scenario and records which type of interface could be on the same subnet as the reference pivot of the initial undergrown subnet. This is necessary to correctly quantify the pivot interfaces, contra-pivot interfaces and outliers of the hypothetical final subnet while checking the last merging rule suggested above. It is imperative that care is taken when testing the first merging rule, as a subnet with a contra-pivot interface which turns out to be a pivot interface (at least in the merging scenario) should no longer be considered as a subnet with contra-pivot interface(s) upon adding more subnets into the merging scenario. It is up to the implementation to decide whether the final algorithm should be tolerant, at the risk of having more subnets with unusual configurations (but less undergrown subnets). A detailed analysis of the practical effects of the subnet post-processing as performed by the public implementation of WISE can be found in Sec. A.2.6.

Finally, at the end of post-processing, the public implementation of WISE also performs additional tasks to complement the data. In particular, for each subnet, alternative prefix length is computed that equates to the longest prefix length (i.e., the smallest subnet size) that would contain all the listed interfaces. Indeed, a sparse subnet can occasionally feature a prefix larger than what is needed to encompass all its interfaces. WISE also uses an alternative definition of a contra-pivot interface for subnets where the TTL distances of pivot interfaces vary considerably due to the problem of warping (as defined in Sec. 6.1.2). In such a scenario, the sole or few IP address(es) that do not fulfill an inference rule when compared to the reference pivot can be identified as contra-pivot interface(s). In other words, this is a *best effort* strategy for finding such interfaces when TTL distances are not consistent enough to identify them on that basis, as WISE still assumes contra-pivot interfaces should appear closer to the vantage point (cf. Sec. 7.4.1).

Much like the subnet inference presented in Sec. 7.4.1, whose complexity is thoroughly detailed in Sec. A.2.4, the post-processing step of WISE features a linear time complexity. The main difference between both operations is that the post-processing scales with the total number of subnets inferred by the previous step rather than with the total number of IP interfaces successfully scanned at target scanning (cf. Sec. 7.3.2). This linear time complexity, which is thoroughly detailed in Sec. A.2.5, is also a consequence of the small size of IPv4 subnets [92].

## 7.5 Closing comments on WISE

In this chapter, the workflow of WISE [52, 53] has been extensively explained and commented on to show how it handles the challenges described in Chapter 6 in practice. The possibility of efficiently implementing these steps, notably by minimizing and parallelizing the probing work, was also discussed. To learn more about the efficiency of the algorithms detailed in this chapter, notably in terms of time complexity, interested readers can refer to Sec. A.2 of Appendix A.

While the accuracy of WISE will be thoroughly assessed in Chapter 8, it should already be noted that the WISE methodology could benefit from its probing being made a bit more complete. In particular, WISE can currently only handle IP addresses that reply to ICMP `echo-request` packets, as such packets are an easy and cheap way to quickly carry out a census of IP addresses that will be aggregated into subnets later (via the target pre-scanning, cf. Sec. 7.3.1). However, it is possible that other IP interfaces found in the initial target prefixes are configured not to reply to `echo-request` packets and can therefore only be detected through `traceroute` probing. By combining multiple ways of discovering *live* IP interfaces (i.e., interfaces that reply to at least one type of probe) and adapt target scanning (cf. Sec. 7.3.2) to each type of probing, a subnet inference tool could possibly discover more subnets than WISE, or at least refine some subnets it currently discovers.

Moreover, the current subnet post-processing (cf. Sec. 7.4.2) could be refined even further to more effectively handle difficult subnet inference scenarios, and several approaches to do so need to be explored. For instance, heuristics based on the typical IP assignment practices observed in the Internet could be added. This could be useful, for example, to avoid inferring a /28 within a /24 prefix if all other subnets found in this prefix are /30, because this implies the /24 prefix is used by the network operator only to manage point-to-point links (typically with /30 or /31 subnets).

Finally, while the public implementation of WISE [49] has been designed uniquely for IPv4, its methodology is arguably more suitable for IPv6. Because most of its probing work is performed on responsive IP addresses and since the subnet inference itself is done offline (cf. Sec. 7.4), an IPv6 WISE could be created as long as it could perform an equivalent of target pre-scanning for IPv6. Earlier tools, in particular `ExploreNET` [118] (cf. Sec. 5.2.2), could not be so easily adapted to IPv6 since they probe the network as they infer subnets, which is feasible with IPv4 and its small subnets but too costly with IPv6. Hopefully, discovering responsive IP addresses in IPv6 and more generally topology discovery in IPv6 is an active research topic [19, 106], opening doors for IPv6 subnet inference. It should be noted, though, that the efficiency of WISE still partly relies on the limits of the IPv4 scope (cf. Sec. A.2.4 and Sec. A.2.5), which means the subnet inference itself should be specifically tuned for IPv6 as well to guarantee fast and accurate subnet inference.

# Chapter 8

# WISE validation and deployment

This chapter thoroughly assesses WISE (described in Chapter 7) and discusses its deployment in the wild to answer research questions 3 and 4 from Sec. 5.4, i.e., whether WISE is accurate and whether it is able to discover the same subnets despite vantage point change. Sec. 8.1 first validates WISE on two groundtruth networks while comparing it with the state-of-the-art of subnet inference in terms of accuracy (i.e., how close to the groundtruth the inferred subnets are) and network use. Sec. 8.2 then presents the first observations conducted in the wild with WISE, using data snapshots collected from the PlanetLab testbed during 2019. Finally, Sec. 8.3 concludes this second part of the thesis by summarizing its main contributions and discussing possible improvements.

## 8.1 WISE validation

This section assesses WISE [52, 53] with the help of two groundtruth networks whose subnets were known in advance. Sec. 8.1.1 first describes in detail the networks that were measured, which state-of-the-art subnet inference tools (cf. Sec. 5.2) WISE was compared to and how all tools were deployed. Sec. 8.1.2 then presents the results of the validation, using custom metrics to assess the accuracy in terms of subnet inference of each tool.

### 8.1.1 Validation methodology

In order to validate WISE, two groundtruth networks were measured in 2019: the entire network of the University of Liège and the backbone of a Belgian ISP. The former had already been used to validate the alias resolution framework of Chapter 3 (cf. Sec. 4.2), while the latter provided a bigger network topology. Indeed, while the ULiège network predominantly consists of a /16 prefix (completed with two small /24 prefixes), the Belgian ISP manages IPv4 prefixes covering several hundreds of thousands of attributable IP addresses. With the friendly help of the SeGI (for reminders:

*Service Général d'Informatique*) and a network administrator working for the Belgian ISP, the real subnet prefixes of the whole ULiège network and the backbone of the ISP could be known in advance to evaluate how accurate the subnet prefixes discovered by WISE were [1].

Not only was WISE validated with these networks, but it was also compared with two other state-of-the-art subnet inference tools: ExploreNET [118] (cf. Sec. 5.2.2) and TreeNET [55] (cf. Sec. 5.2.3). As a reminder, TreeNET improves the subnets discovered by ExploreNET with a heuristic aimed at improving the identification of larger subnets. Rather than deploying both tools separately, an early version of SAGE [47] was used. While the final SAGE (cf. Chapter 11) is built on top of WISE, the early version reuses the subnet inference step of TreeNET and records the subnets as discovered by ExploreNET before refining them with TreeNET's heuristic (if needed). As a result, this specific tool allows subnets to be retrieved as inferred by ExploreNET and as improved by TreeNET.

The subnet inference step of TreeNET consisted of discovering responsive IP addresses, much like the target pre-scanning step of WISE (cf. Sec. 7.3.1), then running ExploreNET on each discovered responsive IP address. For each inferred subnet, TreeNET applied its heuristic if needed, and subsequently avoided running ExploreNET on responsive IP addresses that belonged to the discovered subnet. As such, the subnet inference step of TreeNET reused in the early SAGE [47] relies on active probing to discover subnets, unlike the subnet inference of WISE which works offline (cf. Sec. 7.4).

The ULiège network was measured from a vantage point within it [2] with each tool (i.e., WISE and the early SAGE), for the same reason as in Sec. 4.2: to avoid most probes from being blocked by firewalls. The tools were also run one after the other, rather than concurrently, to avoid sending too many probes at once. The Belgian ISP, on the other hand, was probed in a similar manner but from a European PlanetLab node, therefore a vantage point that was unrelated to the target network.

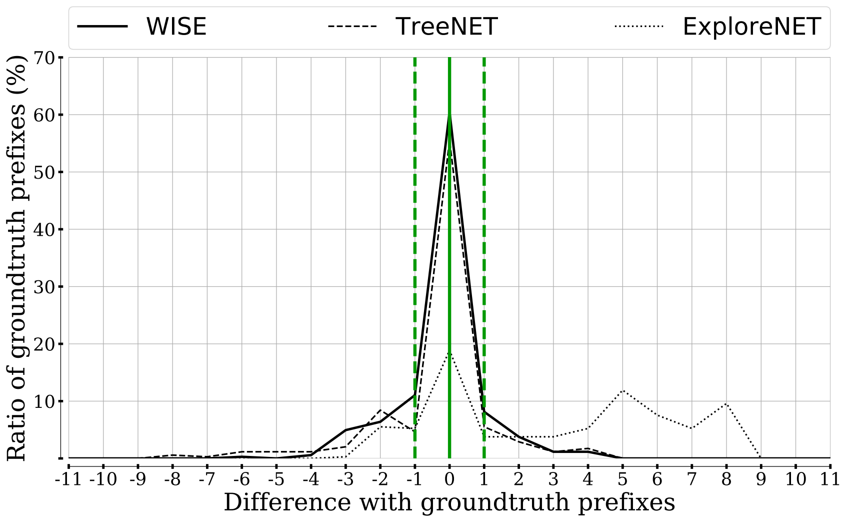
It should be noted that the validation of WISE (along its comparison with other tools) was actually been done twice. The first time was in February 2019 [52], with the first final build of WISE, which had not implemented the overgrowth heuristic at the time (cf. Sec. 7.4.1) and whose post-processing step (cf. Sec. 7.4.2) was still very fundamental. The second time was in September 2019 [53], with a build of WISE true to Chapter 7. The subsequent results will therefore be based on this second validation.

### 8.1.2 Validation results

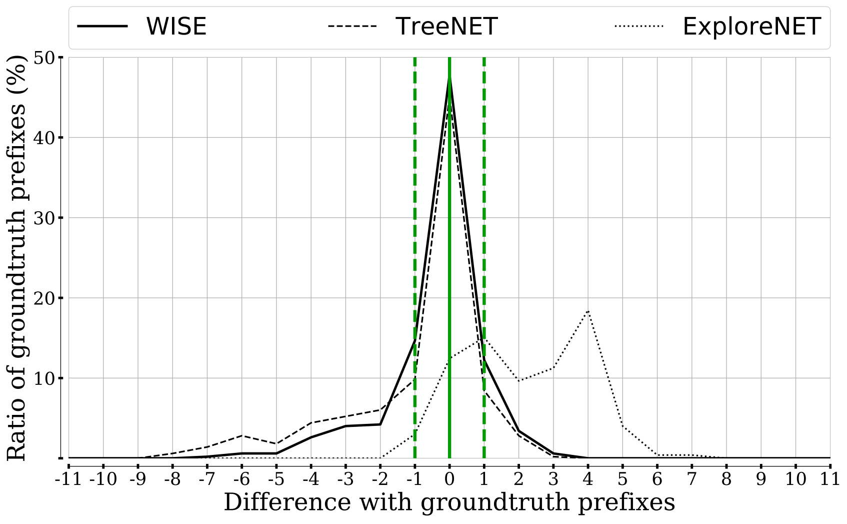
First, to quantify accurately how close an inferred subnet is to a groundtruth, the ***subnet distance*** metric will be defined. This metric is computed as follows: for a given subnet in the groundtruth, an inferred subnet whose prefix is – partially or entirely – overlapping the groundtruth prefix has to be found, and the difference in bits between the inferred prefix length and the groundtruth prefix length is computed. E.g., if there is a groundtruth subnet `10.0.1.0/24` and an inferred subnet `10.0.1.128/25` that overlaps it, the subnet distance will be equal to $25 - 24 = 1$. As a consequence, a subnet distance of zero corresponds to a perfect inference, i.e., the discovered prefix perfectly fits

[1] Again, the details of each groundtruth network will not be made public for confidentiality and security concerns.
[2] From the former WiFi network *ULg-Secure*.

the groundtruth. Overgrown subnets (i.e., that are larger in size than groundtruth subnets) typically produce negative subnet distances, while undergrown subnets (i.e., which cover less attributable IPs than the corresponding groundtruth subnet) result in positive distances. When a groundtruth subnet is overlapped by several (smaller) inferred subnets, the subnet distance can be computed on the basis of the largest inferred subnet. E.g., if a groundtruth prefix /23 is overlapped by one inferred /24 subnet and two inferred /25 subnets, the subnet distance is equal to $24 - 23 = 1$, as the /24 subnet is the largest inferred subnet.



(a) ULiège network (coverage: 98% / 86% / 81%)



(b) Belgian ISP (coverage: 91% / 89% / 75%)

Figure 8.1: Subnet distances for WISE, TreeNET and ExploreNET (September 2019).

It is worth noting that positive subnet distances (i.e., undergrown subnets) are to be preferred for two reasons. First, the lack of responsive IP interfaces within the address space of a given subnet can prevent the inference of its true prefix length, and therefore, an undergrown subnet can still be true to the actual topology. For instance, a /24 subnet where only IP addresses in the first half are being used in practice will be typically detected as a /25 subnet by WISE. On the other hand, overgrown subnets typically cover multiple actual subnets and therefore can be hardly considered as sound: at best, they can be interpreted as a coarse-grained inference delimiting subnets close to each others in the address space. Second, undergrown subnets that are consecutive in respect of the address space can be compared and post-processed to recover the actual prefix(es), as already performed to some extent by the subnet post-processing implemented in WISE (cf. Sec. 7.4.2).

The subnet distance metric can be complemented with a second metric called ***coverage ratio***. This ratio simply expresses the number of groundtruth prefixes that are overlapped by inferred subnets divided by the total number of groundtruth prefixes. While the coverage ratio does not indicate whether or not the inferred subnets are accurate, a high ratio means that the subnet inference has managed to discover exhaustively the subnet-level of the target network, while a low ratio is synonymous of a pessimistic inference or a responsivity issue (e.g., due to rate-limiting policies).

If the ratios of groundtruth subnets overlapped by inferred subnets (Y axis) are plotted for each subnet distance (X axis), the distribution should ideally be centered around 0 (i.e., perfectly inferred subnets), suggesting the inferred subnets are true to the groundtruth. Figure 8.1 gives the subnet distance distributions of the data collected with WISE, TreeNET and ExploreNET, respectively, on both the groundtruth networks mentioned in Sec. 8.1.1. Both Fig. 8.1a and Fig. 8.1b are annotated with the coverage ratios for subnets discovered respectively by WISE (plain line), TreeNET (dashed line) and ExploreNET (dotted line). The green vertical line at subnet distance 0 is a marker for the perfect situation. Additional green dotted lines at subnet distances -1 and 1 highlight the inferred prefixes differing from the groundtruth by at most one bit.

Generally speaking, both Fig. 8.1a and Fig. 8.1b show that WISE is able to reveal more (nearly) exact prefixes than both TreeNET and ExploreNET, going up to 60% of exact prefixes on the ULiège network. WISE also tends to infer more undergrown subnets (positive subnet distance) than TreeNET as well as smaller prefix lengths for overgrown subnets. Moreover, the coverage ratio achieved by WISE was better in both situations, especially with the ULiège network. Such a result can be partially explained by the fact that ExploreNET sometimes discards small subnets due to a lack of responsive IP addresses within small prefixes. Since TreeNET refines subnets initially inferred by ExploreNET to more easily identify large subnets, its coverage can improve but sometimes at the cost of more overgrown subnets, especially in the case of the Belgian ISP. It should also be noted that, despite being less accurate than WISE, TreeNET clearly improves the inference performed by ExploreNET as mentioned in Sec. 5.2.3: both subnet distance distributions for ExploreNET imply a large number of undergrown subnets due to poorly identifying larger subnets.

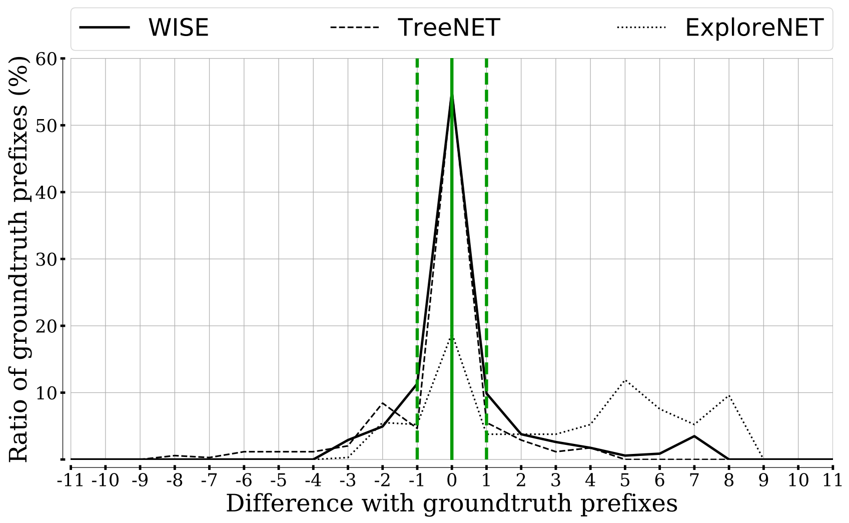
WISE not only achieves greater accuracy than both ExploreNET and TreeNET, but its methodology also renders it more efficient. Table 8.1 compares the runtime and number of probes used by both WISE and the TreeNET subnet inference included in the early build of SAGE [47] (cf. Sec. 8.1.1). The provided metrics only cover the subnet inference steps in both cases, i.e., including the target scanning step of WISE (cf. Sec. 7.3.2) but excluding the pre-scanning (cf. Sec. 7.3.1). In both cases, WISE outperforms the methodology of TreeNET in terms of runtime, as this runtime as reduced by half to discover subnets within the ULiège network and by twenty to perform the same task on the Belgian ISP. Its number of probes was however in the same order of magnitude as TreeNET: this can be explained by the fact that the target scanning probed the responsive IP addresses at a steady rate, while TreeNET (ExploreNET) probed interfaces as it inferred subnets, eliminating target addresses when it discovered large subnets. Interestingly, TreeNET used fewer probes in total than WISE for the ULiège network: this was due to the presence of many large subnets (i.e., /23, /24, /25, etc.) with which TreeNET could eliminate many subsequent target IP addresses. However, there were not as many large subnets in the Belgian ISP, which explains why the total runtime of TreeNET on this network was so much longer when compared to the WISE runtime. This clearly demonstrates that decoupling the subnet inference from the probing is highly benefitial to shorten the runtime.

| | | TreeNET (via early SAGE) | WISE |
|---|---|---|---|
| **ULiège network** | Execution time | 52'30 | 28'34 |
| | Number of probes | 18,065 | 26,175 |
| **Belgian ISP** | Execution time | 9:51'10 | 24'14 |
| | Number of probes | 277,819 | 213,840 |

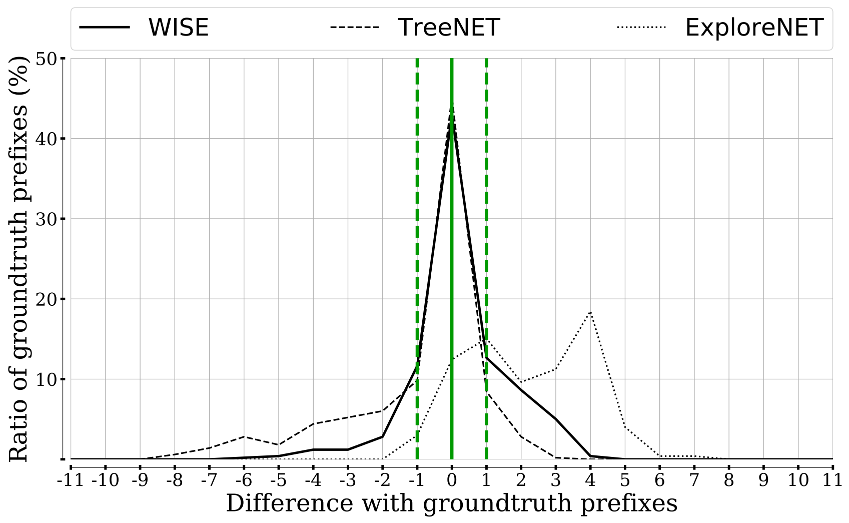
Table 8.1: Execution time and network use of TreeNET and WISE (September 2019).

It should be noted, though, that the steady probing of WISE triggered rate-limiting policies at several routers in the ULiège network. As a consequence, a considerable number of IP interfaces (around 44.5% of the responsive IP addresses) had to be re-probed to maximize the number of anomaly-free trails (cf. end of Sec. 7.3.2). This is why the WISE runtime on the ULiège network was slightly worse than on the Belgian ISP: in comparison, very few timeouts occurred while scanning the responsive IP addresses of the ISP. Despite this issue, WISE still outperformed TreeNET, demonstrating further its methodology is interesting regarding execution time and network use.

To have an even more complete analysis, the results when considering the subnet prefixes adjusted by WISE at the end of post-processing (cf. end of Sec. 7.4.2) will be presented. This meant computing, for each inferred subnet, the longuest prefix length that included all discovered subnet interfaces. Indeed, in some situations, the lack of responsive subnet interfaces meant WISE continued expanding a subnet until just before overlapping previous results. Such a growth can produce an overgrown subnet as well as an accurate prefix. Figure 8.1 used these unadjusted, *raw* prefixes, while Figure 8.2 shows the validation results upon considering the adjusted prefixes. Interestingly, this cause the ratio of exact prefixes discovered by WISE to fall slightly below the same ratio for TreeNET. However, when considering prefixes differing by at most one bit, WISE still outperformed both TreeNET and ExploreNET, and the curve for WISE also slightly moves to the right, which is synonymous of having more undergrown and less overgrown subnets. This shift could be an advantage, if the final subnets were to be post-processed further by a third party (e.g., by identifying the prefix assignment practices of the target network). Nevertheless, the *raw* prefixes can be kept *as is* since they can still match actual subnets accurately thanks to the boundaries of other subnets.



(a) ULiège network (coverage: 98% / 86% / 81%)



(b) Belgian ISP (coverage: 91% / 89% / 75%)

Figure 8.2: Same validation as in Fig. 8.1, using WISE adjusted prefixes.

Finally, though the subnets as inferred by WISE, TreeNET and ExploreNET on both groundtruth networks clearly suggest WISE is the better option, it should be noted that there was little traffic engineering in both cases. In particular, some flickering trails were witnessed in both networks but concerned only a small number of subnets: a /26 subnet in the Belgian ISP that featured two flickering trails could be properly discovered by WISE but was first discovered as a /31 and a /29 subnets by ExploreNET and later (over)grown into a /23 subnet by TreeNET. Outside this example, only one other subnet in the Belgian ISP exhibited flickering trails. In other words, both groundtruth networks were suitable for the methodologies of TreeNET and ExploreNET, which is why the former closely follows WISE in terms of perfectly inferred subnets. WISE is nevertheless expected to more frequently discover the same subnets when traffic engineering induces more consequences. This point will be discussed in Sec. 8.2.3 with the help of snapshots collected in the wild.

## 8.2 `WISE` in the wild

This section discusses observations made in the wild from the PlanetLab testbed with `WISE` during the course of 2019. Sec. 8.2.1 first briefly discusses how and when `WISE` was deployed. Then, Sec. 8.2.2 evaluates the soundness of the discovered subnets in respect of the measurement-based definition (cf. Sec. 5.1.2), Sec. 8.2.3 discusses the subnet persistence, i.e., whether the same prefixes can be inferred despite a change of vantage point, and finally, Sec. 8.2.4 discusses the distribution of the prefix length among inferred subnets.

### 8.2.1 `WISE` deployment

The first final version of `WISE` was deployed on the PlanetLab testbed on December 28th, 2018 in order to collect subnet-level data on the same target ASes as those mentioned in Sec. 6.2.1. This first campaign ended on January 24th, 2019 and also involved vantage point rotation (as defined in Sec. 6.2.1): for each of the 22 target ASes, there were 22 snapshots (minus one or two, due to a lack of responses to probes) collected from each of the 22 PlanetLab nodes used during the campaign. Not only was this strategy aimed at making the most of the available PlanetLab nodes, but it also allowed thoroughly studying the effects of changing the vantage point while running `WISE` [52].

This first campaign was however not the last, and a total of ten campaigns, all involving vantage point rotation, were carried out in 2019 [53]. In particular, the last four campaigns involved the lattest version of `WISE` subnet inference, i.e., featuring all the mechanisms and heuristics described or mentioned in Chapter 7. Subsequent sections will therefore provide observations about the snapshots collected during the seventh campaign (September 2019), but the details about all the campaigns and their snapshots can be browsed online via the `WISE` public GitHub repository [3].

| ASN | Name | Origin | Category |
|---|---|---|---|
| 12956 | Telefonica Global Solutions | Spain | Tier-1 |
| 3257 | GTT Communications Inc. | USA | Tier-1 |
| 6453 | TATA Communications | USA | Tier-1 |
| 6762 | Sparkle | Italy | Tier-1 |
| 10010 | TOKAI Communications | Japan | Transit |
| 13789 | Internap Corporation | USA | Transit |
| 286 | KPN B.V. | Netherlands | Transit |
| 50673 | Serverius Holding B.V. | Netherlands | Transit |
| 6939 | Hurricane Electric LLC | USA | Transit |
| 8220 | COLT Technology | UK | Transit |
| 8928 | Interoute Communications | UK | Transit |
| 1241 | Forthnet | Greece | Stub |
| 224 | UNINETT | Norway | Stub |

Table 8.2: Summary of the 13 Autonomous Systems probed with `WISE` in September 2019.

[3]`https://github.com/JefGrailet/WISE/tree/master/Dataset`

During the seventh campaign of WISE, a total of 13 ASes were measured exclusively from European PlanetLab nodes [4]. Much like with measurement campaigns previously discussed in this thesis (cf. Sec. 4.3.1 and Sec. 6.2.1), the target ASes were selected as a representative sample of the Internet, notably using CAIDA's AS ranking [2] to select ASes that played a major role in the Internet and the Hurricane Electric BGP Toolkit [4] to retrieve their IPv4 prefixes. They also featured at most a few millions of attributable IP addresses so that WISE could discover their subnet-level in the span of a few hours. Table 8.2 lists the 13 ASes measured from September 4th to September 17th, 2019.

### 8.2.2 Subnet soundness in the wild

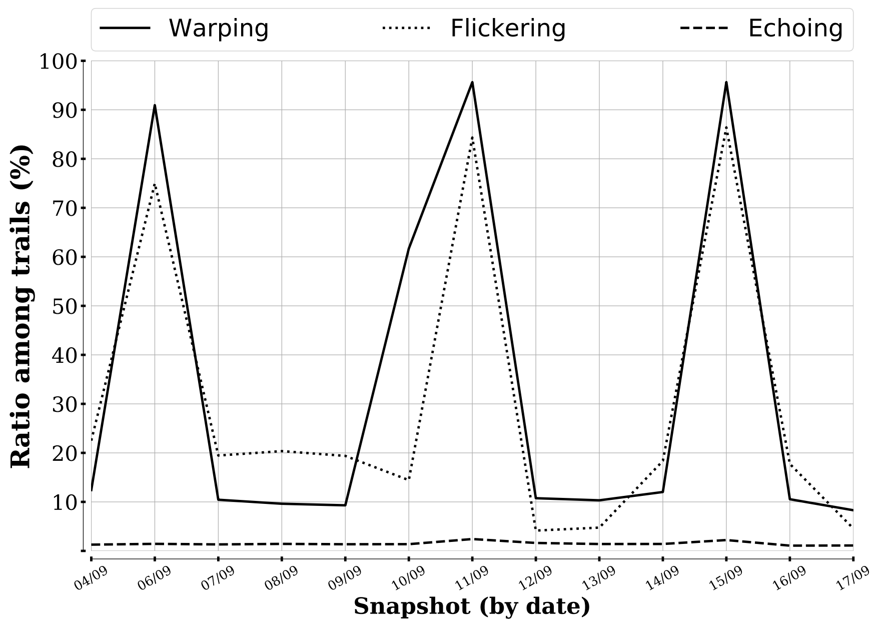
In the absence of a groundtruth to assess the soundness of inferred subnets, criteria needs to be defined to determine whether a given subnet looks sound or atypical. Ideally, a subnet should be as close as possible to the ideal measurement-based definition used by state-of-the-art tools (cf. Sec. 5.1.2), although some room should be made for situations where TTL distances towards pivot interfaces are not all equal. To satisfy these requirements, three rules have been defined to quantify subnet soundness: the ***contra-pivot rule***, the ***spread rule***, and the ***outlier rule***.

The contra-pivot rule states that an ideal subnet should feature at least one contra-pivot interface, as a subnet with none could very well be a part of a larger subnet. The spread rule states that, in the presence of contra-pivot interfaces, these interfaces are either in the minority for large subnets, or no more common than pivot interfaces for small subnets (i.e., the prefix length 29 or greater). Finally, a subnet fulfilling the outlier rule is a subnet containing no interfaces other than pivots and contra-pivots. A subnet satisfying the three rules is considered as sound, and if the TTL distances of the pivot interface(s) are identical while the contra-pivot interface(s) is (are) found exactly one hop sooner, then the subnet strictly satisfies the ideal measurement-based definition used by previous state-of-the-art subnet inference tools.

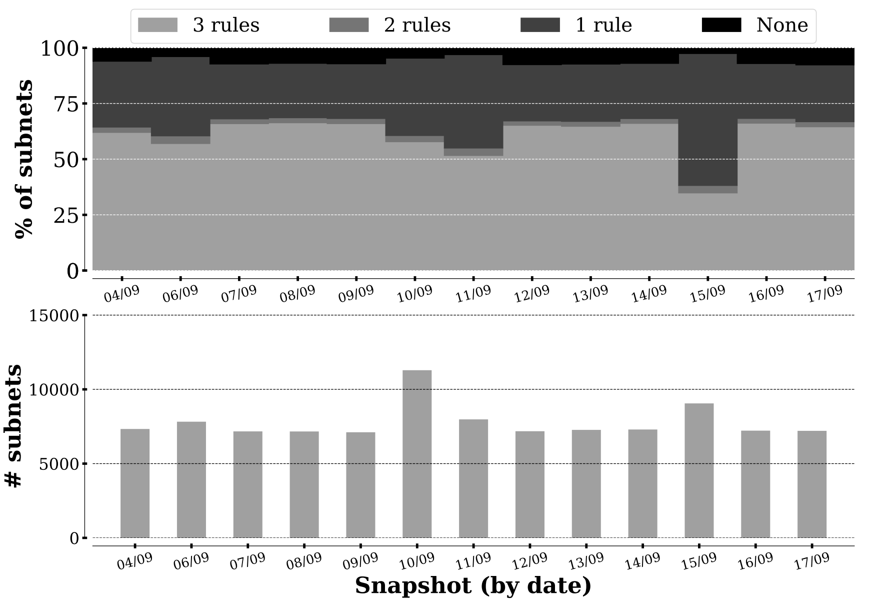
It is worth noting that revealing a given subnet prefix from different vantage points does not mean that the prefixes will exactly meet the same soundness rules. For instance, it is always possible that the subnet revealed from vantage point $X$ could feature an outlier, while the same subnet revealed from vantage point $Y$ will exhibit highly varying distances, making the contra-pivot interface detection difficult (which a final heuristic of post-processing, mentioned in Sec. 7.4.2, attempts to achieve). The three aforementioned rules are therefore also a good indicator of how changing the vantage point (and, consequently, the different paths towards the targeted domain) can impact the inference of a given subnet, notably due to traffic engineering (cf. Chapter 6). Due to the large number of snapshots collected during 2019, this section will only provide figures with representative results, but all figures regarding subnet soundness can be browsed online via the WISE public GitHub repository [5].

Figure 8.3 pictures the subnet soundness for AS6453 (TATA Communications, Inc.; Tier-1), using 13 snapshots collected with WISE from the PlanetLab testbed between September 4th and September

[4]Or PLE nodes, for **P**lanet**L**ab **E**urope; International nodes (PLC, for **P**lanet**L**ab **C**entral) were barely usable at the time.
[5]`https://github.com/JefGrailet/WISE/tree/master/Evaluation/Accuracy/Figures`



(a) Problematic trails



(b) Subnet soundness rules (top) and numbers of subnets (bottom)

Figure 8.3: Subnet soundness for AS6453 (September 4th to September 17th, 2019).

17th, 2019 (included). The figure is split into two: Fig. 8.3a plots the ratios of problematic trails observed during the campaign, in the exact same manner as in Sec. 6.2.2, while Fig. 8.3b provides both the total number of subnets found in each snapshot (bottom) and the ratios of subnets that satisfied zero, one, two or three soundness rules. By putting both figures side by side, it is possible to verify whether or not the variations in problematic trails impacted subnet inference. In the case of AS6453, the ratios of subnets fulfilling three or two rules were steady, as they represented more than

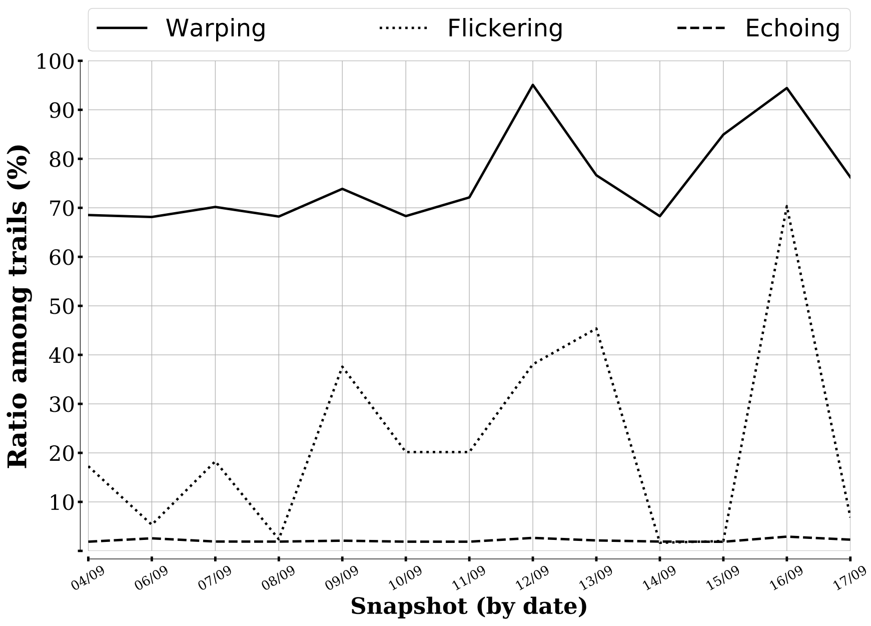
50% and often more than 60% of all inferred subnets, with the exception of one snapshot captured on September 15th, 2019. Interestingly, this date also coincided with the second snapshot to have the largest number of inferred subnets as well as the third spike in warping/flickering trails depicted in Fig. 8.3a. The two other spikes in Fig. 8.3a (September 6th and September 11th) coincided with small drops in the ratio of sound subnets, though the corresponding ratios still featured clearly more than 50% of sound-looking subnets. The snapshot with the largest number of subnets (September 10th) was also satisfying in terms of soundness, which can be explained by the fact that there were comparatively less warping trails and much less flickering trails than for September 15th. Finally, in all cases, the ratio of subnets satisfying no rule at all (i.e., subnets with no contra-pivot interface and one or more outliers) was low, as shown by the thin black bars at the top of Fig. 8.3b. Overall, this first figure shows that `WISE` adapted well to vantage point changes and produced few atypical subnets.

Similar conclusions can be drawn from Figure 8.4, which provides the same visual tools as Figure 8.3 but using the 13 snapshots collected with `WISE` on AS3257 (GTT Communications, Inc.; Tier-1) during September 2019. AS3257 is an interesting case to compare with AS6453 because it is also a Tier-1 AS, but covering much more attributable IP addresses (more than 2 millions) and with more systematic traffic engineering, as demonstrated by the steadily high ratio of warping trails pictured in Fig. 8.4a. Despite these challenges, the ratios of sound subnets were always in the region of 50% in the collected snapshots, as shown by Fig. 8.4b, with an exception on September 16th. This exception also coincided with a spike in flickering trails in Fig. 8.4a. Moreover, in the same way as in Figure 8.3, the ratios of subnets satisfying no soundness rule were also low, demonstrating again that `WISE` could cope well with vantage point changes (and their effects regarding traffic engineering) while producing a vast majority of non-atypical subnets and a majority of sound subnets.

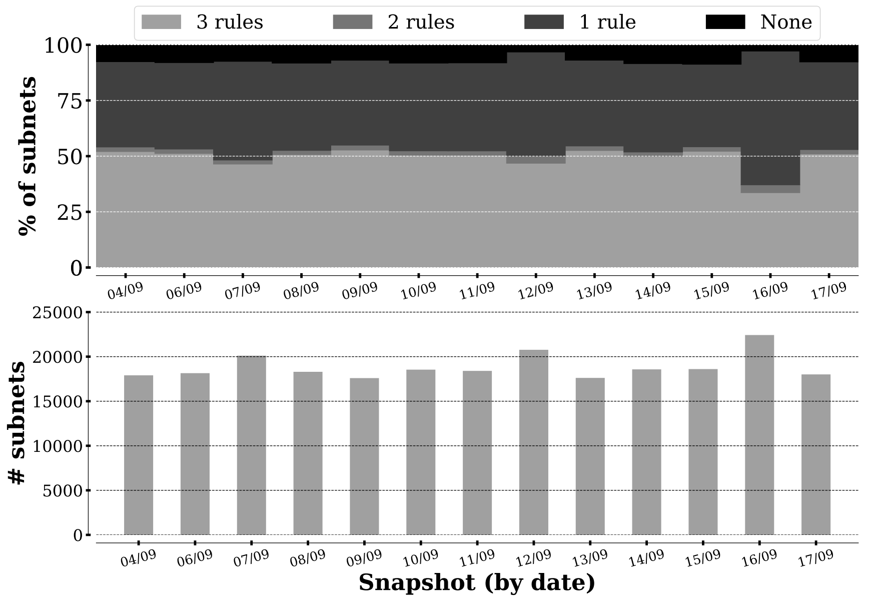
To complement the discussion surrounding Figure 8.3 and Figure 8.4, since they both involve Tier-1 ASes, the observations made on a Transit AS and a Stub AS, respectively, will be discussed.

Figure 8.5 provides the same visual tools as before to evaluate the soundness of subnets discovered in AS286 (KPN B.V.), a Transit AS from the Netherlands, again using 13 snapshots collected with `WISE` from PlanetLab during September 2019. Once more, `WISE` was able to achieve a majority of sound subnets with a low ratio of atypical results (i.e., the top black bar in Fig. 8.5b) in all cases, but with more contrasts than in previous figures. The first and last snapshots (i.e., September 4th and September 17th) notably featured the lowest ratios of sound subnets as well as the largest numbers of discovered subnets, and coincided with spikes in all types of problematic trails. Interestingly, the number of responsive IP addresses in each snapshot was quasi constant, which means the consequences of the vantage point change was that subnet interfaces appeared differently to probes. In particular, the unusually varying ratios of echoing trails shown by Fig. 8.5a suggested many interfaces could not be reached via the same ingress router depending on the vantage point. This was the major difference in respect of Figure 8.3 and Figure 8.4, since they both displayed consistently low ratios of echoing trails.

Interestingly, the highest ratios of sound subnets for AS286 (i.e., almost 70%) coincided with the snapshots whose ratios of echoing and warping trails were the lowest (in particular, from September



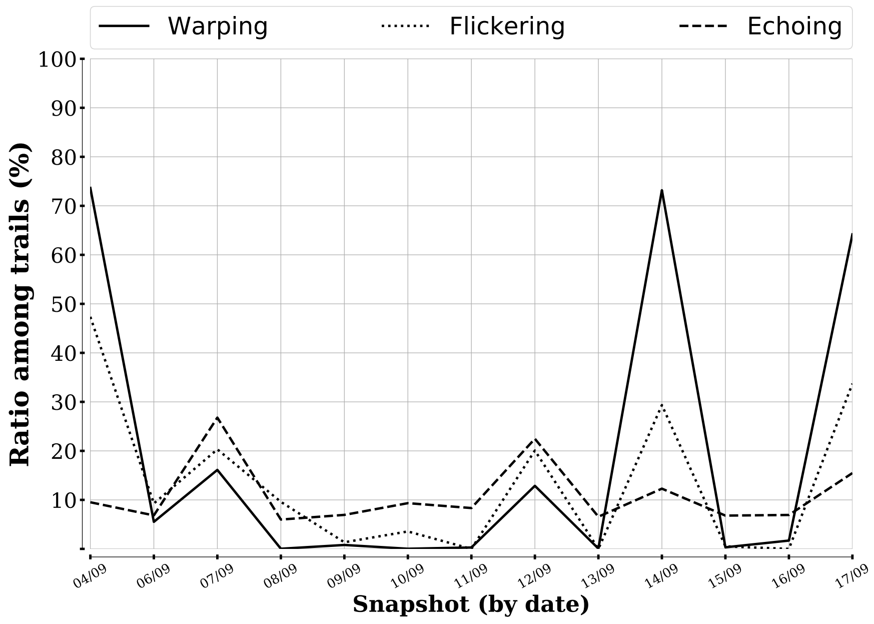
(a) Problematic trails



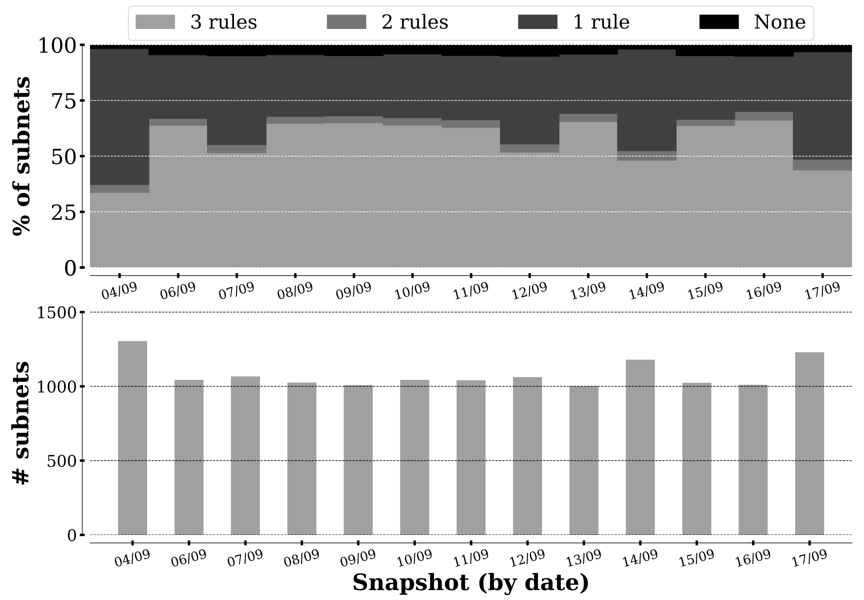
(b) Subnet soundness rules (top) and numbers of subnets (bottom)

Figure 8.4: Subnet soundness for AS3257 (September 4th to September 17th, 2019).

8th to September 11th). On closer inspection, the data showed that contiguous blocks of IP addresses with echoing trails did not have the same properties in terms of TTL distances depending on the vantage point, i.e., in some snapshots the TTL distance was always the same for each IP address of a given block, but in other snapshots the distances for the same block varied. The first situation allowed WISE to infer subnets with the help of the third rule of inference (cf. Sec. 7.4.1), while the latter prevented the application of the same rule. Unfortunately, by definition, echoing trails are



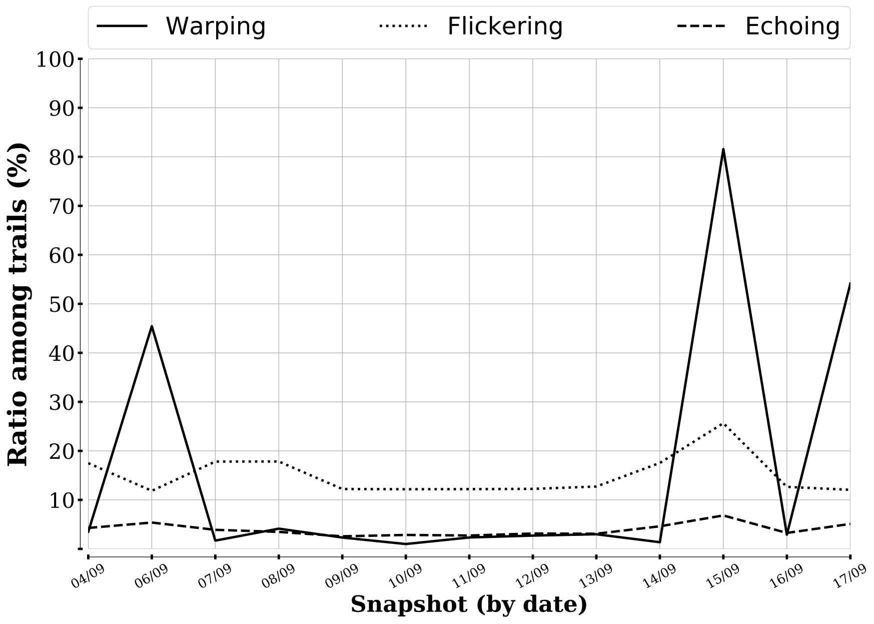
(a) Problematic trails



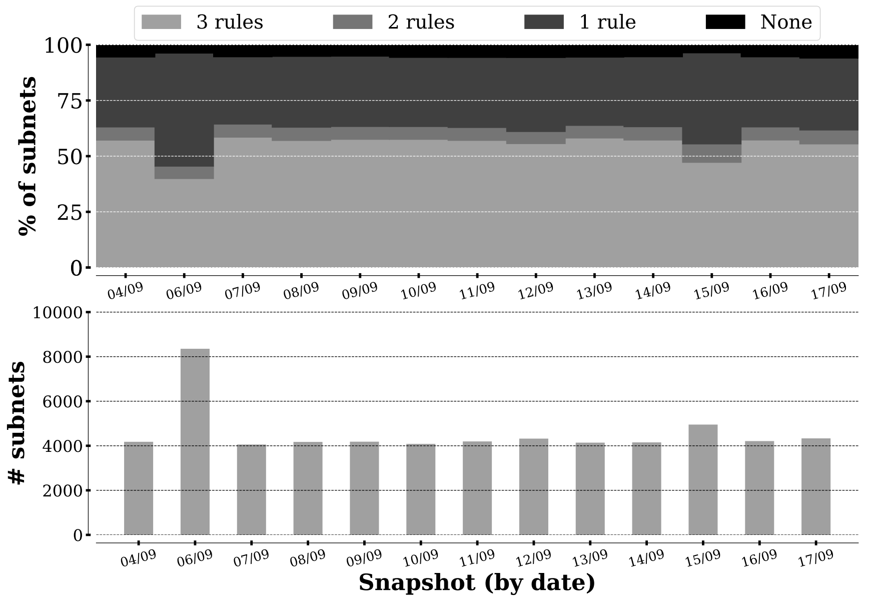
(b) Subnet soundness rules (top) and numbers of subnets (bottom)

Figure 8.5: Subnet soundness for AS286 (September 4th to September 17th, 2019).

always unique (cf. Sec. 6.1.4), so they cannot be identified as warping at the same time since there is no other trail to compare them to. Therefore, WISE was unable to identify "*warping echoing*" trails, and currently offers no practical solution to perform subnet inference with IP addresses featuring such trails. A practical workaround could consist of combining snapshots from multiple vantage points in order to keep those where echoing and warping do not happen at the same time, in a way similar to the *Cheleby* system [70] (cf. Sec. 5.2.4).



(a) Problematic trails



(b) Subnet soundness rules (top) and numbers of subnets (bottom)

Figure 8.6: Subnet soundness for AS224 (September 4th to September 17th, 2019).

Finally, Figure 8.6 provides the usual visual tools to assess the soundness of subnets discovered in AS224 (UNINETT), a Norwegian Stub AS, based on 13 snapshots captured in the same conditions as before. This time, the results were both satisfying and consistent across all snapshots with the exception of two snapshots, respectively captured on September 6th and September 15th, which coincided with spikes in the ratio of warping trails. Interestingly, the variation in the flickering trails did not appear to have a major effect on subnet soundness. The final snapshot (September 17th) also coincided with a spike in warping trails despite providing a high ratio of sound subnets. What was also interesting was that the ratio of warping trails was higher in the snapshot captured on September 15th than in that collected on September 6th, although the former provided more sound subnets than the latter. Closer inspection of the warping trails of both snapshots revealed that the observed TTL distances and their differences varied more in the snapshot captured on September 6th. Likewise, TTL distances for warping trails in the snapshot captured on September 17th were surprisingly regular, with many warping trails featuring two TTL distances with a difference of two hops between them. This regularity likely eased the subnet inference, while more variety in TTL distances could have further worsened contra-pivot discovery and induced `WISE` to mistaking contra-pivots for outliers more often than it would otherwise have done. In other words, `WISE` is able to effectively handle a vantage point change and more generally traffic engineering, but can still struggle if asymmetrical paths are more diverse and exhibit greater differences in length.

### 8.2.3 Subnet persistence

To further assess the extent to which `WISE` discovers the same subnets on a given target AS regardless of the vantage point, the notion of ***subnet persistence*** needs to be defined: given two snapshots of a given target network, a subnet is said to be persistent if its prefix is present in both snapshots. This notion is sub-divided into ***weak persistence*** and ***strict persistence***. Given two snapshots of a given target network, a subnet is said to be strictly persistent if its prefix is found in both snapshots and if the corresponding subnet perfectly satisfies the same soundness rules (as defined in Sec. 8.2.2. As a consequence, a subnet will be weakly persistent if its properties in two given snapshots are such that it does not satisfy the same rules in both cases. The detection of a weakly persistent subnet leads to two conclusions. First, changing the vantage point from one run of `WISE` to another can lead to additional difficulties, such as those described by Chapter 6. Second, despite these additional challenges induced by a vantage point change, `WISE` can re-discover the same subnet prefixes.

The subnet persistence in the same ASes as discussed in Sec. 8.2.2 will be examined. In the same way as before, additional figures for other ASes and other campaigns can be browsed online via the `WISE` public GitHub repository [6], with one minor difference: figures in this section will also depict the intersection of strictly persistent subnets, i.e., the number of subnets that reappear strictly in the same manner in all snapshots of a given campaign.

Figure 8.7 depicts the strict and weak persistence of subnets discovered for AS6453 (TATA Communications, Inc.), based on its snapshots captured in September 2019. In this figure, all the snapshots except for the first one (X axis) were compared to the very first snapshot, captured on September 4th, that will be subsequently denoted as the *reference* snapshot. The light grey bars quantify the strictly persistent subnets while the dark grey bars account for weakly persistent subnets. The dotted bars

[6]Cf. `https://github.com/JefGrailet/WISE/tree/master/Evaluation/Persistence/Figures`

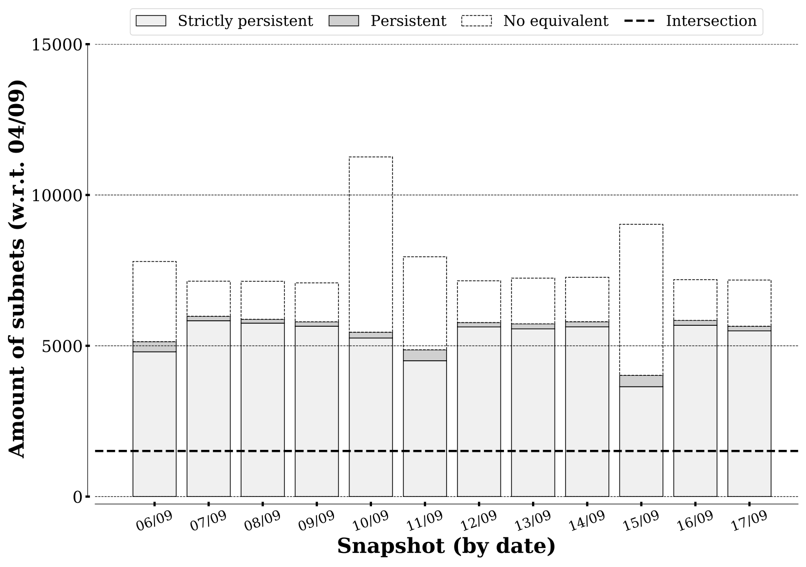


Figure 8.7: Subnet persistence for AS6453 (September 4th to September 17th, 2019).

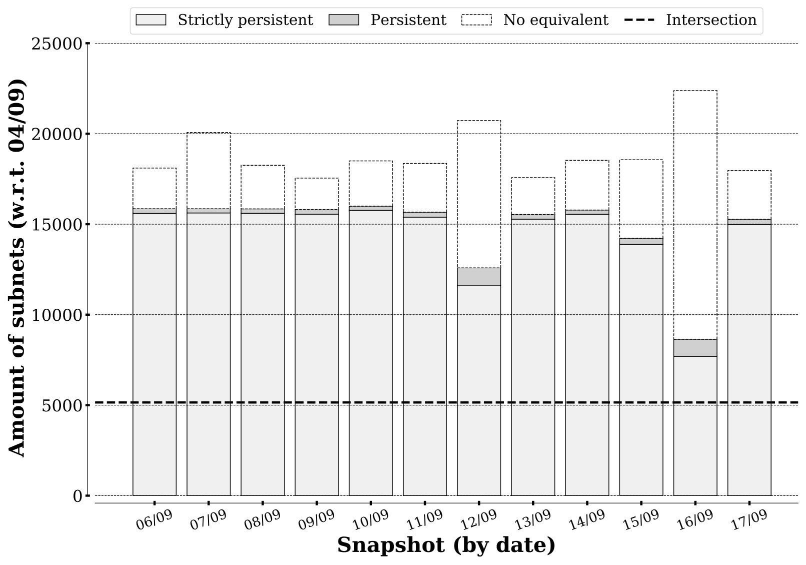
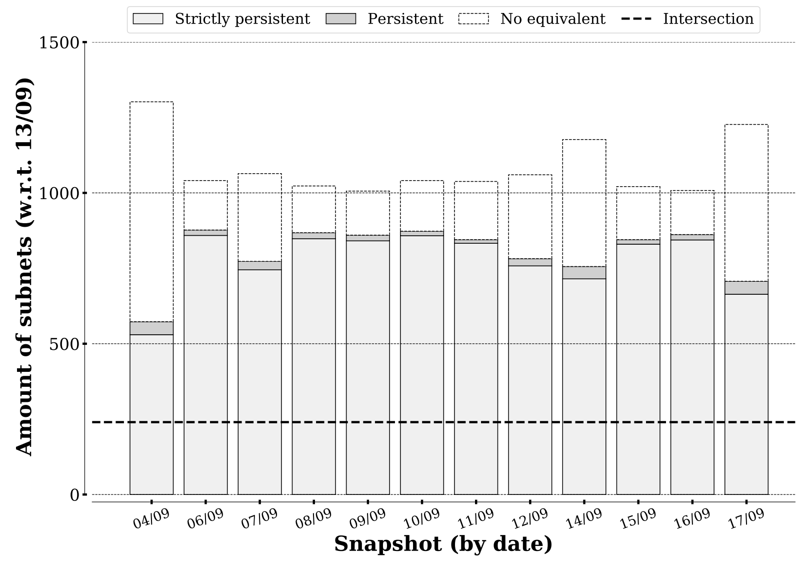


Figure 8.8: Subnet persistence for AS3257 (September 4th to September 17th, 2019).

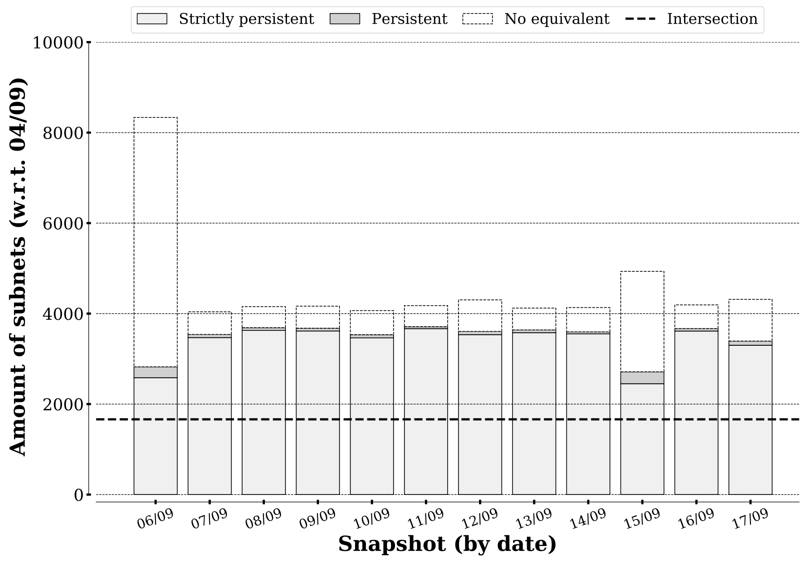
quantify the remaining subnets (i.e., no persistence), and finally, the dotted black horizontal line measures the intersection of strictly persistent subnets.

This first figure shows the subnet persistence was quite high for each pair of snapshots: with the exception of September 10th and September 15th, the parts of each stacked bar that accounted for persistent subnets were clearly the dominant parts. Curiously, the intersection across all snapshots only accounted for slightly more than a thousand subnets. This could be, however, explained by the fact that the intersection also included the snapshots of September 10th and September 15th, as will be detailed shortly. Interestingly, each snapshot provided a noticeable number of weakly persistent subnets in addition to a small number of unique subnets with respect to the reference snapshot.

Figure 8.8 depicts the subnet persistence observed among snapshots of AS3257 (GTT Communications, Inc.) captured in September 2019 in the same manner as Figure 8.7. Again, the parts of the stacked bars depicting persistent subnets were clearly dominant with a few exceptions, dated on September 12th and September 16th in this case. The intersection made up for more than one quarter of all subnets in most snapshots, though it can be inferred that the two previously mentioned snapshots were lowering this intersection in practice.



(a) AS286 (KPN B.V.)



(b) AS224 (UNINETT)

Figure 8.9: Subnet persistence for a Transit and a Stub ASes (September 2019).

Finally, Figure 8.9 depicts the subnet persistence observed in one Transit AS (AS286, Fig. 8.9a) and one Stub AS (AS224, Fig. 8.9b), respectively. The same observations as before can be made:

a high subnet persistence in most snapshots, a few snapshots with less persistence, and finally an intersection that is clearly visible but that appears to be low at first sight, given the number of snapshots with high persistence in respect of the reference snapshot. Note, however, how the reference snapshot for AS286 was not the first to be captured chronologically speaking: this choice was made due to the September 4th snapshot not featuring as many sound subnets as others, as a result of traffic engineering and a combination of warping and echoing (cf. Fig. 8.5).

All previous figures tend to demonstrate that `WISE` could recover the same subnets in most cases, despite changing the vantage point and despite the variations in problematic trails shown in figures from Sec. 8.2.2. Surprisingly, though, the intersection in all figures appeared to be underwhelming: roughly one quarter of the inferred subnets in most cases. However, as previously stated, the few snapshots for each AS that included more unique subnets (because there was not as much persistence) were behind these underwhelming intersections. To demonstrate this, Table 8.3 provides the total number of strictly persistent subnets across all snapshots for each of the ASes discussed in this section leaving out the two least sound snapshots (i.e., least sound subnets).

| **Target AS** | **Omitted snapshots** | **New intersection (# subnets)** |
|---|---|---:|
| AS6453 | September 6th, September 15th | 2,919 |
| AS3257 | September 12th, September 16th | 11,893 |
| AS286 | September 4th, September 17th | 491 |
| AS224 | September 6th, September 15th | 2,965 |

Table 8.3: Better intersections of strictly persistent subnets (September 2019).

In the case of AS3257 and AS224, the number of strictly persistent subnets found across all snapshots became close to the individual numbers of strictly persistent subnets found for each pair of snapshots (i.e., reference snapshot with each of the remaining snapshots). To visualize this, Figure 8.10 provides the same content as Fig. 8.8 without the least sound snapshots and the updated intersection. As for AS6453 and AS286, the intersection was close to half the inferred subnets in most cases. In fact, omitting a third snapshot for AS6453 (September 11th) even increases the intersection to 3,804 subnets, i.e., more than half the number of inferred subnets in the remaining snapshots. These improved intersections show that `WISE` usually discovered the same subnets, though the initial intersections already showed it had such capability, even including the least sound snapshots.

By looking in details at the data, it can be seen, in practice, how `WISE` discovered the same subnets despite how traffic engineering changed how they appeared to probes. This can notably be seen with the subnet `129.242.88.0/21` that `WISE` regularly discovered within AS224. In most snapshots collected in September 2019 for AS224, this subnet appeared with regular TTL distances for pivot interfaces. Example 8.1 shows the first interfaces listed for this subnet, using the snapshot collected on September 4th. However, on September 15th, the TTL distances for the same interfaces varied, i.e., the pivot TTL distance was either 20 or 21 hops. Despite this, `WISE` was still able to infer `129.242.88.0/21`, notably thanks to the trail consistency, as shown by Example 8.2.

More persistent subnets and how they appear depending on the snapshot can be investigated

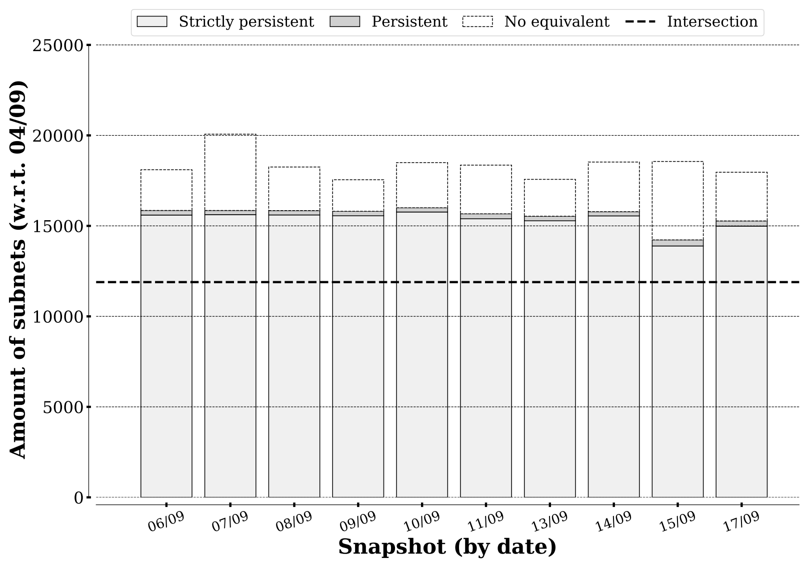


Figure 8.10: Subnet persistence for AS3257 (omitting September 12th and September 16th).

with the help of Python scripts provided by the `WISE` public GitHub repository [7].

Listing 8.1: `129.242.88.0/21` (AS224) on September 4th (first lines)

```
129.242.88.0/21
17 - 129.242.88.1 - [158.39.1.74] - Rule 1
16 - 129.242.88.2 - [128.39.230.44] - Contra-pivot
17 - 129.242.88.3 - [158.39.1.74] - Rule 1
17 - 129.242.88.5 - [158.39.1.74] - Rule 1
17 - 129.242.88.23 - [158.39.1.74] - Rule 1
17 - 129.242.88.40 - [158.39.1.74] - Rule 1
17 - 129.242.88.56 - [158.39.1.74] - Rule 1
```

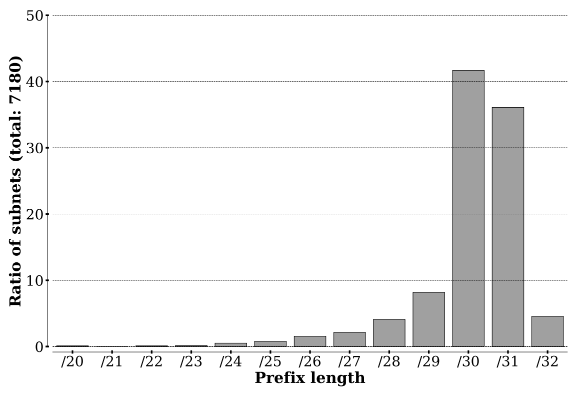
Listing 8.2: `129.242.88.0/21` (AS224) on September 15th (first lines)

```
129.242.88.0/21
20 - 129.242.88.1 - [158.39.1.74] - Rule 1
19 - 129.242.88.2 - [128.39.230.44] - Contra-pivot
20 - 129.242.88.3 - [158.39.1.74] - Rule 1
21 - 129.242.88.5 - [158.39.1.74] - Rule 1
20 - 129.242.88.26 - [158.39.1.74] - Rule 1
20 - 129.242.88.27 - [158.39.1.74] - Rule 1
20 - 129.242.88.40 - [158.39.1.74] - Rule 1
21 - 129.242.88.54 - [158.39.1.74] - Rule 1
```

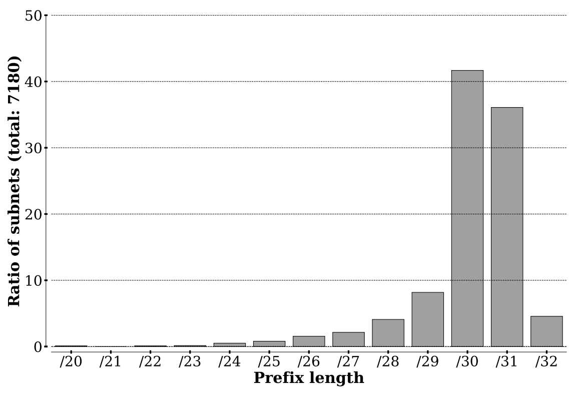
[7] `https://github.com/JefGrailet/WISE/tree/master/Evaluation/Persistence`

### 8.2.4 Subnet prefix length distribution

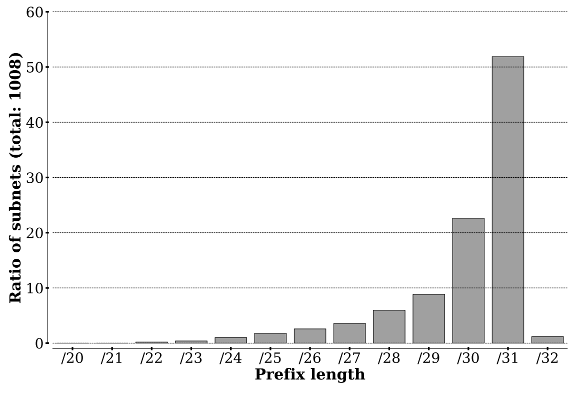
Finally, the prefix length distribution of the subnets discovered by `WISE` will be reviewed, (again) using snapshots collected in September 2019 for the ASes discussed in Sec. 8.2.2 and Sec. 8.2.3. Previous work in subnet inference highlighted the fact that the distribution of prefix lengths tended to follow a power law shape, with the exception of /24 subnets [118] (see also Chapter 5). To study such a distribution for each selected AS, only the last snapshot was used to compute a distribution, with the exception of AS286 where the penultimate snapshot (September 16th) was be used instead, as it was more representative of all the snapshots collected for this target AS. Figure 8.11 depicts the typical subnet prefix length distribution obtained on all four previously discussed ASes.



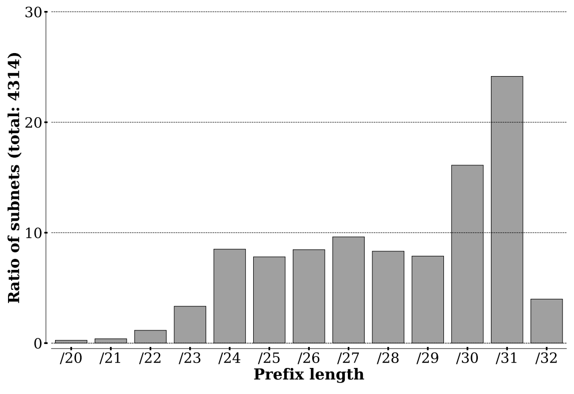
(a) AS6453 (TATA Com.), September 17th, 2019



(b) AS3257 (GTT Com.), September 17th, 2019



(c) AS286 (KPN B.V.), September 16th, 2019



(d) AS224 (UNINETT), September 17th, 2019

Figure 8.11: Subnet prefix length distribution for various Autonomous Systems.

Fig. 8.11a shows the typical distribution obtained on AS6453, using the snapshot captured on September 17th, 2019. This distribution does follow a power law shape, but only up to the /30 prefix: there is indeed slightly less /31 subnets. The few /32 subnets are either loopback interfaces [8], or artefacts. Why there are comparable numbers of /30 and /31 subnets is an interesting question: indeed, conventions in subnetting [92] would tend to make /30 subnets more common, since they would include only two usable interfaces (the subnet prefix and the broadcast address must be

[8]Such subnets exist within both the ULiège network and the backbone of the Belgian ISP mentioned at Sec. 8.1.

omitted), which is the minimum requirement for a point-to-point link. As such, /31 subnets might come off as incomplete measurements. In practice, /31 subnets do exist [9], and might be a way for network operators to economize IP addresses in their IPv4 prefixes. An investigation of this specific practice can be left for future work. Interestingly, /24 subnets are not more common than /23 or /25 subnets (among others) in snapshots of AS6453, contrary to what previous work suggested [118].

For other ASes, however, there is a tendency for /24 subnets to be slightly more frequent than if the distribution was exactly following a power law shape. This was the case for AS3257, as shown by Fig. 8.11b. In this second distribution, the ratio of /24 subnets was comparable to the ratio of /25 subnets (in fact, it was slightly higher), and might be even higher in reality since /24 subnets may have been inferred as longer prefixes by `WISE` in practice (cf. Sec. 7.4). Interestingly, /30 subnets were this time clearly more frequent than /31 subnets, and /32 subnets were a bit more common. Whether this was the result of measurement issues or of specific practices implemented by this AS is something that could be investigated in the future.

Finally, Fig. 8.11c and Fig. 8.11d shows the typical prefix length distribution obtained for AS286 (KPN B.V., Transit) and AS224 (UNINETT, Stub), respectively. These results are even more interesting than the previous ones: Fig. 8.11c shows that the prefix length distribution in AS286 clearly follows a power law shape, even when it comes to /30 and /31 subnets, especially since there are very few /32 subnets. Again, the /24 prefix length is not more common than other lengths. Fig. 8.11d shows a surprising distribution: not only /24 subnets are slightly more common than /25 subnets in the snapshot, but all prefix lengths from /24 to /29 each account for a bit less than 10% of inferred subnets. All observations considered, it becomes difficult to assert whether or not the prefix length distribution within any target network follows a power law shape, with or without a tendency to have more /24 subnets (as suggested by previous research [118]). Moreover, the ratios of /30, /31 and /32 subnets suggest there are very different practices for such prefix lengths depending on the network. Future work regarding the subnet-level could shine light on the specificities of each distribution, and perhaps link them with the type of network where they are found. Again, more figures depicting distributions for other target ASes or other campaigns can be browsed online via the `WISE` public GitHub repository [10].

## 8.3 Closing comments on subnet inference

This second part of the thesis made three major contributions while answering several research questions that were first formulated in Sec. 5.4.

1. The typical consequences of traffic engineering for subnet inference have been characterized and quantified in the wild in Chapter 6 as an anwser to research question 1. It has been demonstrated that these phenomena can vary greatly depending on the vantage point, but are always present, therefore impairing state-of-the-art subnet inference methods.

[9]Some were found in the Belgian ISP discussed at Sec. 8.1.

[10]`https://github.com/JefGrailet/WISE/tree/master/Evaluation/Prefixes`

2. To address research question 2, a new subnet inference tool called `WISE` [52, 53] has been introduced and described in detail in Chapter 7. The detailed algorithms have been used to demonstrate how `WISE` can systematically handle the challenges induced by traffic engineering (especially load balancing) and in linear time, with the Appendix A providing the detailed time complexity analyses (cf. Sec. A.2). `WISE` has also been used to answer research question 3 as it has been validated on two groundtruth networks and compared favourably with `TreeNET` [55] and `ExploreNET` [118] in Sec. 8.1.

3. Using measurements of various Autonomous Systems collected from the PlanetLab testbed over extended periods of time, research question 4 has been thoroughly addressed too. Sec. 8.2 notably demonstrated that the `WISE` methodology can discover a large number of identical subnets for a given target network upon changing the vantage point while providing a considerable number of sound-looking subnets.

These major contributions are completed by two minor contributions.

1. The potential for adapting the `WISE` methodology to IPv6 has been discussed in Sec. 7.5, and it has been notably pointed out that the main obstacle to this is the discovery of responsive IP addresses within IPv6 prefixes – something the research community is actively researching [19, 106]. This therefore provides a partial answer to research question 5.

2. It was shown that the distribution of the prefix length of subnets within Autonomous Systems of different types does not necessarily follow a power law shape, therefore nuancing conclusions advanced by previous research [118].

Though the research work presented in this part of the thesis thoroughly expanded the topic of subnet inference, several improvements are possible. As already mentioned in Sec. 7.5, other ways of discovering potential subnet interfaces could be explored and would complement the current `WISE` methodology, at the very least to refine the subnets `WISE` can currently discover, and at best to discover more subnets. The post-processing step (cf. Sec. 7.4.2) could also be completed with additional heuristics to better identify subnets under certain scenarios, notably by relying on IPv4 assignment practices (as suggested in Sec. 7.5).

Also, as briefly mentioned in Sec. 8.2, some phenomena induced by traffic engineering and/or specific equipment (cf. "*warping echoing*" trails, cf. Sec. 8.2.2) could still impair `WISE` inference to the point where properly identifying the issues from a single vantage point could prove to be too difficult. One way to circumvent these issues could consist of developing strategies for combining multiple snapshots (as defined in Sec. 6.2.1) captured from distinct vantage points to combine the soundest subnets together into a single picture of the subnet-level of the target network.

Finally, it is worth noting the `WISE` methodology also complies with the idea of handling topology discovery in a holistic manner (cf. Sec. 1.1.1). As explained in Sec. 7.3.3, `WISE` embeds the alias resolution framework presented in the first part of this thesis (cf. Chapter 3) to solve potential alias lists resulting from flickering trails (cf. Sec. 6.1.3), which is consistent with the interactions between alias resolution and subnet inference first announced in Sec. 1.1.1.

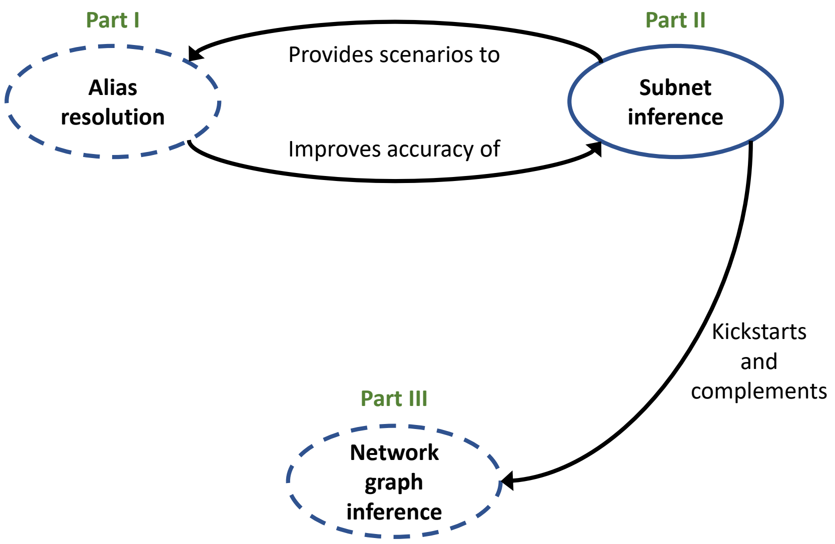


Figure 8.12: Interactions of the WISE methodology in a holistic approach.

Envisinoning subnet inference in a holistic scheme is not, however, restricted to the incorporation of alias resolution in the problem: as already explained in Sec. 1.1.1 and Sec. 4.1.1, subnet inference can also act as a first step for hop-level topology mapping, as notably demonstrated by TreeNET [55]. In the third and final part of this thesis, WISE will be used as a basis for designing a new network graph inference methodology, which will use subnets to map the hop-level graph of a target network with a more thorough and more careful use of both subnets and traceroute data than TreeNET (these ideas being developed in Chapter 10 and Chapter 11). Subnets discovered with the WISE methodology will also be used to complement hop-level maps by identifying network links (as will be explored in Chapter 11), which will later ease the creation of bipartite models for the Internet (in Chapter 13). In summary, subnet inference will be a key tool for achieving accurate and exhaustive network graph inference. Figure 8.12 summarizes the interactions of the WISE subnet inference methodology with the other topology discovery topics discussed in this thesis.

# Part III

# Network Graph Inference and Modeling

# CHAPTER 9

# INTRODUCTION TO NETWORK GRAPH INFERENCE AND MODELING

This chapter provides an introduction to both network graph inference and network modeling. Sec. 9.1 begins by summarizing the state-of-the-art regarding the classical tool `traceroute`, reviewing notably how it has been used up to this day to study network graphs and what its well-known issues are. Sec. 9.2 proceeds with the review of network mapping tools based on IGMP [33] probing, a unique probing approach explored by the research community late in the 2000's. Then, Sec. 9.3 presents the state-of-the-art of network modeling as well as bipartite graphs, a modeling formalism which showed potential for accurately accounting for the structure of the Internet. Finally, Sec. 9.4 summarizes the limits of the research discussed in this chapter and lists the research questions that will be tackled in this third and final part of this thesis.

## 9.1 `traceroute`-based topology mapping

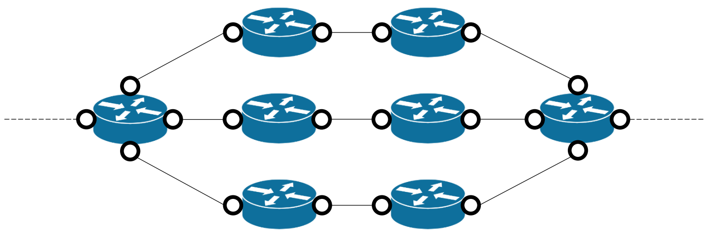
Since the 1990's, `traceroute` [119] has been one of the main tools of both network operators and academics. This section provides a representative sample of research work on topology discovery involving `traceroute`. Sec. 9.1.1 first provides a short history of `traceroute` itself, covering the characterization of `traceroute` paths over the years and their known issues. Sec. 9.1.2 then covers `traceroute` extensions, i.e., topology discovery tools that complement the traditional `traceroute` probing with additional probing work and/or heuristics to discover additional network elements or improve `traceroute` paths. Sec. 9.1.3 subsequently reviews a representative set of topology discovery tools involving `traceroute` that aims at discovering the router-level of a target network (see also Sec. 2.1). Finally, Sec. 9.1.4 briefly reviews state-of-the-art methods used to perform efficient `traceroute` probing, and in particular, *fast* topology mapping tools, i.e., tools that perform intensive `traceroute` probing to quickly discover large networks and/or the entire IPv4 Internet.

### 9.1.1 `traceroute`: history, characterization and issues

The original `traceroute` [119], designed by Jacobson in 1989, is a simple tool that sends a sequence of UDP probes towards a provided target IP address to discover the path between the vantage point and the target address. The first probe packet uses, by default, a TTL value of 1, each consecutive probe using an incremented TTL value, and this until the target replies to a probe or until a given number of consecutive probes turn out unsuccessful (e.g., several timeouts). Each probe packet whose TTL value expires causes the receiving device to send a `time-exceeded` ICMP message back to the vantage point, with the reply packet providing the IP address of the receiving interface as its source IP address. Therefore, `traceroute` can discover the interfaces of each router crossed on the way to some destination IP address and uses this as a way of revealing the network path. It can also be used to estimate the TTL distance (cf. Sec. 5.1.2) between the vantage point and the target.

However, as discussed in Sec. 2.4 and Chapter 6, `traceroute` paths have to be handled carefully. In particular, they are rarely stable, i.e., the discovered interfaces tend to vary from one measurement to another [12, 81, 82]. Moreover, since the 2000's, it has been common knowledge that a set of `traceroute` paths constitute a **d**irected **a**cyclic **g**raph (or DAG) of the IP-level [98].

While changes in `traceroute` paths between a given source and a given destination can be explained by different factors, such as changes in peering between Autonomous Systems (or ASes, cf. Sec. 9.3.1) [82], the research community has thoroughly investigated the effects of load balancing architectures (already mentioned in Sec. 2.4), i.e., a subset of devices that aim at balancing the amount of traffic between them to make the most of the available links. Such a strategy notably avoids a need to constantly increase the link capacity, which is usually a costly operation. Load balancing architectures can induce false link detection with `traceroute`: because the paths taken by each probe can vary, the final `traceroute` paths can suggest links that do not actually exist.


(a) Toy load balancer

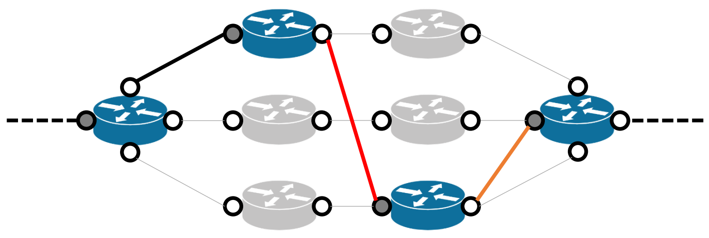

(b) A `traceroute` path with false links (grey circles are the discovered interfaces)

Figure 9.1: False link discovery with the regular `traceroute`.

Figure 9.1 illustrates the false link discovery issue, with Fig. 9.1a depicting a toy load balancer and Fig. 9.1b providing a potential (incorrect) path `traceroute` that could detect. In the event of the latter, the red edge depicts a truly false link, while the orange edge corresponds to a link that does not exist in the strict sense but which connect routers that do share a common link. The Paris `traceroute` tool was designed by Augustin et al. in 2006 to reduce false link discovery and to straighten the discovered paths in per-flow load balancers by manipulating header fields of ICMP [101], UDP [100], and TCP [103] probes [11]. Figure 9.2 illustrates the effects of Paris `traceroute`, reusing the toy load balancer from Fig. 9.1a. The same research group later extended Paris `traceroute` by incorporating a **M**ultipath **D**etection **A**lgorithm (MDA) to enumerate load balanced paths in the wild and characterize them [14]. Ulterior research by the same group in 2011, still using Paris `traceroute`, demonstrated that the large majority of `traceroute` paths collected in the wild cross at least one load balancer [12], further supporting the hypothesis that a collection of paths naturally forms an IP-level DAG.

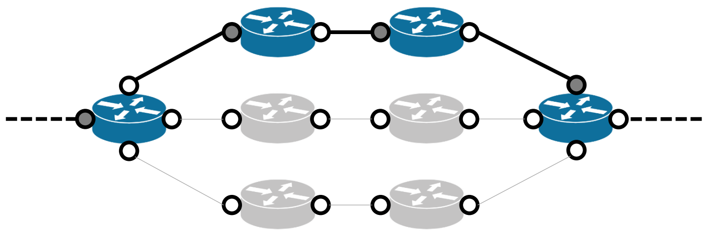


Figure 9.2: Accurate path discovered by Paris `traceroute` [11] (same network as Fig. 9.1a).

In addition to characterizing the effects of load balancing structures on `traceroute` paths, the research community also investigated the problem of anonymous hops more than once [59, 68, 125] (already mentioned in Sec. 6.1.1), i.e., routers that do not reply to `traceroute` probes (by default, because of a temporary failure, etc.), resulting in a missing interface in a `traceroute` path. Anonymous hops can be a problem for path interpretation, much like another well known issue: path cycles (also mentioned in Sec. 6.1.1), i.e., in a same path, a same router interface can appear more than once, suggesting that the packets have passed through the same device several times and have not taken an optimal path through the network. As highlighted in a 2016 study by Marchetta et al., path cycles can be explained by numerous causes, such as MPLS tunnels [85].

### 9.1.2 `traceroute` extensions and alternatives

While `traceroute` itself has been the topic of many studies, many other tools aim at complementing it, usually by combining additional probes and/or heuristics to refine the discovered paths and/or discover additional topological data. For example, this thesis has already mentioned `TraceNET` [116] (cf. Sec. 5.2.1), a `traceroute`-based subnet inference tool which infers the subnets that correspond to the links crossed between each router discovered via (Paris) `traceroute`. In the same spirit, `traIXroute`, introduced in 2016 by Nomikos and Dimitropoulos, analyzes `traceroute` data to locate IXPs (**I**nternet e**X**change **P**oints) within `traceroute` paths by exploiting existing data on IXP prefixes, IXP members, and IXP interfaces of BGP routers [95].

A completementary method, if not an alternative to `traceroute` probing lies in the *Record Route* option provided among the options of the IP protocol [102]. If used, this option allows one IP packet to record an interface of each router it crosses. *Sidecar*, introduced by Sherwood and Spring in early 2006, creates TCP connections to embed IP packets featuring the *Record Route* option to silently discover router interfaces [112]. Such an approach features has several advantages, such as preventing false link discovery or uncovering interfaces beyond NATs (**N**etwork **A**ddress **T**ranslators, devices that remap IP addresses and port numbers to other IP addresses in another network). Another tool introduced in 2008 by Sherwood et al., *DisCarte*, used disjunctive logic programming to merge regular `traceroute` paths with *Record Route* data in order to produce accurate paths [111]. Nowadays, using the *Record Route* has unfortunately become practically impossible due to current filtering practices, which usually consist of dropping any IP packet using an option [44].

### 9.1.3 `traceroute`-based router-level mapping tools

Because `traceroute` probing naturally discovers router interfaces, a number of tools have exploited it to build router-level maps of the Internet (cf. Sec. 9.3.1), usually in combination with various heuristics and alias resolution (cf. Chapter 2). The *Mercator* tool, introduced in 2000 by Govindan and Tangmunarunkit, performs `traceroute` probing in combination with custom heuristics and the same alias resolution method as `iffinder` [71] (cf. Sec. 2.3.1) to build router-level maps [45]. *Rocketfuel*, designed by Spring et al. around 2002, combines `traceroute` probing from hundreds of different sources with its `Ally` alias resolution component (cf. Sec. 2.3.2), BGP routing tables, DNS (to distinguish backbone from **P**oints-**o**f-**P**resences) and custom heuristics to build comprehensive router-level topologies of ISPs [113]. The `AROMA` (*Accurate ROuter-level MAp*) tool, elaborated by Kim and Harfoush in 2007, provides several improvements over the methodology of *Rocketfuel* [74]. It notably adds alias resolution based on reverse DNS (cf. Sec. 2.3.5) and an improved selection of the targets of the `traceroute` probing to discover more exhaustive paths while using less probes, resulting in both better maps and significantly smaller overhead probing. However, none of these tools were designed with the effects of load balancing in mind, which became increasingly common by the end of the 2000's [12] (cf. Sec. 9.1.1), in addition to relying on alias resolution methods that can now only cover a limited set of router interfaces, as discussed in Sec. 4.3.3.

Designed early in the 2010's, the `MERLIN` hybrid tool [83] combines (Paris) `traceroute` probing with IGMP probing performed from several vantage points to discover accurate intra-domain router-level maps; IGMP probing itself, which is now unfortunately outdated, is covered in more detail in Sec. 9.2. Finally, `TreeNET` [55], introduced to the community in 2016, performs `traceroute` probing towards subnets previously discovered via `ExploreNET` [118] to build tree-like maps of target networks. `TreeNET` has been already briefly discussed in Chapter 4, as it provided a space search reduction scheme for alias resolution. However, as highlighted in Sec. 4.3.5, tree-like maps can be difficult to interpret when load balancing influences most of the discovered `traceroute` paths.

### 9.1.4 Efficient `traceroute` and fast topology mapping

In addition to the numerous works studying the various issues affecting `traceroute` paths and how to mitigate them, several publications have also put an emphasis on performing efficient, large-scale `traceroute` probing. For instance, the *Doubletree* algorithm was designed by Donnet et al. in 2005 to reduce the overhead probing when running `traceroute` from both a single or multiple vantage points [39]. Heuristics of *Doubletree* have been notably reused by `WISE` [52, 53] (cf. Sec. 7.3.2).

During the 2010's, the research community explored another application of `traceroute` probing: ***fast topology mapping***. Instead of targetting a specific network, fast topology mapping tools aim at probing large network chunks or even the entire IPv4 Internet to capture comprehensive pictures of the Internet in a short timespan. Introduced by Beverly in 2016, *Yarrp* (**Y**elling **a**t **R**andom **R**outers **P**rogressively) sends `traceroute` probes towards all IPv4 /24 prefixes to capture a IP-level topology of the entire Internet in around one hour from a single vantage point [18]. *Yarrp* achieves this by encoding information in the ICMP and TCP headers [101, 103] to avoid maintaining a state for each probe and by using a pseudo-random probing scheme to avoid triggering security policies that could prevent replies being received.

*Yarrp* would later act as a basis for *Diamond-Miner*. Introduced in early 2020 by Vermeulen et al., *Diamond-Miner* (sometimes called *D-Miner*) expands upon *Yarrp* by adding the detection of load balanced paths [122], reusing heuristics from the **M**ultipath **D**etection **A**lgorithm (MDA) that was included in the classical Paris `traceroute` in 2007 by Augustin et al. [14]. *Yarrp* also inspired *FlashRoute*, introduced by Huang et al. in 2020 [65], which combines algorithmical elements from Paris `traceroute` [11], *Yarrp* and *Doubletree* to accurately discover `traceroute` paths throughout the entire IPv4 Internet while reducing the overall probing time by a factor of three.

## 9.2 IGMP-based topology mapping

Besides `traceroute` and all associated topology discovery tools, the research community has also explored an alternative probing technique based on the IGMP protocol [33]. The IGMP (**I**nternet **G**roup **M**anagement **P**rotocol) protocol is used to manage multicast groups in IPv4, and was initially designed so that multicast routers could dynamically learn which interfaces within neighboring subnets were part of a multicast group. However, two additional types of IGMP messages were later added by the DVMRP protocol [105] (**D**istance **V**ector **M**ulticast **R**outing **P**rotocol) to monitor routers. The `mrinfo` tool [67] was designed in 1995 by Jacobson to take advantage of these two additional messages, respectively `ASK_NEIGHBORS` and `NEIGHBORS_REPLY`, to list all multicast enabled interfaces of remote routers along with those adjacent to them. Though the DVMRP protocol is outdated, IPv4 multicast routers still replied to the aforementioned IGMP messages during the 2000's [96].

IGMP probing has the major advantage of silently discovering router interfaces and their adjacencies without involving alias resolution (cf. Chapter 2). It does, however, have two drawbacks: first, IGMP probing only works with routers that enable multicasting, and second, it can only work if the target domain (or any intermediate domain on the way) does not filter `ASK_NEIGHBORS` IGMP messages. The research community nevertheless used this probing technique to discover redundant links between ASes [91] and to map intra-domain topologies [97]. The latter could be achieved with the `mrinfo-rec` (`rec` for *recursive*) probing scheme, which sends new IGMP `ASK_NEIGHBORS` queries on each adjacent interface found in a `NEIGHBORS_REPLY` IGMP message, unless this interface has already been probed previously. Figure 9.3 depicts a toy topology that can be discovered thanks to `mrinfo` (`mrinfo-rec`), where the interfaces of $R_3$ have been highlighted in bold. Example 9.1 provides the `NEIGHBORS_REPLY` IGMP message that can be received by probing `10.0.0.2` (from $R_3$) with `mrinfo`. In this example, the listed interfaces (left side) correspond to interfaces of the target router, while the interfaces at the right of the → symbols belong to adjacent multicast routers.

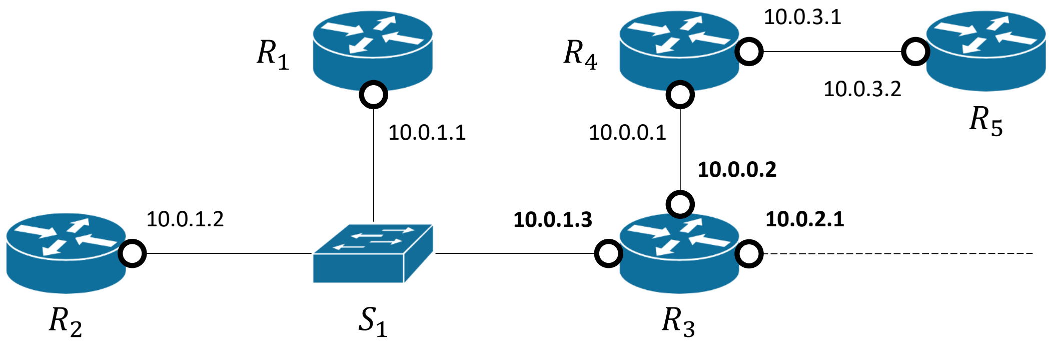


Figure 9.3: Toy topology (based on [90]) that can be discovered via `mrinfo` (`-rec`).

Listing 9.1: `NEIGHBORS_REPLY` IGMP message obtained by probing $R_3$ (via `10.0.0.2`) with `mrinfo`

```
10.0.0.2 [version 12.4]
    10.0.0.2 -> 10.0.0.1 [1/0/pim/querier]
    10.0.1.3 -> 10.0.1.1 [1/0/pim/querier]
    10.0.1.3 -> 10.0.1.2 [1/0/pim/querier]
    10.0.2.1 -> 0.0.0.0 [1/0/pim/leaf]
```

Starting from 2004, the `mrinfo-rec` scheme has been deployed to build a large dataset [96]. Such a dataset has notably been used by Mérindol et al. in 2010 to infer the presence of Layer-2 equipment in the Internet and evaluate its effects on the observed behaviour of routers [90]. The possibility to discover Layer-2 devices with `mrinfo-rec` is also suggested by Example 9.1, as there are two adjacencies listed for the interface `10.0.1.3`, while a point-to-point link would allow only one. These multiple adjacencies result from the Ethernet switch $S_1$ illustrated in Figure 9.3. Finally, the hybrid tool `MERLIN` (**ME**asure the **R**outer **L**evel of the **IN**ternet), introduced in 2011 by Marchetta et al., combined Paris `traceroute` [11], `mrinfo` and the `Ally` component of *Rocketfuel* [113], to comprehensively discover the router-level of a target domain using multiple vantage points [83]. Unfortunately, IGMP probing eventually became obsolete due to IGMP filtering becoming increasingly common early in the 2010's, as evidenced in a study by Marchetta et al. in 2012 [84].

## 9.3 Network modeling

While network graph inference has been an important part in the research work keeping the community busy since the late 1990's, network modeling has also been explored from multiple angles [98]. Indeed, data collected from the Internet can be used to model realistic topologies and later used to assess new technologies in simulated environments, such as new network protocols. In particular, realistic data can be fed to network models to randomly generate realistic topologies for simulation purposes. Sec. 9.3.1 first covers the topic of modeling the Internet from the perspective of the router-, AS- and hop-levels while Sec. 9.3.2 presents an alternative approach: bipartite graphs.

### 9.3.1 Router-level, AS-level and hop-level

A common model for the Internet consists of a graph where vertices account for a specific type of network element while the edges account for their adjacencies. For instance, in a ***router-level graph*** (see also Sec. 2.1), each vertex models an individual router while each edge accounts for the fact that two routers can exchange packets at the data link layer through a common link. This simple model allows the Internet to be studied with classical graph metrics [35], and in particular, router-level graphs allow for the study of the ***router degree***, i.e., how many routers a given router is directly connected to (e.g., via point-to-point links or LANs). Figure 9.4 pictures a toy example of a router-level graph, where the router at the center of the figure has a degree equal to seven due to sharing links with seven other routers. The distribution of the router degree in the Internet has been studied and debated many times by the community: in particular, Faloutsos et al. claimed in 1999 that the router degree distribution was typically shaped like a power law [40].

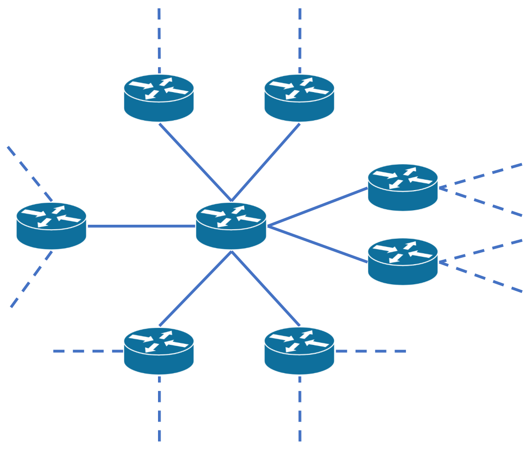


Figure 9.4: An example of a router-level graph. The router in the center has a degree equal to 7.

Studying the router-level of the Internet is notably what drove the research community into investigate alias resolution in the first place (cf. Chapter 2). In particular, a 2007 study by Gunes and Sarac highlighted that bad or incomplete alias resolution can drastically change the observed properties of the router-level of a target network [57]. However, the study of the router-level is not restricted to interpreting data collected through alias resolution, and several models try to account for physical, statistical, or economical properties of computer networks [98]. For instance, an analytical approach for modeling the Internet topology, proposed by Alderson et al. in 2005, relied on statistics and graph theory to build annotated router-level graphs accounting for technological constraints and economic considerations [10]. Another model, introduced by Wang and Loguinov in 2010, estimated the wealth of each ISP and used that knowledge to infer the evolution of a network topology over time with the help of a multiplicative stochastic process [123]. In particular, for a given point of time, this model ensured that the degree of each router stayed proportional to the associated ISP's wealth.

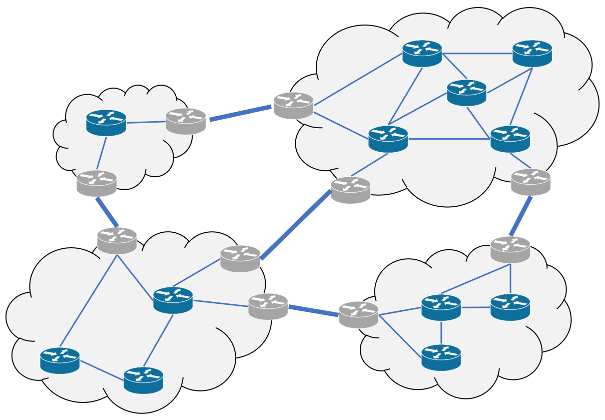


Figure 9.5: A toy example of an AS-level graph. The clouds depict individual ASes.

Over time, the research community has also studied the AS-level, i.e., the dynamics occurring between Autonomous Systems [37, 61]. More precisely, each AS has a given number of neighbors, i.e., other ASes it is directly connected with, to which it will route packets depending on the routes advertised by each neighbor through the BGP protocol [79]. Figure 9.5 depicts a toy example of an AS-level graph, where clouds model individual ASes and where grey routers on the edge of clouds correspond to the AS Border Routers (or ASBRs), i.e., the routers that handle inter-AS links. The discovery of AS relationships was notably investigated in a study by Dimitropoulos et al. in 2006 [36]. The same study also showed that the distribution of the number of peers for each AS also tends to follow a power law, just like the routers themselves at the intra-AS level. The AS-level plays a critical role in the routing, as it can completely change the routes taken by packets over time. In particular, `traceroute` paths collected from a same vantage point towards fixed destinations over small or long periods of time may never be stable, as evidenced by a study conducted in 2009 by Magnien et al. [82].

The research community has often relied on power law assumptions for network modeling, and these assumptions have been commented on and challenged several times [26, 28, 76, 78, 89]. In particular, a short literature review conducted in 2008 by Haddadi et al. concluded that the network topology generators of the time relied too much on power law assumptions, both at the (intra-AS) router-level and the AS-level [62]. Moreover, data collected with the help of `mrinfo` [67, 96] (cf. Sec. 9.2) was used to show how the observed degree of some routers may be overestimated due to the influence of Layer-2 devices [90], meaning the power law distribution might be an artefact.

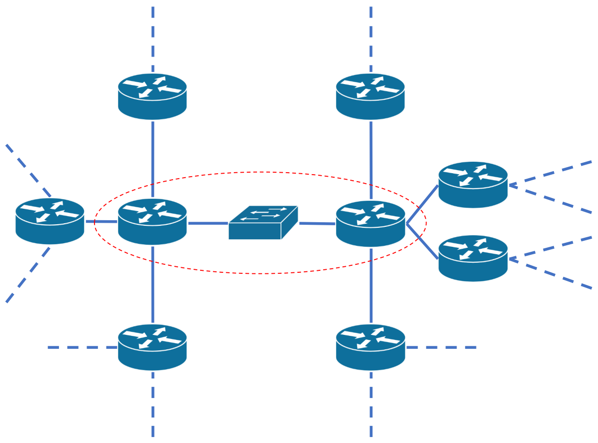


Figure 9.6: The large degree router from Fig. 9.4 may consist of a mesh involving Layer-2 equipment.

Indeed, Layer-2 devices may be used to connect several routers into a mesh that still functions as a single hop from the perspective of a measurement tool (such as `traceroute`, cf. Sec. 9.1.1). As a consequence, a router inferred from a measurement (e.g., via route hops observed in `traceroute` paths) might exhibit a larger degree than it actually has because this degree encompasses the one-hop adjacencies of the routers it is connected in mesh with. A simple example of this problem is illustrated in Figure 9.6, which pictures the same topology as in Fig. 9.4, but considering the central router actually consists of two routers connected into a (small) mesh via an Ethernet switch, as highlighted by the red dashed oval. This oval therefore accounts for a single hop from a measurement perspective, and features the same number of adjacent routers as the central router from Fig. 9.4. However, the actual largest router degree is five, since the two routers within the red oval have a degree of respectively four and five. This issue can motivate a change of perspective: to account for meshes of routers which still work as individual hops for packet forwarding, one can go from a router-level to hop-level view. A ***hop-level graph*** is therefore a variant of a router-level graph where each vertex can be either a single router or a mesh of routers. As such, Fig. 9.6 also constitutes an example of hop-level graph if the red dashed oval is viewed as a single vertex.

### 9.3.2 Bipartite graphs

***Bipartite graphs*** are a type of graph where the vertices are divided into two disjointed sets, $\top$ and $\bot$, so that every edge exclusively connects a $\top$ vertex with a $\bot$ vertex. Figure 9.7 is a toy example of a bipartite graph, where the $\top$ set consists of vertices $\alpha$, $\beta$, $\gamma$ while vertices A, B, C, D, E correspond to the $\bot$ set. Bipartite graphs are naturally suitable for model networks where vertices can be categorized into two families of elements, and have been frequently used by the scientific community to study social networks, such as file-sharing communities [66, 93, 124]. Using a bipartite formalism to model the Internet graph has been discussed several times, notably in a study conducted by Guillaume and Latapy in 2006 [56]. Indeed, since routers are usually connected together through other network elements, such as subnets or Layer-2 equipment (e.g., Ethernet switches), conceptualizing the

Internet as a bipartite graph where one set accounts for routers while the other set includes network elements that connect routers together is an intuitive modeling approach.

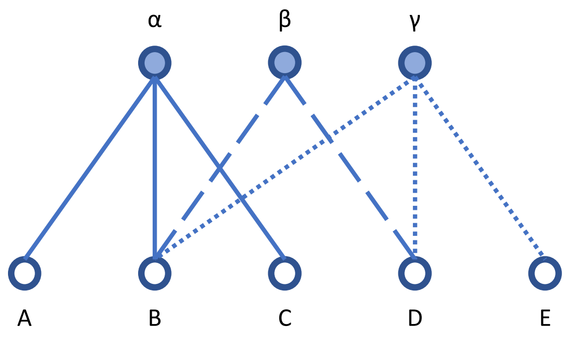
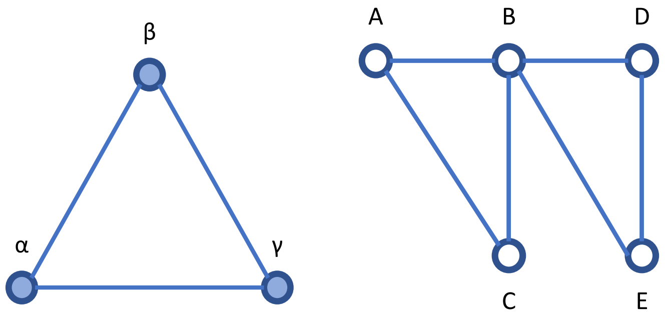


Figure 9.7: A simple example of bipartite graph. The ⊤ vertices are $\alpha$, $\beta$ and $\gamma$.

Bipartite graphs are useful not only because they can model some types of networks more easily than other formalisms, but also because they can be used to infer simpler graphs through the mechanism of ***bipartite projection***. Given a bipartite graph, a regular graph can be created with either ⊤ or ⊥ vertices by creating an edge between a pair of vertices of one party each time they share a common neighbor vertex from the second party. Therefore, if a bipartite formalism is chosen to model a computer network where one party consists of routers, it is possible to recover a router-level graph through projection (e.g., ⊤-projection if routers are ⊤ vertices) and to explore other levels of the network topology by projecting on the other party as well. Figure 9.8 depicts projections on the bipartite graph from Figure 9.7, with Fig. 9.8a giving the ⊤-projection and Fig. 9.8b illustrating the ⊥-projection. Finally, many interesting metrics have been designed for bipartite graphs by the scientific community [77] and are available by default in popular programming libraries to study networks in generals, such as the NetworkX library in Python [34].



(a) ⊤-projection (b) ⊥-projection

Figure 9.8: Projections of the bipartite graph depicted in Fig. 9.7.

While bipartite graphs have been studied for decades, the first actual bipartite formalism for the Internet was introduced to the research community in 2013 by Tarissan et al. [114]. This bipartite formalism includes Layer-2 equipment (such as Ethernet switches) as ⊤ vertices while routers constitute the ⊥ vertices. Such a formalism is in fact a consequence of contemporary advances in topology mapping: a few years earlier, IGMP [33] probing was explored by the research community with the help of `mrinfo` [67] to discover routers and Layer-2 devices [83, 90, 97] (cf. Sec. 9.2) and to challenge the idea that the distribution of the router degree in the Internet follows a power law [40]. The `mrinfo` dataset [96] was used by Tarissan et al. as a basis to create realistic bipartite graphs to model the Internet, which were subsequently used to highlight its topological properties and to create a bipartite graph generator able to randomly create new topologies for simulation purposes [114]. To account for point-to-point links between routers, i.e., routers being connected together without an intermediate Layer-2 device, Tarissan et al. used *virtual* Layer-2 devices that only connect two routers linked in practice by a point-to-point link. This workaround allows the projection of a bipartite graph on the routers to accurately depict the router-level. Figure 9.9 pictures a toy network (Fig. 9.9a) and its equivalent bipartite graph as defined by Tarissan et al. (Fig. 9.9b), where the orange dashed lines account for the single point-to-point link.

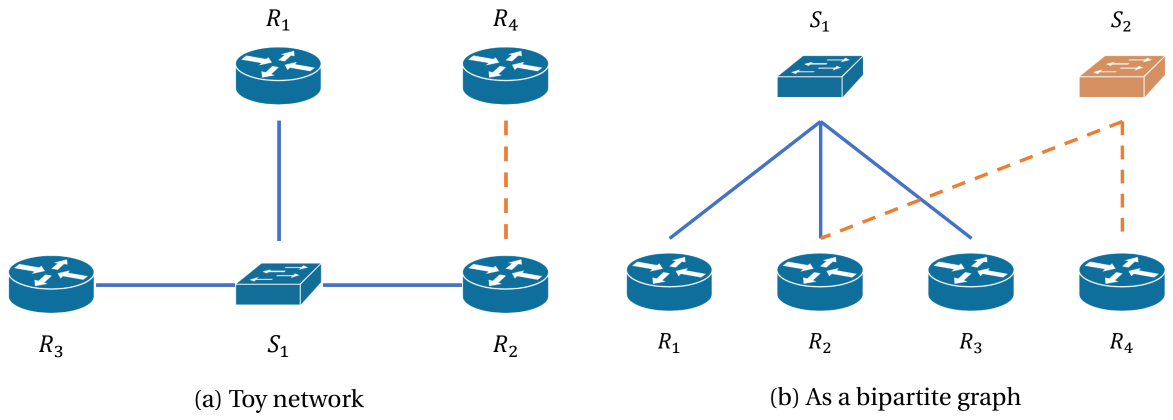


Figure 9.9: The Layer-2/Layer-3 bipartite formalism proposed by Tarissan et al. [114].

While realistic and thoroughly assessed, this bipartite formalism has two drawbacks. On the one hand, it relies heavily on data collected via IGMP probing, although such a method has become difficult to use due to filtering practices applied by network operators [84] (cf. Sec. 9.2). On the other hand, the formalism itself relies on a workaround to model point-to-point links. Though the aforementioned *virtual* Layer-2 devices make it possible to create an accurate picture of the router-level while projecting a bipartite graph on the routers, Tarissan et al. mentioned that such a workaround can also impact the structure of randomly generated bipartite graphs. As will be discussed in Chapter 13, a solution to this issue could consist of including subnets in the formalism, since (small) subnets usually correspond to point-to-point links in the Internet (cf. Chapter 5).

## 9.4 Closing comments and research questions

Many approaches for both network graph inference and network modeling have been explored by the research community over the past two decades. In particular, the research involving IGMP probing has proven to be one of the most insightful, as it has even led to Layer-2 equipment discovery [90] (cf. Sec. 9.2), which in turn motivated a bipartite model of the Internet [114] (cf. Sec. 9.3.2). Unfortunately, IGMP probing has been rendered practically impossible to use due to new filtering practices [84]. Other approaches to discover topological data have also lost relevance over time, such as the path discovery via the *Record Route* option of the IP protocol [102], rendered obsolete by modern filtering practices [44]. Furthermore, network mapping approaches that could still be deployed today do not account well for modern topological properties of the Internet, such as the prevalence of load balanced paths [12] (cf. Sec. 9.1.1), as discussed in Sec. 9.1.3. Finally, modern efforts tend to focus on discovering other network elements, such as IXPs [95], or on fast topology mapping (cf. Sec. 9.1.4).

Due to state-of-the-art approaches either becoming deprecated or losing accuracy over the years, new network graph inference schemes are much needed to capture accurate snapshots of intra-domain topologies. Moreover, though numerous network modeling works focus of one particular level of the network stack, research resulting from data collected through IGMP probing has demonstrated the potential of bipartite formalism for the Internet [114]. Indeed, bipartite graphs are naturally suited to computer networks, especially when considering the intra-domain level (i.e., within a single AS), due to network components usually being linked together through other components (e.g., a point-to-point link between routers is typically a /30 or /31 subnet). This third and final part of the thesis will address the challenges of accurately discovering intra-domain topologies and modeling these topologies as bipartite graphs by answering the following research questions.

1. **How can `traceroute` paths be best handled to map the topology of a target network in today's Internet ?** As mentioned in particular in Sec. 9.1.1, the effects of load balancing in today's Internet are a fact merely an hypothesis, and the structure arising from a collection of `traceroute` paths will be much closer to a **d**irected **a**cyclic **g**raph (or DAG) than a tree [98]. It is therefore crucial to find a way to account for this inherent property of `traceroute` data, rather than finding a workaround. This question will be thoroughly discussed in Chapter 10.
2. **Is it possible to design a topology inference tool that can systematically build the DAG of a target network ?** Accounting for the DAG structure of `traceroute` data is one thing, but to accurately map a target network, it must be possible to go from an IP-level perspective to at least a hop-level, if not a router-level perspective. Chapter 11 will present and describe a topology discovery tool able to perform this task.
3. **Is this new tool capable of building DAGs that are true to the target domain, regardless of the vantage point ?** The tool described in Chapter 11 will be thoroughly validated, then deployed in the wild to evaluate the effects of changing the vantage point to measure a same given target network. This evaluation will be provided in Chapter 12.
4. **Can the data collected by the new tool be interpreted with bipartite formalisms to study the topological properties of a target network ?** Using many snapshots (as defined in Sec. 6.2.1) of various Autonomous Systems (or ASes), new bipartite formalisms will be discussed and used to highlight structural properties of the modern Internet. Chapter 13 will address this question.

# Chapter 10

# Interpreting traceroute paths

This chapter explores in detail how `traceroute` records can be interpreted to build the map of a target network. As such, it addresses research question 1 from Sec. 9.4, which is about finding out how the directed acyclic graph shape of a collection of `traceroute` paths can be best accounted for in order to build accurate maps of the modern Internet.

As already mentioned in Sec. 9.1, `traceroute` is one of the most widely used tools in topology discovery, notably thanks to its ability to reveal the router interfaces crossed by the probes sent towards a specific destination. Because the paths towards a specific network will share many similarities, notably exhibiting the same first interfaces, it is tempting to view a collection of `traceroute` paths towards a specific target domain as a kind of tree where a node models a single interface and features one or several successors and – ideally – one predecessor. However, network topologies can notably include convergence points, i.e., devices where multiple network paths converge as a result of load balancing architectures and other traffic engineering policies [14] (cf. Sec. 2.4 and Sec. 6.1.3). Because of convergence points and their multiple predecessors, a collection of `traceroute` paths should actually be closer, in practice, to a **d**irected **a**cyclic **g**raph (or DAG) [12, 98] (cf. Sec. 9.1.1).

With the help of heuristics, the topology discovery tool `TreeNET` [55] is nevertheless able to build a tree from a set of paths (towards a set of subnets) collected with Paris `traceroute` [11]. How it manges to do this in practice and the limits of such a methodology are explained in detail in Sec. 10.1.

`traceroute` is also well known to suffer from several issues induced by traffic engineering or specific router configurations, some of which have already been discussed in previous chapters (cf. Chapter 6). Another approach to exploit `traceroute` paths would consist of processing the collected paths to identify the various problems affecting them. In doing so, it becomes possible to post-process the discovered paths in order to more easily build the map of a target network, a strategy that can notably benefit `TreeNET`. The `traceroute` extension `RTrack` [46] implements such a strategy to improve the discovered paths. The precise issues it deals with are detailed in Sec. 10.2.

However, post-processing the data collected with (Paris) traceroute in order to ease topology mapping is open to debate. In fact, it might not even be necessary, i.e., it is possible to work with the initial data, or even with partial traceroute records. Re-using the concept of neighborhood used by TreeNET (cf. Sec. 4.1.1) and the concept of trail introduced with WISE (cf. Sec. 6.1.1), a target domain can be mapped out step by step without trying to mitigate traceroute issues and while embracing the possibility for a router interface discovered by traceroute to have several predecessors in the paths where it appears. Such a strategy is extensively presented and assessed in Sec. 10.3.

Finally, Sec. 10.4 concludes this chapter by summarizing the merits of each approach and arguing which one would be preferable to build accurate maps of the modern Internet.

## 10.1 Network mapping with complete traceroute paths (TreeNET)

This section reviews the topology mapping methodology of TreeNET [55]. As its name implies, TreeNET builds a tree-like mapping of a target network by collecting Paris traceroute [11] paths towards a set of subnets found within the same target network. The subnets are initially inferred with the help of ExploreNET [118] (cf. Sec. 5.2.2) then refined with a correction heuristic implemented by TreeNET to recover large subnets first detected as several small subnets by ExploreNET [55] (cf. Sec. 5.2.3). Sec. 10.1.1 first explains in detail how TreeNET uses the (complete) traceroute paths towards the inferred subnets to build its tree-like structure, then Sec. 10.1.2 reviews the limits of such a methodology, some of which have already been discussed in Chapter 4.

### 10.1.1 From traceroute paths to a tree-like mapping

TreeNET starts building its tree-like mapping of the target domain once it has collected subnet-level data and (Paris) traceroute paths towards each inferred subnet. In practice, these paths are obtained by using a pivot interface as the target. As a reminder, an interface of a provided subnet is considered to be a pivot interface of this subnet when it does not belong to the ingress router (i.e., the last router crossed before reaching the subnet) and when it shares comparable properties with other interfaces found within the same subnet (see also Sec. 5.1.2).

The subnets themselves act as the leaves of the tree, while the root replaces the vantage point. The internal nodes model the individual hops of the traceroute paths and their succession, i.e., each internal node of depth $N$ is labeled with a router interface that appears at the $N^{\text{th}}$ position in one or more paths. The immediate result of this construction is that the internal nodes bordered by multiple leaves constitute neighborhoods, i.e., network locations that are bordered by subnets located at most one hop away from each other and which consist, in practice, of a single router or multiple routers connected in mesh. Moreover, the internal nodes of the tree can be interpreted as a hop-level topology (see also Sec. 9.3.1), and identifying the interfaces associated with each hop (e.g., from surrounding subnets) is a form of space search reduction for alias resolution [51] (cf. Sec. 4.1.1).

**Algorithm 13** Insertion of a subnet in the tree

```
Require: $N$, root node of the tree
 1: procedure INSERT(Node $N$, Subnet $S$)
 2:     $R \leftarrow S$.getRoute()
 3:     if $N$.getNextChild($R[N$.getDepth() + 1]) == $\emptyset$ then
 4:         Prev $\leftarrow N$
 5:         for $i \leftarrow N$.getDepth() : $R$.getLength() do
 6:             New $\leftarrow$ new Node($R[i]$)
 7:             Prev.addChild(New)
 8:             Prev $\leftarrow$ New
 9:         Prev.addChild(new Node(S))
10:         return
11:     I $\leftarrow$ INSERTIONPOINT($N$, $R$)
12:     INSERT($I$, $S$)
13:     $P \leftarrow I$.getParent()
14:     while $P \neq \emptyset$ do
15:         $L \leftarrow R[P$.getDepth()]
16:         if $L \neq P$.getLabel() then
17:             $P$.addLabel($L$)
18:             set $\leftarrow$ NODESATDEPTH($N$, $P$.getDepth())
19:             for $M \in$ set do
20:                 if $L \in M$.getLabels() then
21:                     $P$.merge($M$)
22:         $P \leftarrow P$.getParent()
23:     return
```

Of course, `TreeNET` has to carefully handle the case of nodes that might feature multiple predecessors in the paths. It does this by allowing nodes to model multiple router interfaces discovered via (Paris) `traceroute` at the same distance TTL-wise. As a consequence, these nodes, which are denoted as ***multi-label nodes*** (and already mentioned in Sec. 4.1.1), no longer model a single neighborhood (when surrounded by leaves) but rather a superposition of multiple neighborhoods. They are created while the tree is being built, using a specific tree construction process, summarized in pseudo-code by Algorithm 13. Whenever a new subnet has to be added to the tree, `TreeNET` first finds the ***insertion point*** in the tree, defined as the deepest internal node which shares a common router interface (or label) with the route recorded for the new subnet (line 11). Finding this insertion point can be implemented efficiently in practice by using a side data structure which lists all nodes for a given depth, starting with the deepest nodes first. Then, the missing internal nodes (i.e., hops between the matching label of the insertion point and the new subnet) and the new leaf are inserted below the insertion point (lines 3– 10). `TreeNET` subsequently checks the tree from the insertion point up to the root, inserting at each internal node the router interface found at the equivalent TTL distance whenever the route of the new subnet differs from the data currently stored in the tree (lines 12– 23). For each internal node, if the label from the route of the new subnet already appears in another internal node of the tree at the same depth, it is merged with the current node (lines 19– 21). In the end, each router interface found in the `traceroute` data should appear only once in the tree at a given depth. It is worth noting that, in the pseudo-code, the insertion of the missing nodes is handled in a recursive manner to simplify the construction when the tree is empty or when a route completely differs from the route data inserted up to that point. The pseudo-code also assumes the root node provides access to the data structure allowing fast look-up of nodes at a given depth.

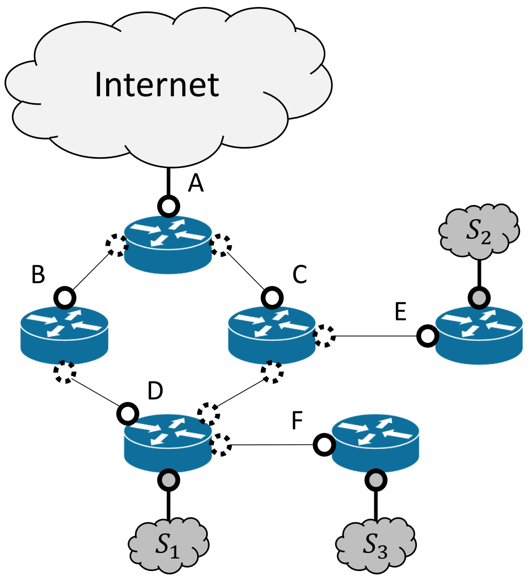
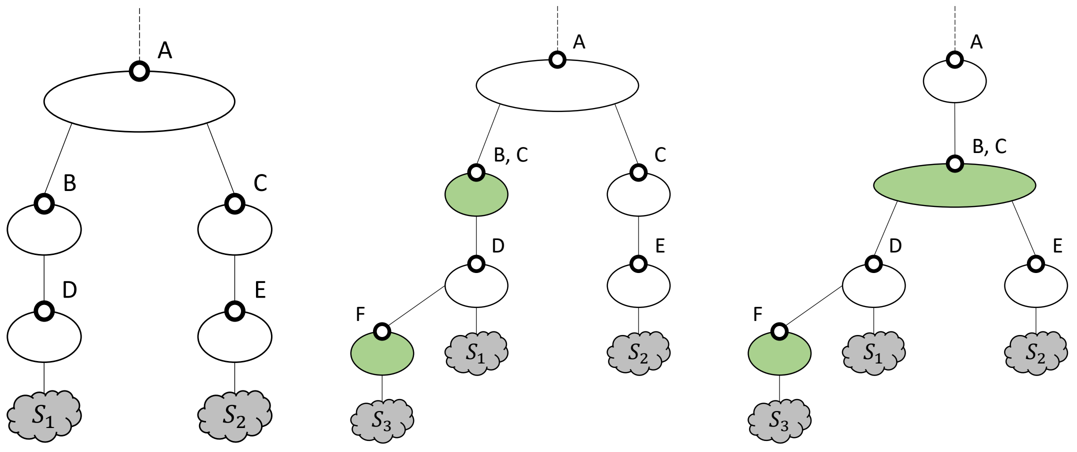


Figure 10.1: Toy network mapped by Fig. 10.2. Plain circles depict the discovered interfaces.



(a) Initial tree (b) Insertion point for $S_3$ (c) After merging $B, C$ and $C$

Figure 10.2: Inserting $S_3$ in a tree already containing $S_1$ and $S_2$ (topology of Fig. 10.1).

To complement Algorithm 13, Figure 10.1 and Figure 10.2 both represent a toy topology and how it would be built as a tree-like mapping by `TreeNET`. Given the topology shown in Fig. 10.1, possible (Paris) `traceroute` paths towards subnets $S_1$, $S_2$, and $S_3$ could be {A, B, D}, {A, C, E}, and {A, C, D, F} respectively. The insertion of $S_1$ and $S_2$ is immediate, as Fig. 10.2a shows. Fig. 10.2b and Fig. 10.2c illustrates how the insertion of $S_3$ occurs, with the changes in the tree being highlighted in green. The insertion point is obviously the node bearing the router interface $D$ as a label. As the parent node does not have the label $C$, such a label is added; however, label $C$ already appears in the tree at the same depth, therefore the branch is merged with the node containing both labels $B$ and $C$. In the final tree, all router interfaces recorded in the `traceroute` data appear only once in the tree. Of course, a router interface will only appear once if and only if all its occurrences in `traceroute` data appear at a fixed TTL distance (as defined in Sec. 5.1.2). If one router interface appears at different TTL distances in the paths, it will appear in several internal nodes found at distinct depths.

The time complexity to build the tree-like mapping depends on two variables: $N$, the number of subnets to insert into the structure, and $M$, the total number of router interfaces that are unique with respect to a given TTL distance in the `traceroute` data (i.e., if an interface is observed at two TTL distances, it is counted twice). Given that `traceroute` paths are short in length (i.e., usually much less than 64), meaning the final tree will feature an equally small depth, and assuming the creation of a leaf (i.e., single subnet) is $\mathcal{O}(1)$ (this can be done in practice by listing subnets within the parent internal node), then the insertion of a single subnet can be $\mathcal{O}(\log M)$ with efficient data structures (or $\mathcal{O}(1)$ with hash maps, though having a hash map for each relevant TTL distance is costly). Therefore, building the tree-like mapping starting with $N$ subnets is $\mathcal{O}(N \log M)$. The latest implementation of `TreeNET` can be found online at the dedicated public GitHub repository [1].

### 10.1.2 Limits of the tree-like mapping

While easy to implement and suitable for simple topologies (such as the ULiège network, cf. Sec. 4.2), the tree-like topology mapping implemented by `TreeNET` has several limitations. An immediate limitation, already hinted at Sec. 10.1.1, is that `TreeNET` is not well armed against asymmetric paths found in load balancing architectures [14]. Indeed, the tree construction guarantees a router interface can only appear once at given depth, but does not guarantee it will occur only once in the whole tree: if such an interface is observed at several distinct TTL distances in the `traceroute` data, it will appear in multiple internal nodes as well. Because of this, the neighborhoods `TreeNET` attempts to discover can be split into several smaller neighborhoods.

Another issue is the handling of anonymous hops, i.e., specific hops within `traceroute` paths which could not be identified because the corresponding router did not reply to the `traceroute` probe, either because of some network delay or because of its configuration (cf. Sec. 6.1.1). Since anonymous hops can, in practice, account for multiple devices, nodes that model them [2] are difficult to interpret, especially when they are multi-label and therefore bear several other router interfaces.

The fact that each node can only have one predecessor can also make the interpretation of a tree more complex. In particular, multi-label nodes can only be correctly interpreted if the associated router interfaces are further analyzed through alias resolution. While this can be a tool for discovering

[1] `https://github.com/JefGrailet/treenet`

[2] Such hops can be denoted by a special label or by the IP address `0.0.0.0` in practice.

individual routers in load balanced architectures [50], this also means TreeNET is highly dependent on the accuracy of alias resolution for a topology to be easy to interpret. As argued in Chapter 4, alias resolution can be difficult to achieve in today's Internet and can partly rely on space search reduction – which TreeNET is meant to perform. This is why Sec. 4.3.5 suggests there were some large alias resolution scenarii for specific target Autonomous Systems: there were simply some multi-label nodes with large amounts of router interfaces, therefore amounting to multiple neighborhoods at once (cf. Sec. 10.1.1) and restricting the benefits of space search reduction.

Overall, the root of all problems of TreeNET is that it was designed so that its tree-like mapping accounts for the entirety of the recorded traceroute paths, therefore including their issues as well, such as anonymous hops and consequences of load balancing. As such, TreeNET works well with topologies where little traffic engineering is experienced, but can struggle to offer accurate topology mapping otherwise. This suggests using the complete and raw traceroute paths should be avoided: either the traceroute data is pre-processed, as will be discussed in Sec. 10.2, or it is handled so that only the useful portions of each path is used, as will be elaborated in Sec. 10.3.

## 10.2 Detecting and correcting traceroute illnesses (RTrack)

In order to mitigate the various traceroute issues that complicate the interpretation of a tree-like mapping as built by TreeNET (cf. Sec. 10.1), a stand-alone tool named RTrack [46] (whose name stands for "***R****ate-liming* ***Track****er*") was designed during the first half of 2017 to quantify traceroute issues outside the context of TreeNET (cf. Sec. 10.1). It therefore amounts to a traceroute extension that aims at detecting three issues: anonymous hops induced by rate-limiting, *route stretching*, and *route cycling*. RTrack not only detects such issues, but it also processes the collected paths to mitigate them. Sec. 10.2.1 first details each of the three aforementioned problems and how RTrack mitigates them, after which Sec. 10.2.2 reviews the benefits and drawbacks of such a strategy.

### 10.2.1 traceroute issues and correction heuristics

As already reminded in Sec. 10.1.2, ***anonymous hops*** [59, 68, 125] refer to hops in a traceroute path where the corresponding probe has timed out, i.e., no reply was sent by any router. This can be a default behaviour (some routers are configured not to reply to probes), but also the consequence of ***rate-limiting***. Rate-limiting is a type of policy implemented by routers to reduce the traffic coming from a given source, which consists of dropping packets for a short period of time after a certain number have been received. Such a policy can be useful to control the volume of the network traffic but can also act as a security measure against network attacks.

The two other aforementioned problems rather characterize anomalies observed with non-anonymous hops. On the one hand, ***route stretching*** simply refers to the fact that the same router interface is observed at distinct TTL distances inside a set of traceroute paths as a consequence of asymmetric paths in load balancing architectures [14]. As such, it amounts to a generalization of the phenomenon of warping trails discussed in Sec. 6.1.2. ***Route cycling***, on the other hand, refers to how the same router interface is observed multiple times within a same `traceroute` path, hinting at a cycle in the routing. It therefore amounts to a generalization of the cycling hops (cf. Sec. 6.1.1).

Though all three issues are well known to the research community [14, 85], `RTrack` expands on the state-of-the-art by implementing heuristics to quantify rate-limiting and to mitigate all three issues, i.e., by producing *corrected* paths free from the observed anomalies. Such paths can be subsequently used to improve the construction of a tree-like mapping as performed by `TreeNET` (cf. Sec. 10.1).

Listing 10.1: Set of `traceroute` paths hinting at a rate-limiting policy

```
10.0.64.1, 10.0.64.128, 10.0.64.192, 10.0.65.32, 10.0.65.64, 10.0.128.1
10.0.64.1, 10.0.64.128, 10.0.64.192, 10.0.65.32, 10.0.65.64, 10.0.128.3
10.0.64.1, 10.0.64.128, 10.0.64.192, 10.0.65.32, 10.0.65.96, 10.0.128.4
10.0.64.1, 10.0.64.128, *, 10.0.65.32, 10.0.65.96, 10.0.128.7
```

Rate-limiting can be identified and mitigated with the help of the *two policemen and a drunk* heuristic. The idea is the following: given a set of paths that feature the same observed interfaces at given TTL distances, if the same sequence of three interfaces can be found in most paths while other paths feature a comparable sequence where the middle interface is an anonymous hop, then this anonymous hop can be replaced with the middle interface of the sequence without the anonymous hop. Example 10.1 shows a small set of `traceroute` paths where the last path features an anonymous hop that can be inferred with the help of the three other paths. Indeed, the first three paths feature the sequence 10.0.64.128, 10.0.64.192, 10.0.65.32 while the last path contains 10.0.64.128, $*$, 10.0.65.32 at the same positions, suggesting the $*$ can be replaced with 10.0.64.192. At the same time, the router bearing the 10.0.64.192 interface could be considered as implementing a rate-limiting policy.

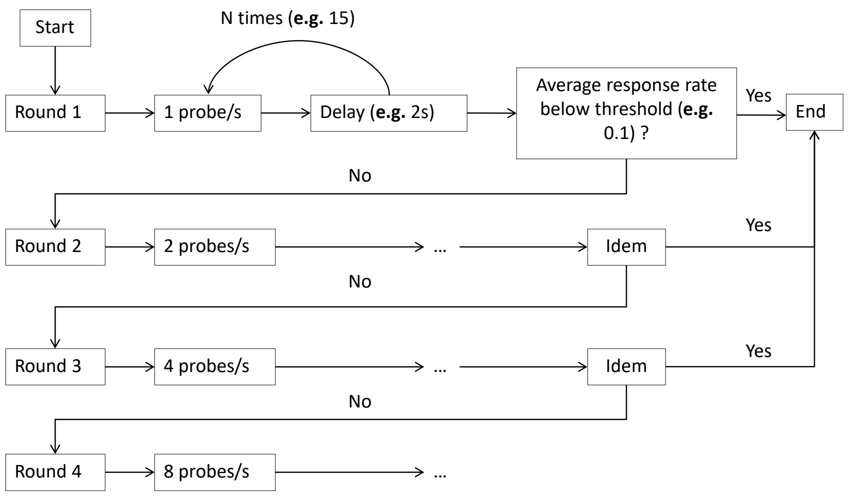
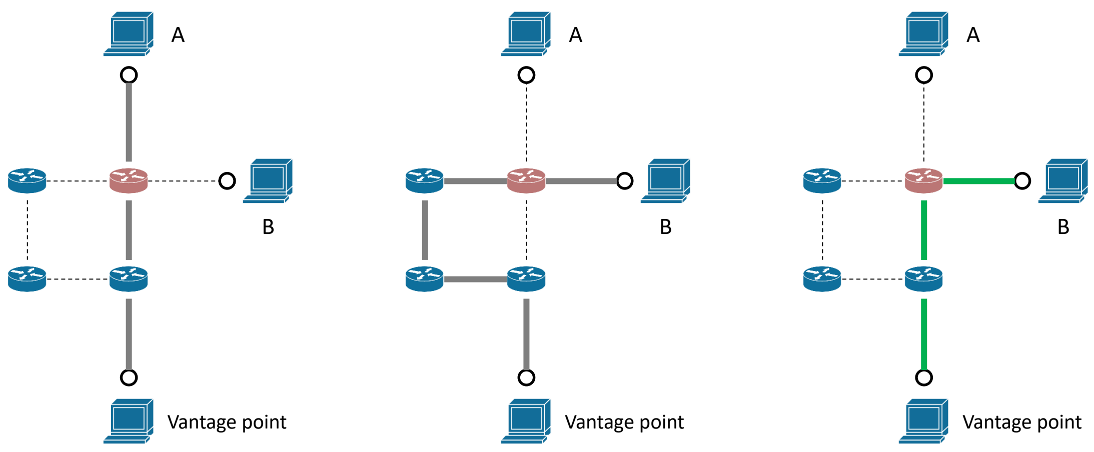


Figure 10.3: Heuristic used to estimate the rate-limiting experienced with a given router interface.

To verify rate-limiting does occur with the interfaces used to fix anonymous hops with the *two policemen and a drunk* heuristic, an algorithm can be used to quantify at which probing rate the response rate (i.e., the number of replies divided by the number of probes) starts to decrease. This algorithm, expressed as a flow-chart by Figure 10.3, consists of repeatedly sending a `traceroute` probe adjusted for the allegedly rate-limiting interface for a short period of time. After this first *round*, a second *round* is scheduled but sending two probes at once each time. A third round can then be scheduled with four probes, then a fourth round with eight probes, etc., the number of probes doubling each time. The heuristic stops as soon as the response rate becomes too low (below a threshold, e.g. 0.1), or if it reaches a certain threshold of probes sent simultaneously during a round. The number of probes sent simultaneously at the last round, if a low response rate appears, can be considered as a threshold for the rate-limiting experienced with the targetted router interface.



(a) Path towards $A$ (b) Path towards $B$ (c) Fixed path towards $B$

Figure 10.4: Mitigating route stretching. The first path is used to correct the second one.

Route stretching does not require any particular heuristic to be detected, as it can simply consists of making a census of the TTL distances at which a given router interface is observed in the `traceroute` data (much like the detection of warping trails, cf. Sec. 7.3.3). As for its mitigation, i.e. guaranteeing the router interfaces always appear at the same TTL distance, the `traceroute` paths where the observed TTL distances are the smallest can be selected and the portion of these paths up to the problematic interface can be used as a prefix. Then, for any path where the observed TTL distance of the problematic interface is higher, the portion of this path after the interface can be chosen as a suffix. The prefix, the problematic interface and the suffix are then put together into a shorter route which is consistent with other paths when it comes to the TTL distances (a property which is notably desirable in the context of `TreeNET`, cf. Sec. 10.1.2). A similar strategy can be applied to mitigate route cycling: the portion of the route up to the first occurrence of the cycling interface is used as a prefix, while the end of the route after the last occurrence is used as a suffix. Figure 10.4 shows a toy example where route stretching is observed and can be mitigated. The second path in the figure (Fig. 10.4b) is updated with the data from the first path (Fig. 10.4a) to produce an updated path where the router interface observed at varying TTL distances, symbolized by the red router in the Figure 10.4, is found at the same TTL distance as in the first path (Fig. 10.4c).

It is worth noting that the correction of paths where stretching was observed has been validated back as far as 2017 on a Belgian ISP after running `RTrack` towards said ISP from the PlanetLab testbed [3]. A network administrator working for this ISP could verify that all *corrected* paths were true to the topology and did not include false links. Interestingly, the administrator also noted that the additional router interfaces appearing in the *stretched* paths were associated to back-up links.

### 10.2.2 Pre-processing `traceroute` paths: a good idea ?

Though `RTrack` [46] provides several treatments to the `traceroute` data that could benefit `TreeNET` (cf. Sec. 10.1.2), it does not provide any treatment to address the main limitation of `TreeNET`: that each internal node in the tree-like mapping can only have one predecessor. Indeed, if the probes sent by `TreeNET` cross symmetrical paths in load balancers and pass through convergence points, i.e., devices where multiple (possibly load balanced) paths converge [14], the final tree-like map will still feature several multi-label nodes (cf. Sec. 10.1.1) that will be difficult to interpret. This notably mitigates the benefits of `TreeNET` for alias resolution (cf. Sec. 4.3.5). Moreover, the heuristics implemented by `RTrack` only partially mitigate anonymous hops and the effects induced by asymmetrical paths in load balancers, with no particular benefit regarding the consequences of symmetric paths.

Not only is the pre-processing of the `traceroute` data offered by `RTrack` of no help to overcome the main limitation of `TreeNET`, but it can also induce a biased and incomplete view of the topology by hiding some links. In particular, while validating the *corrected* paths on a Belgian ISP (cf. Sec. 10.2.1), the additional hops in the *stretched* paths proved to be back-up links. Including such links in a topology mapping can be a valuable addition, since they play a critical role in the internal routing of the target domain. Figure 10.4 also demonstrates this problem: if only the paths to *A* and *B* are known, and if the path to *B* is *corrected* as shown by Fig. 10.4c, then the two routers on the left side in each sub-figure will be left out of the final mapping based on the improved paths.

These limitations suggest that the very idea of turning `traceroute` data into a tree-like map only works for ideal settings and can only offer a partial discovery upon probing complex target networks. In particular, the proportion of (Paris) `traceroute` paths that are influenced by load balancing is always non-negligible [12] and can be very large depending on the target network and the vantage point (as previously discussed in Chapter 6). Naturally, this induces the probes in crossing multiple convergence points with multiple predecessors. Therefore, a new topology mapping technique that allows a node to have multiple predecessors should be designed. With this premise, the resulting structure would no longer be a tree, but a **d**irected **a**cyclic **g**raph (DAG). By design, this structure should be much closer to the natural shape of raw `traceroute` data [98]. Sec. 10.3 presents the main ideas used to design the new topology mapping scheme and extensively assess them.

[3]As with any validation in this thesis, no groundtruth data will be released for security and confidentiality concerns.

## 10.3 From `traceroute` paths to a directed acyclic graph

This section introduces the main ideas used to map a target domain with a directed acyclic graph rather than a tree-like structure in order to overcome the limitations of `TreeNET` (cf. Sec. 10.1). Sec. 10.3.1 first presents the intuitions and early observations that motivated this research direction. Then, Sec. 10.3.2 formalizes the concepts that will act as the main tools for the systematic mapping of a target network as a DAG. Finally, Sec. 10.3.3 assesses the soundness of these concepts with a validation and an analysis of data collected from the PlanetLab testbed in the second half of 2019, both being conducted with a slightly modified `WISE` [52, 53].

### 10.3.1 Early ideas and observations

In order to build a structure comparable to the tree-like mapping of `TreeNET` where each node can have multiple predecessors just like in the `traceroute` data, the new structure should first make sure the nodes still correspond to the neighborhoods, i.e., network locations that abstract individual routers or meshes of routers (possibly involving Layer-2 equipment) that are typically bordered by a set of subnets located at most one hop away from each other (cf. Sec. 4.1.1). As explained in Sec. 10.1.1, each internal node in the tree built by `TreeNET` corresponds to a neighborhood, which therefore corresponds in practice to an aggregate of subnets whose paths feature the same length and the same last hop. Therefore, neighborhoods can be built on the basis of the penultimate hops towards their respective bordering subnets [4], and if the length requirement (i.e., the paths should have the same length) is excluded, then this way of discovering neighborhoods becomes more suitable for target networks where traffic engineering induces varying TTL distances for the observed IP interfaces.

The key to building the neighborhood-based DAG of a target domain lies in finding how neighborhoods are located in respect of each other. The mapping methodology of `TreeNET` does it by design, but performing the same task without the tree paradigm requires more care, and will be elaborated on in Chapter 11. However, reviewing the `traceroute` data for a set of subnets is sufficient to see how often a neighborhood located at most one hop away from one or more other neighborhoods can be found. Assuming most paths towards subnets ends with a well identified interface rather than an anonymous hop [5], a census of all such penultimate hops should be done before reviewing each route to check if any interface found before its penultimate hop corresponds to any of the other penultimate hops found in other paths. The interface that is the closest to the penultimate hop which satisfies this condition can be considered as a *peer*, i.e., the neighborhood built on the basis of this interface will be next to the neighborhood identified by the penultimate hop. If the difference in hop count between the *peer* and the penultimate hop is equal to one, then both associated neighborhoods are located one hop away from each other.

[4]Careful readers will remark the notion of *trail* from Chapter 6 could be reused here. Sec. 10.3.2 will involve it.

[5]This would correspond, in `WISE` terminology, to an anomaly-free trail.

To illustrate this notion of *peer*, Example 10.2 provides a sample of `traceroute` paths towards subnets of AS224 collected on September 1st, 2017 with `TreeNET` from the PlanetLab testbed [6]. The penultimate hop for the two first subnets is `78.91.96.0`, which also happens to be the interface found just before the penultimate hop for the third subnet. `78.91.96.0` is therefore a peer of `78.91.96.49`. As a consequence, the neighborhood bordered by subnets `78.91.96.12/31` and `78.91.96.14/31` will be found next to the neighborhood bordered by `78.91.96.16/31`, precisely one hop sooner.

Listing 10.2: Sample of `traceroute` paths for subnets of AS224 (September 1st 2017)

```
78.91.96.12/31: ..., 128.39.46.86, 78.91.96.0
78.91.96.14/31: ..., 128.39.46.86, 78.91.96.0
78.91.96.16/31: ..., 128.39.46.86, 78.91.96.0, 78.91.96.49
```

If peers can be found one hop away from penultimate hops on a regular basis, then it becomes possible to build a graph where vertices model neighborhoods and where each edge models the fact that the two connected neighborhoods are next to each other in the topology. By following the *peers* of each neighborhood, the graph can be built step by step, and this while using only the most useful portions of each `traceroute` path. To support this intuition, a subset of the ASes measured with `TreeNET` in early 2017 (cf. Sec. 4.3.1) were measured again on September 1st, 2017 from the PlanetLab testbed, and the collected subnets and paths were parsed to see how close peers were on average [7].

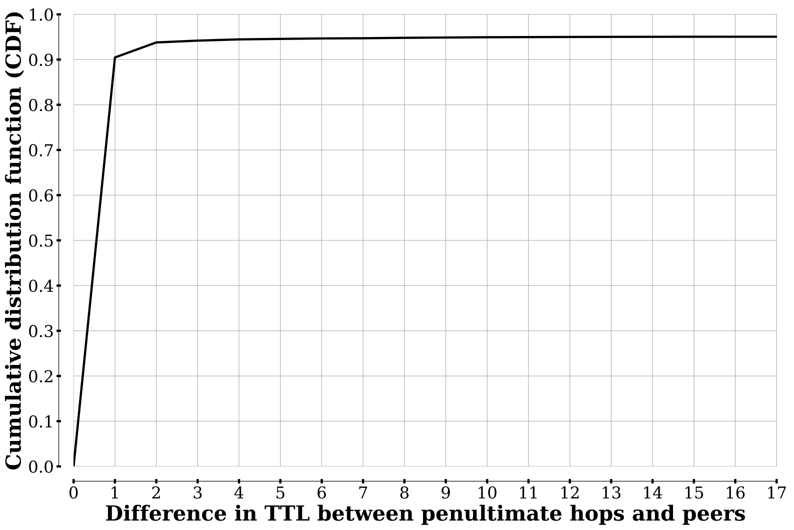


Figure 10.5: Distance in TTL between penultimate hops and their *peers* (September 1st, 2017).

Figure 10.5 presents, as a CDF, the distribution of the difference in TTL between a penultimate hop and a peer in all paths collected on September 1st, 2017, regardless of the target network. The curve does not reach 1 for practical reasons: paths whose penultimate hops were anonymous were

[6]Cf. `https://github.com/JefGrailet/SAGE_beta/tree/master/Motivations/AS224/2017/01-09`
[7]Cf. `https://github.com/JefGrailet/SAGE_beta/tree/master/Motivations`

ignored, and some paths did not include any peers. Being unable to find a peer in a `traceroute` path simply means the associated subnet(s) come first in the topology from the perspective of the vantage point or are in a different location. Hopefully, the curve already reaches 0.9 for a difference of one hop, and only slightly rises starting from a difference of 2 hops. This suggests that, in a vast majority of cases, it is possible to detect neighborhoods and find their direct adjacencies by carefully analyzing the `traceroute` data. Of course, such a task presents several challenges, but these early observations were encouraging enough to warrant exploring this research direction. Before designing algorithms for building graphs, Sec. 10.3.2 first defines precisely the concepts involved in such algorithms and Sec. 10.3.3 assesses them thoroughly with the help of `WISE` [52] (cf. Chapter 7).

### 10.3.2 The building blocks of a graph: neighborhoods and their peers

The definition of ***neighborhood*** will first be reviewed. A neighborhood is formally defined as a network location bordered by a group of subnets located at most one hop away from each other (see also Sec. 4.1.1). As such, a neighborhood is in practice a single router or a mesh of routers that might be connected together with Layer-2 equipment, e.g., Ethernet switches. As a result, building a graph of neighborhoods amounts to building a map of the hop-level of the target network (cf. Sec. 9.3.1), which can be seen as an approximation of the router-level of the same network. As explained in Sec. 10.3.1, neighborhoods can be identified simply by aggregating subnets whose `traceroute` paths all end with the same penultimate hop, an operation which has already been performed by the `TreeNET` methodology (cf. Sec. 10.1.1) but with an additional constraint on the paths (i.e., they must have the same length). If the terminology of subnet inference (cf. Sec. 5.1.2) and the concept of trail (cf. Sec. 6.1.1) are re-used, the so-called penultimate hop can be redefined as the trail towards the pivot interface(s) of a subnet. For the rest of this thesis, the trail of the pivot interface(s) of a subnet will be referred to as the ***trail of this subnet***. Therefore, a neighborhood can be inferred by aggregating subnets that feature the same trail. Such a neighborhood is then said to have been *identified by the trail* of these subnets. Figure 10.6 depicts a neighborhood as redefined with the `WISE` terminology.

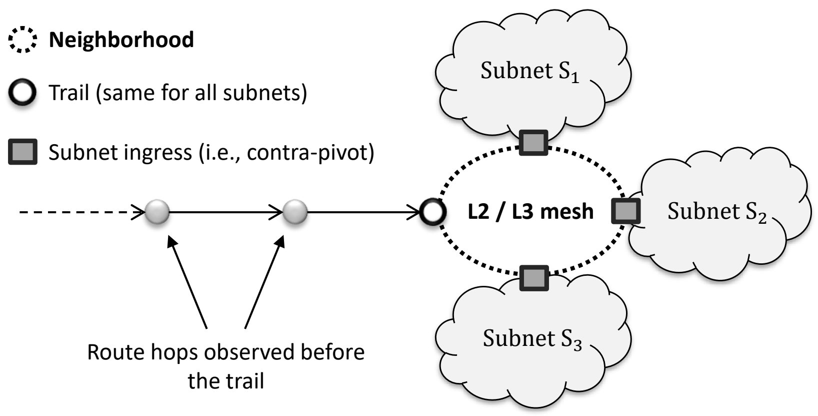

Neighborhood
Trail (same for all subnets)
Subnet ingress (i.e., contra-pivot)
L2 / L3 mesh
Subnet $S_1$
Subnet $S_2$
Subnet $S_3$
Route hops observed before the trail

Figure 10.6: The concept of neighborhood redefined with the `WISE` terminology.

While Figure 10.6 implies the depicted trail is anomaly-free (cf. Sec. 6.1.1), the detection of neighborhoods on the basis of trails is easily generalized to trails with anomalies. Subnets whose trails feature anomalies can still be aggregated on this basis, but with the implication that the discovered neighborhood might be larger than it actually is. If there is, for instance, one anomaly in a trail common to a set of subnets, it is possible there are multiple ingress routers, as the last non-anonymous hop only belongs to a router preceding the actual last hop router(s) (as depicted earlier in Fig. 6.1). This is why neighborhoods identified by trails featuring anomalies will be subsequently denoted as ***best effort neighborhoods***, since they are only an approximation of the actual neighborhood due to the impossibility to clearly identify the last hop before the bordering subnets.

Additionally, there are two special cases of neighborhoods in the context of `WISE` terminology: subnets with flickering trails and subnets featuring echoing trails, i.e., respectively built with the fourth and the third rule of subnet inference (cf. Sec. 7.4.1). The former is only a small exception in respect of the definition of trail for a subnet: any subnet displaying flickering trails (which are anomaly-free by definition, cf. Sec. 6.1.3) within its address space can be considered as having multiple trails. As a consequence, subnets with flickering trails can be aggregated on the basis of the alias list including their trails: as long as one of their trails appears on the alias list, it can be assumed that they border the same neighborhood as any other subnet whose trail also appears on the alias list. Such a neighborhood is therefore said to be *identified by an alias list* (rather than a single trail).

Subnets featuring echoing trails also do not feature a single trail, as, by definition, all their pivot interfaces have unique trails. To make up for this problem, these subnets are aggregated on the basis of their ***pre-echoing hop(s)***, i.e., the router interface(s) that were observed just before the echoing trails during the target scanning (cf. Sec. 7.3.2). Indeed, by design, `WISE` always probes at least one hop prior to an (allegedly) echoing trail to ensure it is not a cycle stretching over multiple hops. This additional hop can be used to aggregate subnets built on the third rule of inference. Therefore, the neighborhoods built on this basis can be deemed as *best effort* too, as they are equivalent, in practice, to aggregating subnets with trails bearing anomalies, and are *identified by these pre-echoing hop(s).*

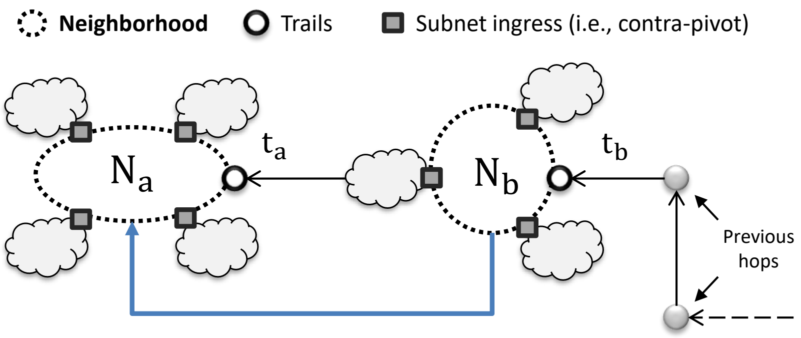


Figure 10.7: The concept of peer in neighborhood-based topology discovery. $N_b$ is a peer of $N_a$.

The notion of *peer* first hinted at by Sec. 10.3.1 will next be presented. In the context of neighborhood-based topology discovery, the concept of ***peer*** can be formally defined as follows: for a given neighborhood $N_a$ identified by a trail $t_a$, a second neighborhood $N_b$ identified by a trail $t_b$ is a peer of $N_a$ if and only if $t_b$ appears prior to $t_a$ in the `traceroute` paths towards the pivot interfaces of the subnets bordering $N_a$. This formal definition is depicted in Figure 10.7. By design, a trail can be a peer to a neighborhood if and only if it is anomaly-free. Therefore, as will be elaborated in Chapter 11, best effort neighborhoods can only appear as endpoints within a neighborhood-based graph.

Since it is not always possible to find a peer immediatly next to a trail in the `traceroute` data, the concept of peer has also been given a best effort equivalent: ***remote peers***. Remote peers simply describe peers that are located several hops prior to the trail(s) used to identify a neighborhood, and need to be treated differently to regular peers while building a graph. Non-remote peers can also be called ***direct peers***, while a neighborhood without peer(s) can be ruled as ***peerless***.

### 10.3.3 An evaluation of neighborhoods and their peers

Before designing an algorithmic solution for building a neighborhood-based DAG (as will be done in Chapter 11), the concepts of neighborhood and peer as defined by Sec. 10.3.2 must be assessed carefully. To do so, the subnet inference tool `WISE` [52, 53], previously described in Chapter 7 and assessed in Chapter 8, is extended to discover neighborhoods and their peers after first discovering subnets. Sec. 10.3.3.1 briefly describes how `WISE` has been modified for this purpose and how it has been deployed. Then, Sec. 10.3.3.2 thoroughly validates the neighborhoods detected by `WISE` on two groundtruth networks, and Sec. 10.3.3.3 carefully evaluates the concept of peer in the wild.

#### 10.3.3.1 Methodology

Neighborhood inference in `WISE` starts at the end of the subnet inference, i.e., after subnet post-processing (as described in Sec. 7.4.2). In order to efficiently detect neighborhoods and their peers, `WISE` begins by collecting (partial) `traceroute` paths towards subnets that are long enough to later discover peers. This is done in two steps: first, `WISE` makes a census of all IP addresses that correspond to the anomaly-free trails of pivot interfaces found in any previously inferred subnet: these addresses will be later used to discover neighborhoods, and therefore will subsequently also act as peers. For convenience, these IP addresses will be denoted as ***peer addresses***. Second, for each subnet, up to five pivot interfaces are selected [8], and `WISE` schedules new backward Paris `traceroute` measurements towards each, this process being identical to the one used during target scanning (cf. Sec. 7.3.2). There is however a difference: `WISE` stops a `traceroute` measurement as soon as it gets a reply from an IP address that matches a peer address. Indeed, from that point, `WISE` has already collected enough data to discover a peer. This also reduces the overhead probing, as there will be only one additional probe to discover direct peers. A `traceroute` measurement is only fully performed if and only if `WISE` cannot find a peer address among the router interfaces that reply to the `traceroute` probes.

After this additional probing, `WISE` can infer neighborhoods by aggregating subnets on the basis of their trails, though it performs additional operations when it has to aggregate subnets featuring

[8]Five is a default value chosen to have representative paths for large subnets; it can be modified upon running `WISE`.

flickering or echoing trails to comply with the previously recommended best practices (cf. Sec. 10.3.2). Once the neighborhoods have been formed, `WISE` can rely on the freshly collected `traceroute` data to enumerate the peers of each one. In other words, much like the subnet inference itself, `WISE` detects neighborhoods and their peers offline, i.e., after the additional `traceroute` measurements. Since the neighborhood inference step in `WISE` was kept more or less unchanged in `SAGE` [48], the topology discovery tool that implements neighborhood-based DAG discovery, the algorithmic details of neighborhood inference and peer discovery will be provided in Chapter 11.

The upgraded `WISE` [9] was deployed on the PlanetLab testbed as early as September 2019. After implementing some algorithmical improvements for peer discovery, an additional campaign was conducted between November and December 2019 [10] while targetting the same ASes as previously (cf. Sec. 8.2.1) to collect enough data to accurately assess the concept of peer, this point being discussed in Sec. 10.3.3.3. The campaign was planned exactly like previous `WISE` campaigns (with vantage point rotation, cf. Sec. 6.2.1) in order to make the most of the available PlanetLab nodes. In addition to these measurements in the wild, the same groundtruth networks as described in Sec. 8.1.1 were again measured with `WISE` to validate the inferred neighborhoods. The results of this validation are discussed in the next section (Sec. 10.3.3.2).

#### 10.3.3.2 Validation of neighborhoods

In order to ensure the concept of neighborhood is sound with respect to actual networks, new measurements of the same groundtruth networks as presented in Sec. 8.1.1 were conducted in late September 2019, with a change: this time, the entire topology of the Belgian ISP was probed, rather than just its backbone. For each network, the groundtruth partners provided a list of all IPv4 prefixes associated with each router. By comparing the first IP addresses of these prefixes with the first IP addresses of the subnets inferred by `WISE`, it was possible to match the majority of prefixes with the inferred subnets and check how many prefixes associated to a given router were found around the same neighborhood in the collected data. For the sake of completeness, the groundtruth partners also provided details on how routers were grouped in their network to properly account for the few meshes [11], since a neighborhood can also consist of a mesh of routers (cf. Sec. 10.3.2).

To validate the discovered neighborhoods, every possible pair of prefixes identified in both the groundtruth networks and the `WISE` measurements will be considered and classified. A pair found both around the same router ID (or distinct IDs from a known mesh) and around the same neighborhood is classified as a true positive. A pair appearing around the same neighborhood but not around the same router (or mesh) is categorized as a false positive. Likewise, two prefixes that are not around either the same router (or mesh) or around the same neighborhood constitute a true negative, and a pair found around a same router (or mesh) but not around the same neighborhood is ruled as a false negative. To complete the analysis, a router whose all known prefixes appear exactly in the same

[9] As always, the source code is publicly available on GitHub: `https://github.com/JefGrailet/WISE`.
[10] Cf. `https://github.com/JefGrailet/WISE/blob/master/Dataset/Campaign10.md`
[11] As always, the groundtruth data will not be made public for security concerns.

| | **ULiège network** | **Belgian ISP** |
|---|---|---|
| True positives rate | 77.27% | 70.06% |
| True negatives rate | 99.33% | 99.83% |
| False positives rate | 0.67% | 0.17% |
| False negatives rate | 22.73% | 29.94% |
| Accuracy | 98.13% | 96.13% |
| Matched routers | 60 | 91 |
| Matched neighborhoods | 45 | 133 |
| Exact matches | 26 (45.76%) | 48 (52.75%) |
| *Best effort* neighborhoods | 1 (2.22%) | 106 (79.70%) |

Table 10.1: Validation of neighborhoods on two groundtruth networks.

inferred neighborhood is referred to as a ***exact match***. Finally, due to the Belgian ISP ground truth network having a high number of *echoing* devices, many subnets found within it were discovered through the 3rd subnet inference rule (cf. Sec. 7.4.1), and, consequently, most of its neighborhoods can be deemed as best effort (cf. Sec. 10.3.2). To account for this, the ***best effort ratio*** provides the ratio of best effort neighborhoods in respect of the total number of neighborhoods.

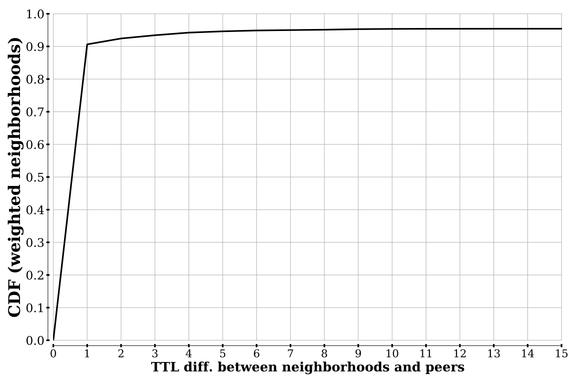
Table 10.1 provides the results of the validation [12]. The first major result is the very low false positive rate: less than 1% in both situations, demonstrating how `WISE` infers neighborhoods rarely produces atypical results in respect of the groundtruth topologies. On the contrary, as demonstrated by the false negative rate, such a methodology is rather pessimistic, but still accounts for 77% of true positives for the ULiège network and 70% for the Belgian ISP. Careful analysis of the measurements showed that false negatives are typically the results of incomplete subnets missing from large neighborhoods due to measurement issues. For instance, a subnet can be observed at the end of an unique path (therefore with a unique trail), or with a trail that includes anomalies due to temporary network failures. In some cases, only the contra-pivot interface of a subnet could be seen during measuring and was falsely identified as a pivot interface. As a consequence, if the subnet was not associated with a new neighborhood, it was erroneously matched with another neighborhood, creating some false positives. Nevertheless, these results demonstrate that the neighborhood inference offers a first good approximation of the topology, with around half of the devices being fully matched with exactly one neighborhood in both cases. This was a particularily good result in the case of the Belgian ISP: as almost 80% of the neighborhoods were inferred with *best effort* strategies, the regular (and more reliable) approach could not have been used, therefore showing how a *best effort* methodology can offer good results too.

#### 10.3.3.3 Peers in the wild

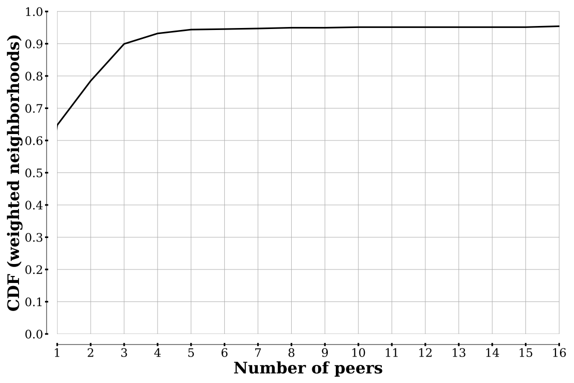
In addition to a consistent neighborhood inference, building the neighborhood-based DAG of a target network also requires the ability to locate neighborhoods in respect of each other. The concept of peer (cf. Sec. 10.3.2) is the main tool for achieving this. However, before designing algorithms relying

[12]The accuracy and rate metrics are identical to those of Sec. 4.2.

on this notion to build graphs systematically, it should be checked whether discovering peers in the wild is a regular occurrence, and if so, how often these peers are direct (i.e., the difference in TTL distance with the associated neighborhood is equal to one). In other words, the intuitions discussed in Sec. 10.3.1 must be confirmed. Indeed, if peers are actually difficult to find and/or are remote most of the time, then building a graph by relying on the concepts described in Sec. 10.3.2 might not be worth the effort. Hopefully, using snapshots of 12 distinct ASes captured with the upgraded `WISE` (cf. Sec. 10.3.3.1) from the PlanetLab testbed between November 19th and December 6th, 2019, it quickly became apparent that most peers detected in the wild are direct. Furthermore, the number of peers identified for each neighborhood can vary significantly. This section supports these claims by providing representative results depicted with figures, but all figures generated with the discussed snapshots can be found online [13].



(a) Distance of peers



(b) Number of peers

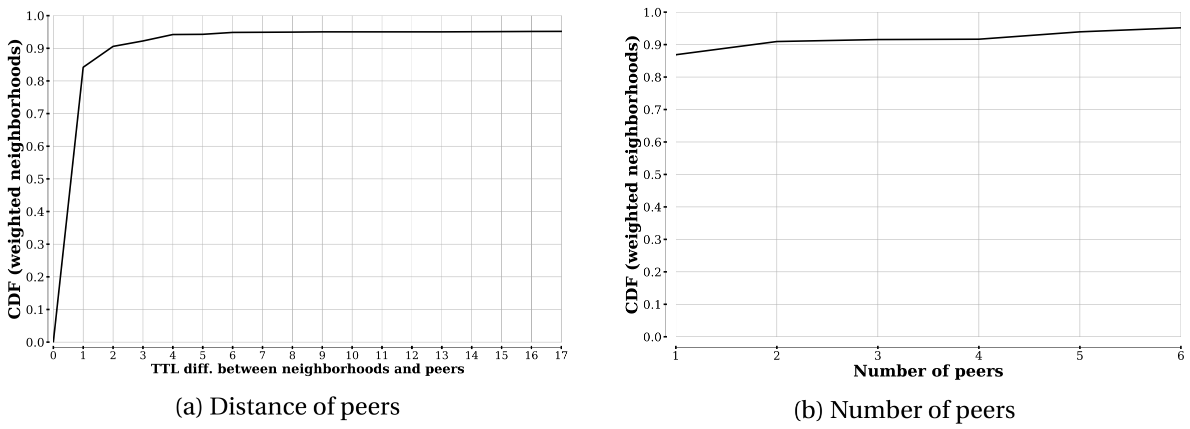
Figure 10.8: Assessing the concept of peer with 12 snapshots of AS6453 (Fall 2019).

Figure 10.8 shows an evaluation of peers for AS6453 (TATA Communications), a Tier-1 AS, using all 12 snapshots captured with `WISE` from PlanetLab (using a different vantage point for each snapshot) between November 19th and December 6th, 2019. Fig. 10.8a depicts the typical differences in TTL distance between each inferred neighborhood and its peer(s) as a cumulative distribution function. It should be noted that the results were weighted: because there were a certain number of artefacts in the data (i.e., neighborhoods discovered with one or two subnets that could not be considered as sound), the neighborhoods were weighted with the number of subnets bordering them while counting the total number of neighborhoods having a given TTL difference with their peers. Neighborhoods identified with many subnets, which can be deemed as more credible, were therefore better represented. Fig. 10.8b illustrates the number of peers per neighborhood in the same manner, again with the values for each neighborhood being weighted with the number of bordering subnets.

Given the weighting, Fig. 10.8a can be interpreted as follows: for a subnet picked at random in all 12 snapshots, the chances of said subnet bordering a neighborhood featuring direct peer(s) are slightly above 90%. This is an important result, especially given that each snapshot was captured

[13]Cf. `https://github.com/JefGrailet/WISE/tree/master/Evaluation/Neighborhoods/Figures`

from a distinct vantage point and given that AS6453 is a Tier-1 AS featured in the top 10 of CAIDA's AS ranking [2]. This shows that, even for topologies that are difficult to measure, both the concepts of neighborhood and peer are viable enough to use them as tools for building a neighborhood-based DAG. Similar results were obtained with other Transit/Tier-1 ASes, such as AS3257 (though the chances dropped from 90% to more than 80%). Fig. 10.8b shows the average subnet (i.e., around 65% of subnets) borders a neighborhood featuring only one peer, but a considerable number of subnets rather border neighborhoods with between two and eleven peers. While this could suggest some neighborhoods feature many predecessors in the topology, it should be kept in mind that, at this stage, peers only consist of isolated IP addresses corresponding to anomaly-free trails (cf. Sec. 10.3.2). As a consequence, to guarantee these peers do belong to separate devices, therefore neighborhoods, an alias resolution scheme should be applied to the peer addresses gathered for each neighborhood featuring multiple peers. How this can be done in practice will be covered in Chapter 11. The variations in the number of peers nevertheless suggest the very concept of peer is suitable to overcome the limitations of the `TreeNET` methodology (cf. Sec. 10.1.2). Interestingly, the distributions do not reach exactly 1 as in Fig. 10.5: the differences with the highest values in both distributions simply account for peerless neighborhoods.

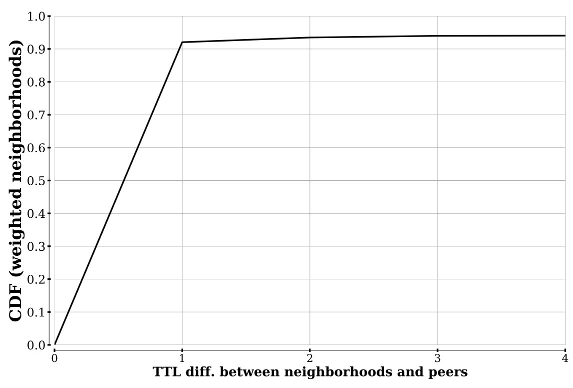


(a) Distance of peers (b) Number of peers

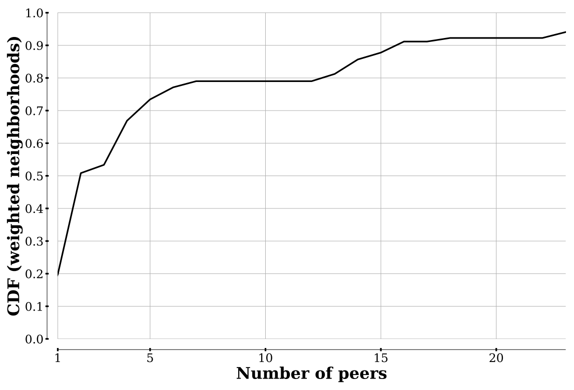
Figure 10.9: Assessing the concept of peer with 12 snapshots of AS286 (Fall 2019).

Figure 10.9 provides the same data visualization as Figure 10.8 for the 12 snapshots collected in Fall 2019 for AS286 (KPN B.V.), a Dutch Transit AS. The results are comparable to those of Figure 10.8, except that the ratio of subnets bordering neighborhoods with direct peers slightly drops to around 85% in Fig. 10.9a. Interestingly, the number of peers per neighborhood is frequently equal to one, as shown by Fig. 10.9b, though there is a noticeable ratio of subnets whose neighborhood has from two to six peers, showing that neighborhoods with multiple peers are not uncommon. This also further shows `TreeNET`'s tree model is not well suited for measuring large networks in the wild.

In fact, neighborhoods with multiple peers can sometimes be a common occurrence. Figure 10.10 depicts with the same visual tools as before the results for AS1241 (Forthnet), a Greek ISP (Stub AS), again using all 12 snapshots of this AS captured in Fall 2019. While Fig. 10.10a shows how a vast majority of the discovered peers were direct peers, Fig. 10.10b surprisingly shows that only 20% of



(a) Distance of peers



(b) Number of peers

Figure 10.10: Assessing the concept of peer with 12 snapshots of AS1241 (Fall 2019).

subnets border neighborhoods with a single peer, and that slightly less than 50% of them border neighborhoods with more than three peers. Such a large number of peers per neighborhood could hide many adjacencies between neighborhoods as well as hide several convergence points (as defined in Sec. 2.4). A glance at the snapshots shows the subnets discovered on AS1241 feature many (aliased) flickering trails, therefore giving more credence to the latter hypothesis. However, such an hypothesis could only be confirmed by performing alias resolution on peer addresses, as suggested previously, to determine whether the discovered peers correspond to distinct devices or to a same device reached through multiple paths. Therefore, a thorough graph building scheme should involve both the concepts elaborated in Sec. 10.3.2 and alias resolution.

## 10.4 Closing comments on `traceroute` paths interpretation

Interpreting `traceroute` paths to build an accurate map of some target topology is a challenging task that has kept the research community busy since the dawn of Internet topology discovery, as discussed in Chapter 9. What makes this task so challenging with today's Internet is how `traceroute` paths are constantly being influenced by load balancing [12] and other routing policies (such as rate-limiting), as already discussed in Sec. 9.1.1. This notably results in variations in the observed distances for specific router interfaces or observing different router interfaces on the way to some specific, well-narrowed network location, i.e., paths are rarely stable.

The topology discovery tool `TreeNET` [51, 55] (cf. Sec. 10.1) provides a topology mapping methodology relying on the idea that a collection of `traceroute` paths towards close locations (subnets, in this instance) should be close to a tree, i.e., the first router interfaces in the paths should be the same and start to diverge when going deeper into the topology of the target network. While this strategy can provide a good picture with simple (or close) networks where little traffic engineering is experienced, such as the ULiège network (cf. Sec. 4.2), it is easily impaired by the consequences of load balancing, typically asymmetrical paths and convergence points [14]. While `TreeNET` partially makes up for the consequences of load balancing architectures with its concept of multi-label nodes (cf. Sec. 10.1.1), such a strategy relies too heavily on alias resolution to break down multi-label nodes into multiple predecessors in the topology, as in the `traceroute` data.

An approach to address `TreeNET`'s shortcomings consists of pre-processing the `traceroute` paths to mitigate the effects of not only traffic engineering but also rate-limiting, as anonymous hops also constitute a hurdle for building the tree-like mapping of a network. The `traceroute` extension `RTrack` [46] (cf. Sec. 10.2) was designed to experiment with such a pre-processing while also quantifying `traceroute` issues in the wild. While its approach for mitigating varying TTL distances for specific interfaces (i.e., *route stretching*, cf. Sec. 10.2) could be validated on the topology of a Belgian ISP, it has a tendency to hide interesting parts of a target network. In the case of the Belgian ISP, it completely hid some back-up links which can play a key role in the internal routing of a network. Moreover, though the heuristics implemented by `RTrack` can help `TreeNET` to some extent, they do not address the problem of having each hop modeled in the tree-like mapping with a unique predecessor, which usually takes the shape of a multi-label node when there are actually several predecessors in the topology.

The solution chosen during the research work conducted for this thesis therefore consists of keeping the concept of neighborhood while expanding it. Rather than creating a structure that would eventually lead to the inference of neighborhoods (like the `TreeNET` methodology), the idea was to go the other way around and build the neighborhoods first, then find how they are located in respect of each other. To achieve this, the notion of neighborhood was complemented with the notion of trail introduced with `WISE` [52] (see also Sec. 6.1.1). The concept of peer was created (cf. Sec. 10.3.2) as a tool to discover neighborhood adjacencies [53]. `WISE` has been subsequently modified to be able to infer neighborhoods and their respective peers after subnet inference, so that both concepts could be extensively evaluated (cf. Sec. 10.3.3). A validation on two groundtruth networks and a study of network snapshots collected in the wild from the PlanetLab testbed demonstrated that both concepts were sound enough to design algorithms to build neighborhood-based **d**irected **a**cyclic **g**raphs (DAGs) [53], i.e., structures which are by design more faithful to the adjacencies observed in `traceroute` data than the tree paradigm [98].

This approach not only overcomes one of the main limitations of `TreeNET`, but it is also more suitable for dealing with consequences of traffic engineering. Indeed, much like the `WISE` methodology for subnet inference, both the concepts of neighborhood and peer (as described in Sec. 10.3.2) do not imply any requirements in terms of TTL distance or `traceroute` path straightness, i.e., always discovering the same interfaces while probing a remote location. The next chapter (Chapter 11) introduces `SAGE` [54], a topology discovery tool built on top of `WISE` that takes advantage of the aforementioned concepts to systematically discover the hop-level topology of a target network in the shape of a neighborhood-based DAG.

# CHAPTER 11

# SAGE: SUBNET AGGREGATION

This chapter introduces a new topology discovery tool that systematically builds neighborhood-based (cf. Sec. 10.3.2) **d**irected **a**cyclic **g**raphs (DAGs): `SAGE` [54] (for **S**ubnet **AG**gr**E**gation). As such, it addresses research question 2 from Sec. 9.4. Sec. 11.1 first reviews the various challenges `SAGE` must carefully overcome in order to easily and accurately build a neighborhood-based DAG, and also the opportunities it opens. Sec. 11.2 then presents the workflow of `SAGE`, with a focus on the inner workflow of the steps needed to build the DAG. Sec. 11.3 subsequently describes in detail the algorithms used to build the DAG. Sec. 11.4 then comments on using the DAG for alias resolution. Finally, Sec. 11.5 concludes this chapter by summarizing the benefits of `SAGE` and its perspectives for topology mapping and modeling.

## 11.1 `SAGE` challenges and opportunities

The purpose of `SAGE` will be first described. As stated at the end of Sec. 10.4, the purpose of this new tool was to algorithmically build the ***neighborhood-based DAG*** of the target network it probes, starting with subnet-level data it first gathered with the `WISE` [52, 53] methodology (cf. Chapter 7). In such a graph, the vertices model individual neighborhoods while edges account for the links that exist between them. Indeed, since neighborhoods correspond in the real world to individual routers or meshes of routers, the links which exist between routers within neighborhoods also constitute links between these neighborhoods. More broadly, neighborhoods can also be interpreted as individual hops from the perspective of a measurement tool, and therefore, a neighborhood-based DAG can also be viewed as a directed ***hop-level graph*** (cf. Sec. 9.3.1). In practice, the direction of the edges only account for the predecessor/successor relationship that exists between distinct hops in the `traceroute` data, since network links go both ways.

Typically, when a neighborhood $N_b$ is a (direct) peer of a neighborhood $N_a$ (cf. Sec. 10.3.2), the corresponding vertices in a neighborhood-based DAG will feature a common edge. In addition to accounting for how neighborhoods are located in respect of each other, a neighborhood-based DAG can also be annotated to provide a more complete picture of the topology. Indeed, because the links between routers usually consist of subnets, it may be possible to infer which discovered subnet corresponds to which edge. The very idea of discovering which subnets correspond to each network link has been already explored by `TraceNET` [116] (cf. Sec. 5.2.1), the main difference being that `TraceNET` was designed to infer subnets that accommodate the interfaces observed in `traceroute` data while `SAGE` is supposed to already have subnet-level data upon building a graph by design. Moreover, since an edge in a neighborhood-based DAG only accounts for a relationship between two neighborhoods, subnets that act as LANs between multiple neighborhoods can be, in practice, mapped to several edges. Figure 11.1 pictures a toy example of an annotated neighborhood-based DAG. In this example, the subnet $S_3$ is mapped to two edges in the DAG, suggesting $S_3$ is in practice a small LAN where all three $N_2$, $N_3$ and $N_4$ can mutually reach each others directly.

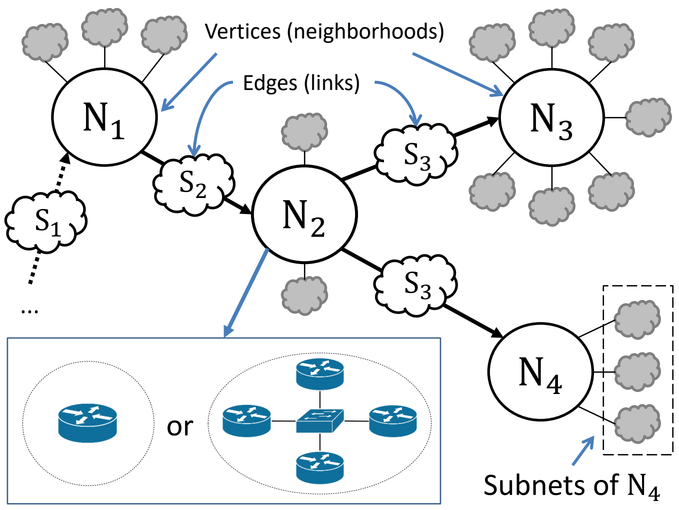


Figure 11.1: Toy example of an annotated neighborhood-based DAG.

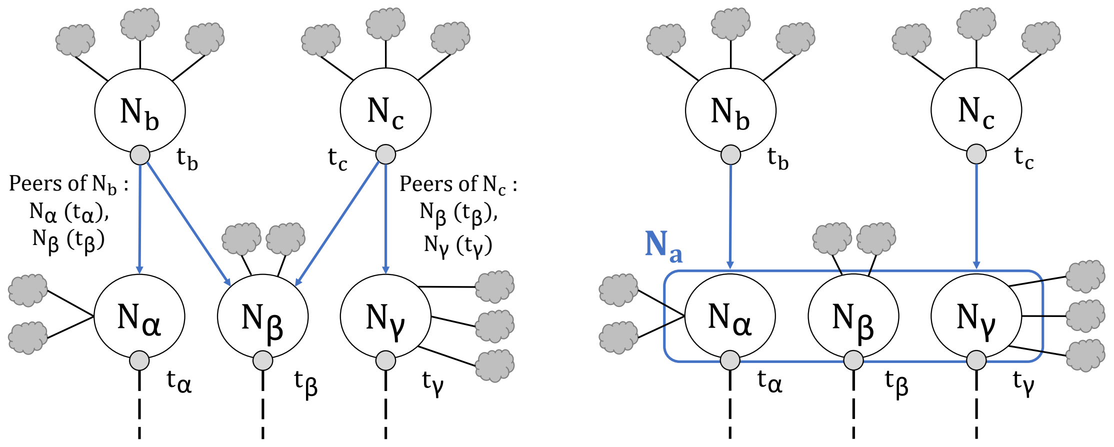
Annotating a neighborhood-based DAG with previously discovered subnets is not just a convenient addition: it is meant to already prepare the graph for its subsequent transformation into a bipartite graph. Indeed, because subnets can bind two or more routers in a network topology, they make a convenient first party for a bipartite formalism. Bipartite graphs are out of the scope of this chapter, but will be more extensively detailed in Chapter 13.

Moreover, before considering bipartite graphs, several challenges must be overcome to accurately build a neighborhood-based DAG to begin with. Sec. 11.1.1 covers the problem of identifying whether multiple peers listed for a neighborhood belong to the same device, as peers initially show as IP addresses, also denoted as ***peer addresses*** (cf. Sec. 10.3.3.1). Indeed, a topology can very well feature one or more convergence points (cf. Sec. 2.4) due to symmetrical load balanced paths. Sec. 11.1.2

proceeds with the detection of unidentified peers that could not initially be mapped with a neighborhood in the first place, also called *blindspots*. Finally, Sec. 11.1.3 reviews how the edges of the final graph can be mapped with subnets but also categorized for convenience.

### 11.1.1 Identification of convergence points

The first challenge SAGE must tackle is the detection of convergence points. Indeed, when a neighborhood features several peers (which are by definition anomaly-free trails, cf. Sec. 10.3.2), there is no guarantee that the IP addresses associated with these peers (i.e., peer addresses) are strictly from distinct devices. For instance, a neighborhood can first appear as several smaller neighborhoods because the probes have reached the interfaces used to identify them (i.e., the associated trails) from distinct paths instead of always reaching the same interface (therefore the same trail). In other words, it is possible a neighborhood acts as a ***convergence point***, i.e., a device where distinct (possibly load balanced) paths converge and which is best identified by an alias list. Such a device can be compared to the ingress router of subnets featuring flickering trails (cf. Sec. 6.1.3) and can therefore be detected, in practice, with the help of alias resolution, as already discussed in Sec. 2.4.



$N_b$
$N_c$
$t_b$
$t_c$
Peers of $N_b$ : $N_\alpha$ ($t_\alpha$), $N_\beta$ ($t_\beta$)
Peers of $N_c$ : $N_\beta$ ($t_\beta$), $N_\gamma$ ($t_\gamma$)
$N_\alpha$
$N_\beta$
$N_\gamma$
$t_\alpha$
$t_\beta$
$t_\gamma$
$N_a$

(a) Before alias resolution

(b) After alias resolution

Figure 11.2: Discovering a convergence point with SAGE.

To address this challenge, SAGE must identify the router interfaces that might belong to an hypothetical convergence point and conduct alias resolution on them. An elegant solution to this problem consists of anticipating the largest sets of alias candidates that might correspond to (distinct) convergence points and resolving them with the help of the fingerprint-based framework introduced by Chapter 3. After listing the peers for all neighborhoods, the lists featuring multiple peers can be collected and merged them when they share common interfaces, relying on the property of ***alias transitivity*** (first defined in Sec. 2.2). As a reminder, this property works as follows: given three interfaces *A*, *B*, and *C*, if *A* and *B* are aliases and if *B* and *C* are aliases too, then *A* and *C* are aliases as well. Thanks to this trick, the largest hypothetical alias lists only need to be resolved once to avoid verifying one alias pair multiple times while reviewing the peers of each neighborhood. When an alias pair/list is discovered, this means a convergence point has been discovered, and the neighborhoods identified by each alias candidate/peer address can be subsequently merged together.

Fig. 11.2 shows a toy topology where SAGE should discover a convergence point by applying subsequently alias transitivity and alias resolution (with the framework of Chapter 3). In this toy example, the two neighborhoods $N_b$ and $N_c$ initially have the lists of peers (as IP addresses) $t_\alpha, t_\beta$, and $t_\beta, t_\gamma$, respectively (Fig. 11.2a), given that $t_\alpha$, $t_\beta$ and $t_\gamma$ each identify a separate neighborhood. By aggregating these peers to create the largest hypothetical alias $\{t_\alpha, t_\beta, t_\gamma\}$, SAGE can recover the (initially split) neighborhood $N_a$ depicted in Fig. 11.2b as long as $t_\alpha$, $t_\beta$, and $t_\gamma$ are aliased together.

### 11.1.2 Detection of blindspots (unidentified peers)

There is another challenge that lies in the detection of convergence points. In fact, the convergence point detection presented in the previous section (Sec. 11.1.1) assumes that each IP address found in the `traceroute` data will identify a neighborhood. Of course, this is not always verified in practice, and it is possible that an interface that could belong to a convergence point does not appear as a trail for any subnet and therefore cannot be considered either as a potential peer address or as a tool to identify a neighborhood. This can be a problem if such an interface still appears in the `traceroute` data, because it can induce some neighborhoods to have remote peers despite being one hop away from a convergence point in the real topology. In SAGE terminology, such an interface is referred to as a ***blindspot***. More broadly, router interfaces not identifying neighborhoods that appear in the `traceroute` data used to discover peers are called ***miscellaneous hops***. In other words, blindspots constitute a subset of miscellaneous hops.

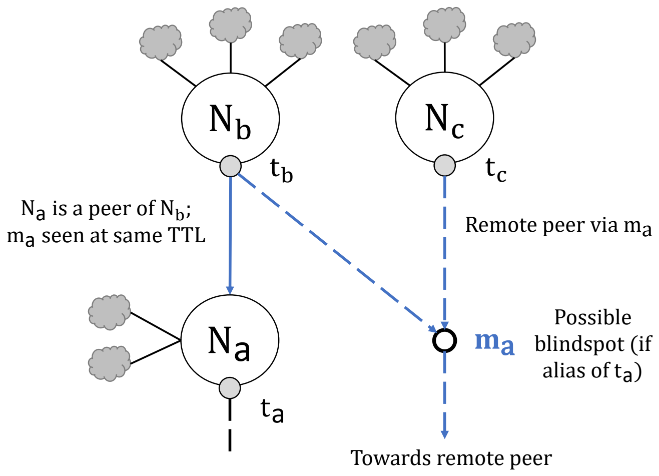

$N_b$
$N_c$
$t_b$
$t_c$
$N_a$ is a peer of $N_b$; $m_a$ seen at same TTL
Remote peer via $m_a$
$N_a$
$m_a$
Possible blindspot (if alias of $t_a$)
$t_a$
Towards remote peer

Figure 11.3: Because it does not identify a neighborhood, $m_a$ could constitute a blindspot.

Figure 11.3 depicts a toy scenario where the blindspot issue could arise, reusing the same visual tools as Fig. 11.2. In the figure, the miscellaneous hop $m_a$ is not initially identified as a peer address but appears at the same TTL distance as $t_a$ (which identifies neighborhood $N_a$) in the `traceroute` data used to discover peers of $N_b$. Because the `traceroute` data used to find the peers of $N_c$ does not contain any router interface other than $m_a$ at a distance of one hop, $N_c$ can only have remote peer(s) on paper. However, if $m_a$ could be relabelled as a blindspot by demonstrating it is actually an alias of some peer address, therefore showing it belongs to a convergence point, it would become feasible to use it as a direct peer of $N_c$.

Hopefully, blindspots can be detected to some extent: while listing the peers of each neighborhood, `SAGE` can also list the miscellaneous hops appearing at the same TTL distance and include them as additional alias candidates in the alias resolution process. If they end up on an alias pair/list with one or several peer addresses, these miscellaneous hops can be relabelled as blindspots. After detecting all potential blindspots, `SAGE` can review all neighborhoods with remote peers to verify whether these blindspots appeared earlier in the `traceroute` data so these neighborhoods can get closer (and ideally direct) peers. In Figure 11.3, successfully aliasing $t_a$ and $m_a$ together would lead to the discovery of one direct peer for $N_c$. The frequency of blindspots and their cost in terms of alias resolution will be commented on in Sec. 11.3.5.

### 11.1.3 Characterization of edges in a neighborhood-based DAG

One last issue `SAGE` should tackle, which is more an opportunity than a challenge, lies in mapping previously discovered subnets to the edges of the final graph. Indeed, in the real world, subnets can notably act as point-to-point links between two devices (typically, point-to-point links are /30 or /31 subnets). For each edge it infers, `SAGE` can look for the subnet that could correspond to the practical network link. Doing so, `SAGE` not only outputs a true picture of the hop-level adjacencies in the shape of a neighborhood-based DAG, but also complements it with subnet-level data, resulting in an annotated graph that accounts for the neighborhood – subnet topology of the target network. As mentioned in Sec. 11.1, this can be a first step towards a bipartite formalism involving subnets.

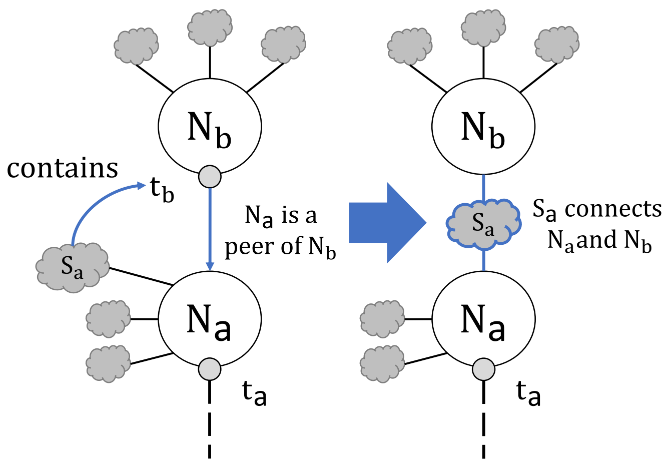


Figure 11.4: The subnet $S_a$ (bordering $N_a$) can be identifed as the link between $N_a$ and $N_b$.

In practice, this task simply consists of looking at the router interface(s) identifying a non-best effort neighborhood and finding the subnet whose prefix encompasses (one of) these interface(s). Such a subnet can therefore be mapped to one or several edges connecting this neighborhood to its direct peer(s). Fig. 11.4 shows a simple example: given two juxtaposed neighborhoods $N_a$ and $N_b$, respectively identified by router interfaces/anomaly-free trails $t_a$ and $t_b$, $N_a$ being closer to the vantage point, a subnet $S_a$ bordering $N_a$ encompasses $t_b$. As a consequence, $S_a$ can be considered as the subnet connecting $N_a$ and $N_b$ together.

In practice, the subnet which connects two neighborhoods does not always appear around the neighborhood closest to the vantage point, and can also border a third neighborhood, depending on the size of the subnet (a large subnet can be mapped to several edges) and/or the internal routing of the target network. This is why the subnet mappings are categorized in two in `SAGE` terminology:

- ***direct links*** are subnet – edge mappings whose subnet borders the neighborhood (connected by the edge) closest to the vantage point, and
- ***indirect links*** are subnet – edge mappings whose subnet borders a neighborhood that differs from the neighborhoods connected by the edge. Contrary to direct links, indirect links can have no mapping, i.e., they account for unmapped parts of the subnet-level. They also constitute the default solution to model edges towards best effort neighborhoods (cf. Sec. 10.3.2).

Direct links are so called because they naturally arise from both the subnet-level and the `traceroute` data. Indirect links might first be seen as a measurement artifact, but can be actually explained by several factors. For instance, a subnet might be used as a link in the topology while not being routed in the same manner, i.e., the probes used to discover the subnet do not take the same path as the probes crossing the same subnet while targeting another part of the network. Indirect links also encompass larger subnets which can appear in both direct and indirect links, and as such, can reveal new adjacencies between neighborhoods. However, indirect links can occasionally result from undetected convergence points, as will be discussed in Sec. 12.1.2 of Chapter 12.

Finally, as (in)direct links only account for edges between neighborhoods that are found one hop away from each others, a third kind of link is also used by `SAGE` to account for remote peers. ***Remote links*** are special edges that store all unique sequences of intermediate hops observed between a neighborhood and its remote peer(s) rather than storing a mapping with some subnet, since there is, by definition, more than one intermediate link with remote peers. The main motivation behind this mechanism is practical: this way, `SAGE` can keep track of intermediate `traceroute` paths without adding any additional vertex to the graph, as intermediate paths can sometimes be quite long and would require many intermediate vertices to be fully depicted – without any subnet bordering them. This way, a visual render of a graph as discovered by `SAGE` can highlight only the measured topology, as intermediate paths can notably be caused by detours that cross other networks that were not intended to be measured by `SAGE`.

## 11.2 SAGE workflow

As explained in Sec. 10.4, SAGE [54] is built on top of WISE [52, 53], i.e., it reuses exactly the same methodology to discover the subnets it requires to discover neighborhoods and build an (annotated) neighborhood-based DAG. After discovering the subnets, SAGE runs a fourth main algorithm step to build the neighborhood-based DAG that will be subsequently denoted as ***neighborhood inference***. Indeed, the main challenge for building the final graph consists of accurately discovering the vertices of the graph, i.e. neighborhoods, and how they are located in respect of each other. In particular, the detection of convergence points (cf. Sec. 11.1.1) is crucial to recover neighborhoods for which trail-based inference (cf. Sec. 10.3.3.1) is not sufficient. In addition to the neighborhood inference, SAGE also performs a fifth and final algorithm step, nicknamed ***full alias resolution***, which simply reuses the alias resolution methodology of TreeNET [51] to discover alias pairs/lists within each neighborhood (cf. Sec. 11.4). These additional alias pairs/lists can be used to later recover a router-level graph (cf. Sec. 2.1 or Sec. 9.3.1), as will be discussed in Sec. 13.4 of Chapter 13.

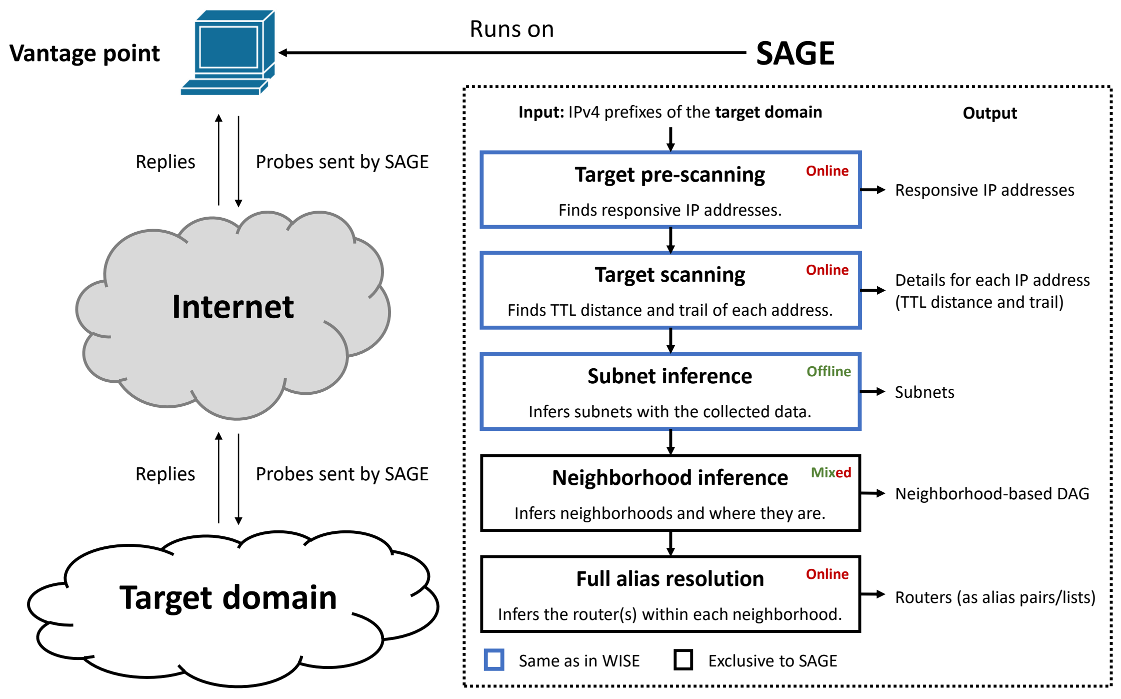

Vantage point
Runs on
SAGE
Replies
Probes sent by SAGE
Internet
Replies
Probes sent by SAGE
Target domain
Input: IPv4 prefixes of the target domain
Output
Target pre-scanning
Online
Finds responsive IP addresses.
Responsive IP addresses
Target scanning
Online
Finds TTL distance and trail of each address.
Details for each IP address (TTL distance and trail)
Subnet inference
Offline
Infers subnets with the collected data.
Subnets
Neighborhood inference
Mixed
Infers neighborhoods and where they are.
Neighborhood-based DAG
Full alias resolution
Online
Infers the router(s) within each neighborhood.
Routers (as alias pairs/lists)
Same as in WISE
Exclusive to SAGE

Figure 11.5: The complete workflow of SAGE (**S**ubnet **AG**gr**E**gation).

Figure 11.5 provides the complete workflow of SAGE. As emphasized by this figure, SAGE reuses most of the WISE methodology (cf. blue rectangles) and benefits from the linear time complexity of its algorithms (cf. Sec. A.2). As a consequence, it can easily be deployed from a single vantage point to capture a snapshot (as defined in Sec. 6.2.1) of a given target network, notably because its own algorithmical additions can be easily implemented in (quasi-)linear time (cf. Sec. 11.3). This difference already sets SAGE apart from previous topology discovery tools, such as *Rocketfuel* [113] and MERLIN [83], both tools involving hundrends or at least dozens of vantage points. Just like WISE, SAGE is an open source tool written in C/C++ whose code can be browsed online at the dedicated GitHub repository [1]. Because WISE itself has not been implemented for IPv6, SAGE is available for IPv4 only as well. Nevertheless, SAGE is as easy to deploy as WISE, and only requires some target IPv4 prefixes to start running. Example 11.1 shows the command line that could be used to run the public implementation of SAGE to probe the ULiège network [2], with the -l option providing a prefix for the various output files.

Listing 11.1: Running SAGE to measure the ULiège network

```
$ sudo ./sage 139.165.0.0/16,193.190.228./24,193.190.252.0/24 −l ULiege_net
```

Before diving into its inner workings in Sec. 11.3, it should be noted that the neighborhood inference step has its own workflow. This step first starts with a preliminary traceroute probing that aims to collect (partial) traceroute paths towards each subnet so that SAGE already has all the data it requires to infer both neighborhoods and their peers. This additional probing starts with a census of the anomaly-free trails of all subnets, i.e. peer addresses (as defined in Sec. 10.3.3.1), so that the traceroute probing for a given target IP address can be minimized by stopping at peer addresses (the details of such an operation will be provided in Sec. 11.3.2).

Once both subnets and the additional traceroute data are available, the actual neighborhood inference can start. Its workflow, summarized in Figure 11.6, consists of the five sub-steps detailed below. In the figure, each sub-step is annotated with the challenges it must tackle in order to guarantee the final neighborhood-based DAG is a true snapshot of the target network.

1. ***Subnet aggregation:*** the subnets previously discovered via the WISE methodology are aggregated into neighborhoods, based on their respective trails (with additional operations for subnets with flickering/echoing trails, cf. Sec. 10.3.2).

2. ***Peer discovery:*** using the traceroute data collected preliminarily, peers are listed for each previously discovered neighborhood. Miscellaneous hops (cf. Sec. 11.1.2) are also listed.

3. ***Peer aggregation:*** the previously listed peers are aggregated to form the largest hypothetical alias lists to detect convergence points through alias resolution. Blindspots (cf. Sec. 11.1.2) are also detected in the process and used to reduce the total number of remote peers.

4. ***Vertex creation:*** using the knowledge on convergence points (if any), the final vertices of the neighborhood-based DAG are created.

5. ***Edge creation:*** the edges between juxtaposed neighborhoods are created and mapped with a previously discovered subnet when it is possible. Remoke links are inserted in the graph to account for remote peers and enumerate the (observed) intermediate paths.

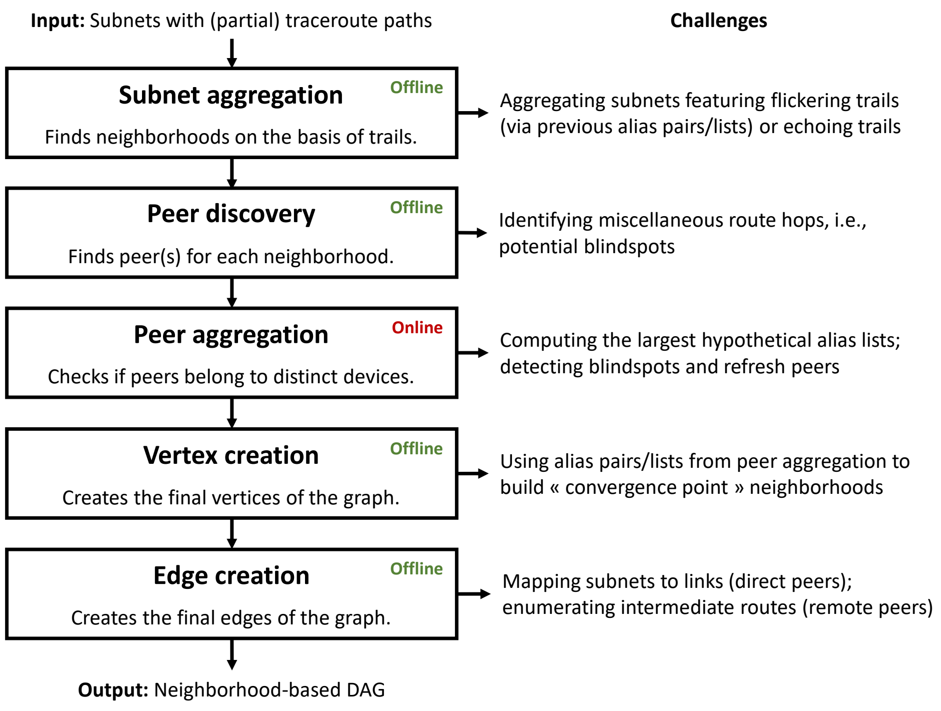

**Input:** Subnets with (partial) traceroute paths
Challenges
Subnet aggregation
Offline
Finds neighborhoods on the basis of trails.
Aggregating subnets featuring flickering trails (via previous alias pairs/lists) or echoing trails
Peer discovery
Offline
Finds peer(s) for each neighborhood.
Identifying miscellaneous route hops, i.e., potential blindspots
Peer aggregation
Online
Checks if peers belong to distinct devices.
Computing the largest hypothetical alias lists; detecting blindspots and refresh peers
Vertex creation
Offline
Creates the final vertices of the graph.
Using alias pairs/lists from peer aggregation to build « convergence point » neighborhoods
Edge creation
Offline
Creates the final edges of the graph.
Mapping subnets to links (direct peers); enumerating intermediate routes (remote peers)
**Output:** Neighborhood-based DAG

Figure 11.6: The workflow of the neighborhood inference step of SAGE.

Depending on the implementation, each sub-step of the neighborhood inference can be achieved in linear or quasi-linear time in respect of the inputs. The public implementation of SAGE opts for the latter option for convenience (i.e., it takes advantage of standard libraries of C++), but using specific data structures can, in theory, result in a linear time complexity for most of the algorithmic steps.

## 11.3 SAGE neighborhood inference

This section completely reviews the neighborhood inference step of SAGE [54] as well as some of its additional operations. Sec. 11.3.1 first reviews some implementation details of WISE (cf. Chapter 7) that have been reused in SAGE. Sec. 11.3.2 subsequently reviews the preliminary partial `traceroute` probing required to discover neighborhood adjacencies with the concept of peer (cf. Sec. 10.3.2). Subsequent sections proceed with the review of each sub-step of the neighborhood inference: subnet aggregation (Sec. 11.3.3), peer discovery (Sec. 11.3.4), peer aggregation (Sec. 11.3.5; with practical results), vertex creation (Sec. 11.3.6) and edge creation (Sec. 11.3.7). The time complexity of each unique algorithm will also be briefly discussed, and interested readers can find the detailed time complexity analyses in Appendix A (cf. Sec. A.3).

[1]Cf. `https://github.com/JefGrailet/SAGE`
[2]This will only produce satisfying results if run from the *ULiège* WiFi network, as mentioned in Sec. 4.2.1.

### 11.3.1 Implementation requirements

As it is built upon WISE [49], SAGE not only reuses the same methodology for subnet inference but also parts of its implementation. In particular, SAGE reuses all three dedicated data structures first discussed in Sec. 7.2: the IP dictionnary, the alias set and the IP aggregator. The IP dictionnary is used, as always, to register new details concerning the IP addresses involved in the various algorithms, while the alias set and the IP aggregator are required to handle the discovery of convergence points and the subsequent merging of several neighborhoods associated with them (cf. Sec. 11.1.1).

Naturally, SAGE also relies on the data previously collected with the WISE methodology: the IP dictionnary SAGE works, with starting from the neighborhood inference, is the final IP dictionnary obtained by WISE at the end of subnet inference, while the alias set that registers the alias pairs/lists discovered through the analysis of flickering trails (cf. Sec. 7.3.3 and Sec. 7.4.1) should be kept "as is". In fact, the public implementation of SAGE manages multiple alias sets, each set being associated with a different algorithmic step: the alias set computed by discovering convergence points is in practice distinct from the set computed for subnet inference. Doing so allows this public implementation to clearly distinguish the alias pairs/lists used during subnet inference from those used during neighborhood inference, but also to avoid re-resolving some alias candidates during the latter.

Indeed, SAGE implements an ***alias transitivity heuristic*** to make the most of the alias pairs/lists discovered during target scanning (cf. Sec. 7.3.2). When listing the peers of each neighborhood (cf. Sec. 11.3.4), SAGE verifies if any peer addresses match an alias pair/list found during target scanning. If this is the case, any occurrence of each IP address found in the pair/list is removed for the list of peers and the first IP address of the pair/list is inserted instead. The underlying idea is to exploit once more the property of alias transitivity (cf. Sec. 2.2): if an alias list $\{A, B, C\}$ was discovered at the end of target scanning and if a list of peers for a neighborhood consists of $\{A, B, C, D, E, F\}$, then the list of peers can be simplified into $\{A, D, E, F\}$, because if $D$ (for instance) is later found to be an alias of $A$ while discovering convergence points (cf. Sec. 11.3.5), then it is also, by transitivity, an alias of both $B$ and $C$. $A$ can later be replaced again with $\{A, B, C\}$ (i.e., after discovering convergence points) so the final alias pairs/lists can remain exhaustive. Such an heuristic slightly reduces the overhead probing (though there are usually a few flickering trails, cf. Sec. A.2.4) but also avoids an alias pair/list from being later invalidated as a consequence of a temporary network failure. Whether this heuristic should be used or not is up to the implementation, but is nevertheless recommended.

### 11.3.2 Preliminary partial `traceroute` probing

While SAGE normally has enough data to infer neighborhoods by the end of the WISE subnet inference, it does not already have the data it requires to exhaustively detect their respective peers. In other words, while it would be capable of inferring neighborhoods, SAGE may not be able to find out how exactly they are located in respect of each other, which is essential to build a neighborhood-based DAG. This is why the neighborhood inference starts with a preliminary additional `traceroute` probing. This probing consists of running new backward Paris `traceroute` measurements towards a subset of pivot interfaces of each subnet so peers can be discovered just after discovering neighborhoods. In order to minimize the number of additional probes, `SAGE` also manages to collect ***partial*** **`traceroute`** ***paths***, i.e., paths that only contain the data required to eventually discover how neighborhoods are located in respect of each other.

Before running the `traceroute` probing, `SAGE` makes some preparations. First, it makes a census of all the anomaly-free trails associated with the pivot interfaces of all discovered subnets and records them as such in the IP dictionnary (cf. Sec. 7.2.1). Indeed, such trails are the basis for building the neighborhoods that can later act as peers, since a peer can only be an anomaly-free trail (cf. Sec. 10.3.2), and is why they can also be called ***peer addresses*** (cf. Sec. 10.3.3.1). Flagging these addresses therefore allows `SAGE` to prepare peer discovery without first aggregating subnets into neighborhoods. The next step consists of building a list of target IP addresses for the new `traceroute` probing. This consists of listing a certain number of pivot interfaces per subnet, bounded by a fixed number [3]. Listing several pivot interfaces per subnet allows `SAGE` to find a more exhaustive list of peers and/or miscellaneous hops with large subnets, which is important to achieve exhaustive discovery of convergence points, and also to detect potential blindspots (cf. Sec. 11.1.2).

The final list of targets can be subsequently divided between multiple threads, exactly as with the target scanning step of the `WISE` methodology (cf. Sec. 7.3.2). Regardless of multithreading, each of the listed pivot interfaces has to be re-probed with backward Paris `traceroute` (much like during target scanning, cf. Sec. 7.3.2), using the TTL distance for the trail associated with the pivot interface minus one as the initial TTL value. For example, if the TTL distance for a given pivot interface is equal to 15, then its (anomaly-free) trail is normally equal to 14 and the TTL value of the first probe will be equal to 13. If the trail contains anomalies, its TTL distance is evaluated as the TTL distance for the associated non-anonymous, non-cycling router interface. Backward `traceroute` probing is subsequently performed until a reply comes from a router interface which is recorded in the IP dictionnary as a peer address or until the TTL value of the next probe reaches zero (i.e., this amounts to collecting a full `traceroute` path). While ensuring a route hop matches a peer address, `SAGE` must also prevent a minor issue called ***self-peering***. If the route hop matches a peer address, it should be unrelated to the subnet encompassing the target pivot (i.e., it is neither encompassed by the subnet nor a trail for it) and should not be an alias of the trail associated to the target pivot. Without these additional verifications, there is a minor risk of eventually discovering a neighborhood that is a peer for itself, which can make subsequent operations ambiguous.

If a route hop discovered through the additional backward Paris `traceroute` probing turns out to match a peer address and shows no risk of induced self-peering, the probing stops and the partial route is recorded. Ideally, when the peer address is found one hop sooner than the trail for the target pivot, `SAGE` only sends one additional probe and already gets the data it needs to eventually discover peers. This is why this additional `traceroute` probing results in partial records most of the time (cf. the quantification of direct peers of Sec. 10.3.3.3), as full `traceroute` measurements are performed

[3] Like `WISE`, `SAGE` uses up to 5 pivot interfaces by default. This can be tuned by the user upon running the tool.

if and only if SAGE cannot find a valid peer address among the replies. Being unable to find a peer usually means the target subnet is one of the subnets closest to the vantage point or belongs to a different component of the target network. It is up to the implementation to decide whether full routes should be recorded too, as they can constitute additional useful data for the user.

### 11.3.3 Subnet aggregation

Subnet aggregation, i.e. aggregating subnets into neighborhoods, can be simply achieved by grouping together subnets that feature the same trail(s). More precisely, if the trail of a subnet is not a flickering or an echoing trail, it will be grouped with all subnets that share the exact same trail. If the trail of a subnet is, on the contrary, flickering, it will be grouped with other subnets based on the alias list encompassing their respective trail(s).

Subnets featuring echoing trails, therefore built with the third rule of inference (cf. Sec. 7.4.1), cannot be aggregated into neighborhoods by the same means. However, a *best effort* strategy can be used: subnets with echoing trails can be aggregated on the basis of their respective ***pre-echoing hop(s)*** (cf. Sec. 10.3.2), i.e., the hops observed before their echoing trails. These pre-echoing hops can also be combined with the TTL distance of the pivot interfaces to make sure the resulting best effort neighborhoods are not too large: indeed, subnets featuring echoing trails should have very regular TTL distances by design (cf. the third rule of WISE subnet inference, Sec. 7.4.1). While the best effort neighborhoods built this way might be oversized in respect of the actual network, previous research has showed that this strategy can provide a very satisfying approximation [53] (cf. Sec. 10.3.3.2).

Regardless of which variables are used to aggregate subnets, subsequently denoted ***aggregation properties***, the practical algorithm for building the neighborhoods is the same: given a list of subnets, a map of lists, where each unique property is mapped to a list of subnets, can be built. The aggregation property of a subnet can then be used to add it to the list associated with this property or to create a new list if this property was not in the map before. After inserting all subnets into the map, the lists stored in the map can be enumerated and individually turned into neighborhoods.

Algorithm 14 shows the generic pseudo-code for subnet aggregation, starting with a given list of subnets $S$ whose aggregation properties have been prepared (e.g., if a subnet features flickering trails, its aggregation property is the corresponding alias pair/list). After creating the map (line 2), the algorithm iterates over the subnets, inserting each one on a list stored in the map and identified by an aggregation property (lines 3–11). If a property is not yet known, a new list will be created (lines 9–11). Once all subnets have been inserted into the map, each list in the map is turned into a neighborhood. Finally, all the created neighborhoods are stored on another list returned by the function (lines 12–17).

An alternative way to build neighborhoods bordered by subnets featuring several trails (because of flickering) can be by applying Algorithm 14 to all subnets without echoing trails, then processing the final lists more carefully (i.e., those used at line 16 to instantiate the neighborhoods). When the aggregation property associated to a list (line 14) corresponds to a flickering trail, its encompassing

**Algorithm 14** Subnet aggregation (first sub-step of neighborhood inference)

```
Require: $S$, a list of subnets
1: function AGGREGATE(List<Subnet> $S$)
2:     listsByProperty ← new Map<Property, List<Subnet> >()
3:     for subnet $\in S$ do
4:         property ← subnet.getAggregationProperty()
5:         propertyList ← listsByProperty.find(property)
6:         if propertyList $\neq \emptyset$ then
7:             propertyList.push_back(subnet)
8:         else
9:             newList ← new List<Subnet>()
10:            newList.push_back(subnet)
11:            listsByProperty.insert(property, newList)
12:    neighborhoods ← new List<Neighborhood>()
13:    for element $\in$ listsByProperty do
14:        label ← element.getNeedle()
15:        subnets ← element.getContent()
16:        neighborhoods.push_back(new Neighborhood(label, subnets))
17:    return neighborhoods
```

alias list can be looked up in the alias set produced by the WISE methodology (cf. Sec. 11.3.1), and therefore, all lists corresponding to IP addresses found in the alias list can be extracted then merged before creating the final neighborhood (line 16). This is what the public implementation of SAGE does in practice, with the implementation simply repeating Algorithm 14 to subsequently aggregate the remaining subnets, i.e., with echoing trails. Note, also, how the aggregation property in the pseudocode is used as a label for the neighborhoods (line 14): this means a neighborhood is identified by an alias list if its subnets have flickering trails, by the combination of pre-echoing hops and TTL distance if they exhibit echoing trails, or by a trail if subnets do not exhibit flickering or echoing trails. As such, the resulting neighborhoods can be individually identified exactly as described in Sec. 10.3.2.

If the look-up in the map (created at line 2) is in constant time, subnet aggregation can scale linearly with the total number of subnets to aggregate. However, a look-up in logarithmic time will still result in a decent time complexity, i.e., $\mathcal{O}(N \log N)$ if $N$ denotes the total number of subnets. A detailed analysis of the time complexity of this step can be found in Sec. A.3.1.

### 11.3.4 Peer discovery

Subnet aggregation is immediately followed by peer discovery, which consists of reviewing the previously collected partial `traceroute` records (cf. Sec. 11.3.2) towards the subnets bordering a given neighborhood to discover its peer(s). This sub-step therefore begins by enumerating all the partial routes collected towards each subnet bordering it for each neighborhood. In order to get an exhaustive list of peers while staying as close to the neighborhood as possible, a ***peer offset*** can be computed as the difference in TTL distance between the trails of the aggregated subnets and the closest potential peer(s). To do so consists of initially setting the offset to one and checking all the last hops of the partial routes. If none of the hops correspond to a recorded peer address, the offset is incremented and the search resumes with the route hops located one hop closer to the vantage point. The process is halted when a (remote) peer is found or when the peer offset goes beyond the limits of all partial routes. In the latter case, no peer offset can be actually computed and the neighborhood is considered as peerless (cf. Sec. 10.3.2). If a peer offset can be computed, then all route hops corresponding to this offset are gathered in two lists, one for peers (as addresses) and a second one for miscellaneous hops, the second list being later involved in the detection of blindspots (cf. Sec. 11.1.2) during peer aggregation (cf. Sec. 11.3.5). The peer discovery for a given neighborhood ends by sorting both lists and purging them of duplicates.

**Algorithm 15** Peer discovery (second sub-step of neighborhood inference)

```
Require: $N$, a neighborhood (as an object)
 1: procedure DISCOVERPEERS(Neighborhood $N$)
 2:     routes ← new List<Vector<RouteHop> >()
 3:     for subnet ∈ $N$.getSubnets() do
 4:         subnetIPs ← subnet.getInterfaces()
 5:         for interface ∈ subnetIPs do
 6:             if interface.hasRoute() then
 7:                 routes.push_back(interface.getRoute())
 8:     if routes.size() == 0 then
 9:         return
10:     offset ← 0
11:     foundPeers ← $false$
12:     for !foundPeers do
13:         offset++
14:         for route ∈ routes do
15:             routeLength ← route.size()
16:             if routeLength ≥ offset then
17:                 if route[routeLength − offset].isPeer() then
18:                     foundPeers ← $true$
19:                     break
20:     if !foundPeers then
21:         return
22:     $N$.setPeerOffset(offset)
23:     for route ∈ routes do
24:         hopIndex ← route.size() − offset
25:         if route[hopIndex].isPeer() then
26:             $N$.addPeer(route[hopIndex])
27:         else
28:             $N$.addMiscellaneousHop(route[hopIndex])
```

Algorithm 15 summarizes, in pseudo-code, the peer discovery sub-step for a single neighborhood $N$. It is worth noting that this pseudo-code assumes that the (partial) routes collected at the start of neighborhood inference (cf. Sec. 11.3.2) can be accessed via the subnet interfaces which were targeted to obtain them. This notably allows all such routes to be enumerated by reviewing the interfaces of each subnet bordering $N$ without going back and forth in a side data structure (lines 2–7). Once the routes have been listed, the peer offset can be computed by using a double loop to review all listed routes at offset 1, 2, 3... until a peer is found or until the offset is too large for all routes (lines 10–19). If peer(s) are found at a given offset, all peers and other miscellaneous hops appearing at this offset will be enumerated and memorized along with the offset itself in the neighborhood object (lines 22–28). It is worth noting that this pseudo-code does not explicitely show the instructions used to sort and clean the lists of peers/miscellaneous hops, as this task is implicitely carried out by the *addPeer()* and *addMiscellaneousHop()* methods shown respectively at lines 26 and 28. Whenever the peer discovery can rule out the possibility that $N$ has peers, it stops (lines 8 and 20).

As long as the preliminary `traceroute` data has been collected beforehand (cf. Sec. 11.3.2) and can be consulted in an object-oriented manner, just like in Algorithm 15, peer discovery for a single neighborhood will scale linearly with the total number of subnets bordering it, and peer discovery applied to all neighborhoods will also scale linearly with the total number of discovered subnets. The detailed time complexity analysis for peer discovery can be found in Sec. A.3.2.

### 11.3.5 Peer aggregation

Peer discovery is followed by the most important sub-step of neighborhood inference: peer aggregation. As already explained in Sec. 11.1.1, there is no guarantee that several peers detected for one neighborhood belongs to distinct devices (and therefore, distinct neighborhoods) or not. It is indeed possible that a large neighborhood acts in the topology as a convergence point of a load balancing architecture. Because of this, a convergence point can first appear as several neighborhoods during subnet aggregation (cf. Sec. 11.3.3). Therefore, before actually building the neighborhood-based DAG, SAGE must solve this problem by building hypothetical alias lists that could correspond to convergence points and resolve them (as already suggested in Sec. 11.1.1). The discovered alias pairs/lists will help SAGE to re-construct large neighborhoods during the next sub-step, a.k.a. vertex creation (cf. Sec. 11.3.6), in order to provide a more accurate picture of the topology.

Peer aggregation is best implemented by using an IP aggregator (cf. Sec. 7.2.3), though, in practice, its behaviour can be simulated with a carefully updated map. For each neighborhood, the list of peers and the list of miscellaneous hops found at the same peer offset are copied, merged together and inserted into the aggregator. After inserting the merged lists of all neighborhoods, the aggregator naturally contains the largest hypothetical alias lists, i.e., the largest potential convergence points. These alias lists can then be resolved with the help of the fingerprint-based framework introduced in Chapter 3, as the largest hypothetical alias list usually provides a relatively small total number of alias candidates, which means Internet-scale methods such as MIDAR [73] (cf. Sec. 2.3.3) would be unnecessary to resolve it. An analysis of empirical data captured with SAGE can be found at Sec. A.3.4 and demonstrates in a practical manner that hypothetical alias lists are indeed small enough for the framework elaborated in Chapter 3.

Regardless of how alias resolution is performed, the peer aggregation should result in a new alias set (cf. Sec. 7.2.2), as using each alias list (if any) to refine neighborhoods constitutes the task of the next sub-step (vertex creation, cf. Sec. 11.3.6). However, before this sub-step, the alias set can be used to detect blindspots (cf. Sec. 11.1.2) to eventually discover new (and closer) peers for some neighborhoods: it is possible to review the miscellaneous hops listed for each neighborhood (if any) to see if any have been aliased to at least one peer address during peer aggregation. If this hypothesis turns out to be correct, then each miscellaneous hop can be considered to be a blindspot. Once blindspots have been detected, a simplified peer discovery (cf. Sec. 11.3.4) can be run on neighborhoods with remote peers to see if any blindspot appears sooner in the partial routes.

Peer aggregation is also the step where SAGE can use the alias transitivity heuristic proposed in Sec. 11.3.1 to benefit from the first alias set built by the WISE methodology while reviewing flickering trails. The public implementation of SAGE uses this heuristic between peer discovery and peer aggregation: if a list of peers includes at least one address matching an an alias list first discovered at the end of target scanning (cf. Sec. 7.3.3), this list is cleaned of any IP address belonging to the alias list and replaced with the first IP address of the same list. Peer aggregation can then be carried out normally, and the alias pairs/lists with flickering trails can later be used to complete (by alias transivity) the alias pairs/lists discovered during peer aggregation.

**Algorithm 16** Peer aggregation (third sub-step of neighborhood inference)

```
Require: N, list of discovered neighborhoods (as objects)
Require: A, an alias resolution utility (object)
 1: function PEERAGGREGATION(List<Neighborhood> N, AliasResolver A)
 2:     aggregator ← new IPAggregator()
 3:     for neighborhood ∈ N do
 4:         peers ← neighborhood.getPeers()
 5:         miscellaneousIPs ← neighborhood.getMiscellaneousIPs()
 6:         hypotheticAlias ← MERGELISTS(peers, miscellaneousIPs)
 7:         aggregator.update(hypotheticAlias)
 8:     hypotheses ← aggregator.getClusters()
 9:     newSet ← new AliasSet()
10:     for hypothesis ∈ hypotheses do
11:         newAliasLists ← A.resolve(hypothesis)
12:         for newAlias ∈ newAliasLists do
13:             newSet.addAlias(newAlias)
14:     return newSet
```

Algorithm 16 shows the pseudo-code of peer aggregation, which starts with *N*, the list of neighborhoods discovered during subnet aggregation (cf. Sec. 11.3.3) annotated with peer data inferred during peer discovery (cf. Sec. 11.3.4). In addition to *N*, the pseudo-code also involves *A*, an alias resolution utility provided as an object. The reason why the pseudo-code includes this parameter is because such an utility accounts not only for alias resolution in general but also for additional operations, such as interacting with the IP dictionary to map a raw IP address with an entry and to get the side data that could be relevant to the resolution process (if necessary). In practice, the public implementation of SAGE uses similar utilities during peer aggregation. It also uses the alias set data structure recommended in Sec. 7.2.2 (line 9), in this case to return the alias pairs/lists of peer aggregation whilst, at the same time, ensuring they are isolated from the alias pairs/lists discovered by the WISE methodology (as suggested in Sec. 11.3.1). The process of aggregating IP addresses into the largest hypothetical alias lists is also abstracted by an IP aggregator structure (cf. Sec. 7.2.3; line 2).

All in all, if all recommended structures are used, peer aggregation becomes straightforward: the IP aggregator is created then updated with the peer data of each neighborhood (lines 2–7), after which the largest hypothetical alias lists (called *hypotheses* in the pseudo-code) are collected and resolved via the alias resolution utility $A$ (lines 8–13). The resulting alias pairs/lists can be returned immediatly as a new alias set (line 14). However, it should be noted that the alias set should only contain alias lists with more than one IP address, i.e., IP addresses that were not aliased to others should be omitted. This is implicit in Algorithm 16, since the goal is to discover which peers have aliases (unaliased peers will not change anything during the subsequent steps).

Peer aggregation should scale with the total number of neighborhoods in a quasi-linear manner, i.e., a careful implementation should prevent a quadratic time complexity. Depending on how the recommended IP aggregator (cf. Sec. 7.2.3) is implemented, the overall time complexity should be either $\mathcal{O}(N \log N)$ (look-up in the aggregator is in constant time) or $\mathcal{O}(N \log^2 N)$ (same operation is in logarithmic time) if $N$ expresses the total number of neighborhoods. As for the alias resolution itself, it should scale linearly with the total number of alias candidates if the implementation relies on the framework elaborated in Chapter 3. A detailed commentary on the time complexity of peer aggregation can be found in Sec. A.3.3.

### 11.3.6 Vertex creation

With the knowledge of the peers and their respective aliases, SAGE can finally start building the neighborhood-based DAG with vertex creation, i.e., creating the final vertices of the DAG. This substep starts by compiling the peer data in a new map that will be subsequently denoted as the ***peers map***. In this map (or *haystack*), the keys (or *needles*) are individual IP addresses while the *values* are the final peers, i.e., either the alias lists discovered during peer aggregation (cf. Sec. 11.3.5) or individual IP addresses (i.e., unaliased peer addresses). Building and completing this map allows SAGE to easily list the ***termini neighborhoods*** (or *termini vertices*), i.e., neighborhoods that do not act as peers to other neighborhoods themselves and constitute the ends of the DAG. Naturally, *termini* vertices always include all best effort neighborhoods, since the latter cannot be peers by design (cf. Sec. 10.3.2). Listing such neighborhoods simplifies the creation of vertices: indeed, termini neighborhoods can be directly translated as vertices and can be omitted from the subsequent processing of the peer data (e.g., merging chunked neighborhoods as a consequence of peer aggregation).

With the peers map at hand, the existing neighborhoods can be reviewed to prepare the final data for creating the vertices. Each neighborhood that is identified by one or several subnet trails (several trails corresponding to flickering trails aliased together) is categorized as a *terminus* vertex if and only if there is no value in the peers map for any of its trails. Otherwise, the neighborhood can be stored in another structure mapping each subnet trail with the corresponding neighborhood. This additional temporary structure helps to list the neighborhoods to merge upon creating the final vertex corresponding to a convergence point (detected during peer aggregation). While identifying termini neighborhoods and organizing the data to merge neighborhoods more easily, the neighborhoods themselves can be updated with the final peer data, since peers are now either individual IP addresses or alias lists. For convenience, they can be managed as objects (initially, the peer data stored in neighborhoods only consists of IP addresses). Alternatively, storing the final peer data for each neighborhood can be done with another side map, but if an implementation of SAGE uses the object-oriented paradigm (like the public implementation [48]), the former is more elegant.

**Algorithm 17** Preparing the data for vertex creation (fourth sub-step of neighborhood inference)

```
Require: $N$, the list of discovered neighborhoods (as objects)
Require: $A$, the alias set built by peer aggregation
 1: function PREPARE(List<Neighborhood> $N$, AliasSet $A$)
 2:     peersMap ← new Map<IP, Peer>()
 3:     for n ∈ $N$ do
 4:         peerIPs ← n.getPeerIPs()
 5:         for IP ∈ peerIPs do
 6:             if peersMap.find(IP) $\neq \emptyset$ then
 7:                 continue
 8:             aliasList ← $A$.findAliasList(IP)
 9:             if aliasList $\neq \emptyset$ then
10:                 newPeer ← new Peer(aliasList)
11:                 for alias ∈ aliasList do
12:                     peersMap.insert(alias, newPeer)
13:             else
14:                 peersMap.insert(IP, new Peer(IP))
15:     termini ← new List<Neighborhood>()
16:     neighborsMap ← new Map<IP, Neighborhood>()
17:     for n ∈ $N$ do
18:         n.listFinalPeers(peersMap)
19:         labels ← n.getIdentifiyingIPs()
20:         isPeer ← $false$
21:         for IP ∈ labels do
22:             if peersMap.find(IP) $\neq \emptyset$ then
23:                 isPeer ← $true$
24:                 neighborsMap.insert(IP, n)
25:         if !isPeer then
26:             termini.push_back(n)
27:     return termini, neighborsMap
```

Algorithm 17 shows the pseudo-code to prepare the data for vertex creation, starting with the list of neighborhoods $N$ and the alias set $A$ produced by peer aggregation. This pseudo-code both creates the peers map and uses it to spot the termini neighborhoods while preparing the data. The complete peers map can be easily built by browsing the neighborhoods with a loop (lines 2–14). For each neighborhood, the peers (as individual IP addresses) are reviewed (starting from line 4). If the peer address already exists as a value in the map, the current iteration can be skipped (line 6). Otherwise, the code determines whether an alias pair/list for this peer address exists in $A$. If it does, the corresponding *Peer* object is created and inserted in the map for any IP address that is part of its alias list (lines 8–12). If no alias pair/list can be found, then a *Peer* object encapsulating a single IP address can be created and inserted in the map instead (line 14). After this first loop, the peers map can be used to isolate the termini via a second loop (lines 15–26). If a neighborhood is found to be a peer, it is also stored in the side map *neighborsMap*, which will be used later to look up neighborhoods upon creating vertices that correspond to convergence point (line 24).

Once the data is fully prepared, vertex creation becomes a purely technical operation. The termini neighborhoods can be directly translated into vertices while the remaining vertices, i.e. the peers, will be created either with a single neighborhood (if the associated peer address was not aliased to any other) or with a list of neighborhoods (if the associated peer addresses were previously aliased), preliminarily gathered in a side map (*neighborsMap*) if following strictly the operations of Algorithm 17. The peer data stored by each neighborhood can be transmitted to the final vertices as well, as a starting point for the final sub-step of neighborhood inference: edge creation (cf. Sec. 11.3.7).

Vertex creation will scale with the total number of neighborhoods in the same manner as peer aggregation (cf. Sec. 11.3.5 and Sec. A.3.3). In other words, it should feature a quasi-linear time complexity, i.e. $\mathcal{O}(N \log N)$ or $\mathcal{O}(N \log^2 N)$ depending on the implementation of the recommended data structures. Sec. A.3.5 provides the detailed time complexity analysis of vertex creation.

### 11.3.7 Edge creation

As already explained in Sec. 11.1.3 and Sec. 11.2, edge creation does not just consist of creating the edges between the final vertices of the graph, as the adjacencies between neighborhoods are already accounted for by the peer data. The purpose of this sub-step is actually to find out what the real-life link might consist of when this link binds together two juxtaposed neighborhoods (i.e., separated by a TTL distance of one hop) and to enumerate all possible intermediate routes between two neighborhoods when one is a remote peer of the second (as defined in Sec. 10.3.2).

To start this sub-step first requires a list of the final vertices (created by the previous sub-step, cf. Sec. 11.3.6), each vertex having recorded its peers as their vertex equivalent. I.e., rather than just knowing the peer addresses or the alias lists corresponding to final peers (when aliased), a neighborhood translated as a vertex should already have a pointer/reference to each vertex corresponding to a (final) peer. Doing this mapping is recommended prior to edge creation to spare the effort of maintaining side data structures to determine which peer address corresponds to which vertex.

In order to further ease edge creation, it is also recommended creating a data structure for efficiently finding a subnet containing a given IP address. Such a structure, which will be subsequently denoted as ***subnet dictionnary***, can considerably help to find a subnet upon creating an indirect link (as defined in Sec. 11.1.3), which is considered as soon as the possibility to create a direct link is ruled out. While there are several ways to approach this problem, a time/memory trade-off (very similar to what was recommeded to implement the IP dictionnary, cf. Sec. 7.2.1) can be relied on: the X first bits of the provided IP address are used to compute an index for a large array of lists, so that the list contains a bounded number of subnets (a value for X can be 20, since subnets in WISE/SAGE cannot have a prefix length shorter than 20). Then, the problem comes down to finding the subnet in the selected list that encompasses the given IP, which will be in $\mathcal{O}(1)$ due to the bounded size. Ideally, the subnets should also be annotated with the neighborhood they are bordering prior to this operation for the sake of completeness.

If all previous recommendations are strictly adhered to, edge creation can be solved as follows. First, the tool can instantiate an object or structure modeling the neighborhood-based DAG as a whole, such an object/a structure including the previously described subnet dictionnary. Then, the initial list of vertices is processed in order to add all the subnets bordering each neighborhood in the subnet dictionary. At the same time, the ***gates*** of the DAG are identified and memorized in the graph object, *gate* simply being a shorter name for a peerless neighborhood (cf. Sec. 10.3.2). The name simply refers to how peerless neighborhoods are typically the neighborhoods closest to the vantage point. Vertices that do not fall under this definition are kept in order to create their incident edges with the help of a last loop. For each vertex $v$, its peers (as vertices) are listed and reviewed. If such peers are remote, then remote links (cf. Sec. 11.1.3) are inserted to connect $v$ to each of its peers, with a side operation enumerating the routes that exist between them. If the peers are direct, it must first be checked that $v$ is not a best effort neighborhood (cf. Sec. 10.3.2), otherwise an indirect link without a mapping is inserted. Indeed, because best effort neighborhoods are not, by design, identified by anomaly-free trails, it is not possible to map a subnet to their incident edges. Then, for each peer $u$, each subnet $S_u$ bordering $u$ must be checked to see if it contains one of the subnet trails $t_v$ used to identify $v$. If such a subnet exists, then a direct link is created between $u$ and $v$ and mapped to $S_u$. Otherwise, only an indirect link can be created between $u$ and $v$. A subnet $S_w$ (bordering a vertex $w$) encompassing any subnet trail $t_v$ of $v$ should then be looked for in the subnet dictionary. If no such subnet can be found, the indirect link is inserted without a mapping, otherwise it is mapped to $S_w$.

Algorithm 18 shows the pseudo-code for edge creation, starting with a list of vertices $V$ which are assumed to have updated peer data (i.e., their respective peers are listed in their vertex form). This algorithm starts by identifying the gates of the graph and preparing the subnet dictionary (lines 2–9). All vertices but the gates are then processed to be connected with their respective peers with either a direct, either an indirect or a remote link (lines 10–33). The loop first starts by collecting the peer data (lines 11–12) then verifies how it can connect $u$ (the current peer) with $v$ (the current vertex for which incoming edges are created). If $u$ is a remote peer of $v$ (tested at line 14), then the algorithm should create a remote link between $u$ and $v$ and enumerate the intermediate routes between them to complement the data (lines 32–33). The process of enumerating the routes, while not shown here, simply consists of browsing once more the `traceroute` data collected for the subnets bordering $v$ to find the routes containing occurrences of the anomaly-free trail(s) identifying $u$ (denoted as *label* in the pseudo-code) and to extract the portion of these routes separating $u$ from $v$. For direct peers, the code first checks whether $v$ is a best effort neighborhood and creates an indirect link without a mapping if this is the case (lines 15–17). If $v$ is not a best effort neighborhood, the code looks for

**Algorithm 18** Edge creation (final sub-step of neighborhood inference)

```
Require: $V$, the list of the final vertices (as objects)
 1: function BUILDDAG(List<Vertice> $V$)
 2:     dag ← new DirectedAcyclicGraph()
 3:     toConnect ← new List<Vertice>()
 4:     for v ∈ $V$ do
 5:         dag.addToSubnetDict(v.getSubnets())
 6:         if v.hasPeers() then
 7:             toConnect.push_back(v)
 8:         else
 9:             dag.addGate(v)
10:     for v ∈ toConnect do
11:         peerOffset ← v.getPeerOffset()
12:         peers ← v.getPeers()
13:         for u ∈ peers do
14:             if peerOffset == 1 then
15:                 if v.isBestEffort() then
16:                     u.addEdge(new IndirectLink(u, v))
17:                     continue
18:                 connectingSubnet ← ∅
19:                 labels ← v.getIdentifiyingIPs()
20:                 for label ∈ labels and connectingSubnet == ∅ do
21:                     connectingSubnet ← u.getSubnetContaining(label)
22:                 if connectingSubnet ≠ ∅ then
23:                     u.addEdge(new DirectLink(u, v, connectingSubnet))
24:                     continue
25:                 for label ∈ labels and connectingSubnet == ∅ do
26:                     connectingSubnet ← dag.getSubnetContaining(label)
27:                 if connectingSubnet ≠ ∅ then
28:                     u.addEdge(new IndirectLink(u, v, connectingSubnet))
29:                 else
30:                     u.addEdge(new IndirectLink(u, v))
31:             else
32:                 routes ← v.enumerateIntermediateRoutes(u.getLabels())
33:                 u.addEdge(new RemoteLink(u, v, routes))
34:     return dag
```

a subnet bordering $u$ that contains any of the trails identifying $v$ (again denoted *label*). If such a subnet can be found, a direct link binding $u$ and $v$ together can be created (lines 20–24). Otherwise, an indirect link is created, and is mapped to a subnet bordering a third neighborhood $w$ if such a subnet can be found in the subnet dictionary (lines 25–30), otherwise it is left without being mapped.

It is worth noting that Algorithm 18 does not insert anything into the DAG object (simply denoted as *dag*) while connecting the vertices together (at lines 10–33): indeed, by design, it will be possible to reach all vertices in the DAG by just following the edges starting from any gate. In fact, any traversal of a neighborhood-based DAG consists of starting the visit at a given gate then following the edges. Because cycles can occasionally exist, any traversal should keep track of the vertices that have already been visited to avoid cycling in the DAG while following the edges. In practice, the public implementation of SAGE does a very first visit of the freshly built graph in order to number the neighborhoods, so that a third party software designed to parse and to analyze/process neighborhood-based DAGs can refer to any neighborhood with a unique integer ID.

While previous operations scale with a single variable, edge creation can be considered to scale with two: the total number of neighborhoods $N$ and the total number of subnets bordering all neighborhoods $S$. If the recommended subnet dictionary is implemented, the time complexity of edge creation should be $\mathcal{O}(S^2/N)$, with the extreme case of $N = 1$ trivializing the whole operation. In other words, the time complexity in this case should always be better than $\mathcal{O}(S^2)$. The detailed time complexity analysis for edge creation is provided in Sec. A.3.6.

## 11.4 Alias resolution with the neighborhood-based DAG

In addition to providing a hop-level mapping of a target network (cf. Sec. 9.3.1), a neighborhood-based DAG as built by SAGE [54] can also be used to perform alias resolution as with TreeNET [51, 55]. The tree-like mapping of TreeNET (cf. Sec. 4.1.1) has been previously taken advantage of as a space search reduction scheme for the fingerprint-based framework presented in Chapter 3. Since a neighborhood-based DAG mainly differs from a tree-like mapping by allowing individual vertices to have multiple predecessors in the topology, it can be used in exactly the same manner to perform alias resolution: for a given neighborhood, the anomaly-free trails identifying a neighborhood (if any) can be listed along with the contra-pivot interfaces of the bordering subnets to create a list of alias candidates. For instance, in Figure 10.6 (Sec. 10.3.2), the trail identifying the neighborhood and the contra-pivot interfaces (i.e., the grey squares) form an alias resolution scenario which could correspond to a single router. Any list of candidates deduced from each neighborhood can be fed to the fingerprint-based framework of Chapter 3 to discover one or several alias pairs/lists, therefore facilitating, in practice, the detection of the (meshes of) routers that constitute the neighborhoods.

The public implementation of SAGE also provides such an alias resolution scheme, given that the framework it uses is already implemented to perform alias resolution on flickering trails and (hypothetical) convergence points. This *final alias resolution* comes as the fifth and final algorithmic step of the complete SAGE workflow (cf. Sec. 11.2). If the recommended alias set structure (cf. Sec. 7.2.2) is used, the alias pairs/lists discovered during this final alias resolution can be stored in a third set to segregate them from the pairs and lists discovered just before subnet inference (cf. Sec. 7.3.3) and during peer aggregation (cf. Sec. 11.3.5). To make the most of previously discovered alias pairs/lists, the previously described alias transitivity heuristic (cf. Sec. 11.3.1) can also be reused.

**Algorithm 19** Final alias resolution in SAGE

```
Require: N, a vertex (neighborhood) of the neighborhood-based DAG
Require: V, an array to remember already visited vertices
Require: A, an alias resolution utility
Require: R, the alias set storing the discovered aliases
 1: procedure RESOLVE(Vertex N, Bool V[], AliasResolver A, AliasSet R)
 2:     if V[N.getNumber() −1] then
 3:         return
 4:     V[N.getNumber() −1] ← $true$
 5:     aliasCandidates ← new List<IP>()
 6:     for trail ∈ N.getIdentifyingIPs() do
 7:         aliasCandidates.push_back(trail)
 8:     for subnet ∈ N.getSubnets() do
 9:         contrapivots ← subnet.getContrapivotIPs()
10:         for contrapivot ∈ contrapivots do
11:             aliasCandidates.push_back(contrapivot)
12:     newAliasLists ← A.resolve(aliasCandidates)
13:     for newAlias ∈ newAliasLists do
14:         R.addAlias(newAlias)
15:     outEdges ← N.getEdges()
16:     for edge ∈ outEdges do
17:         RESOLVE(edge.getTail(), V, A, R)
```

The practical algorithm of the final alias resolution simply consists of combining a graph traversal (shortly described in Sec. 11.3.7) with alias resolution utilities. For the sake of completeness, Algorithm 19 provides a pseudo-code for this operation. The parameters of the provided procedure deserve some commentary: in addition to a current vertex $N$ (for **N**eighborhood), the procedure also requires a boolean array $V$ to remember which neighborhoods have been already visited (and resolved), an alias resolution utility $A$ and an alias set $R$ (for **R**esult). In particular, these two last parameters are used in the same manner as in Algorithm 16, with $A$ abstracting how alias resolution is actually performed as well as how it accesses side data, the only difference being that the alias set $R$ is not returned but completed progressively (and assumed to have been created before starting the graph traversal). This procedure is recursive, and starts with checking if $N$ has been already visited (line 2) and recording the visit if this is not the case (line 4). For next step, the alias candidates (lines 5–11) are listed and subsequently resolved with $A$ with the results being stored in $R$ (lines 12–14). The procedure ends by making recursive calls on the subsequent vertices in the DAG (lines 15–17). In the pseudo-code, an edge $u \rightarrow v$ has the tail $u$ and the head $v$, which is why the method *getTail()* is used to retrieve the next vertex to visit. Finally, it goes without saying that the procedure of Algorithm 19 must be started from every *gate* of the DAG to completely process it.

As long as the methodology of alias resolution scales linearly with the total number of alias candidates, much like the framework described in Chapter 3, the final alias resolution step of SAGE should scale linearly with the total number of subnets. Indeed, each neighborhood should feature a maximum number of alias candidates proportional to the total number of subnets bordering it. Sec. A.3.7 provides a detailed time complexity analysis for this very last algorithmic step.

## 11.5 Closing comments on SAGE

In this chapter, how SAGE [54] manages to systematically build a neighborhood-based DAG to map the hop-level of a target network has been covered in detail. The challenges SAGE has to tackle and the workflow of the neighborhood inference have been discussed at length. Additionally, interested readers can consult Sec. A.3 of Appendix A to learn about the time complexity of each algorithm discussed in this chapter, as well as to have some empirical data on the typical results of peer aggregation (cf. Sec. A.3.4). This data not only further demonstrates the soundness of peer aggregation, but also shows that detecting convergence points is critical to achieve accurate topology mapping.

Another major contribution of SAGE is how it shows that topology mapping is achievable with partial `traceroute` measurements: while previous research already aimed at reducing the `traceroute` overhead probing [39, 65] (see also Sec. 9.1), SAGE not only reduces `traceroute` probing but can also virtually achieve topology mapping without collecting complete `traceroute` paths, i.e., from end to end. In practice, SAGE still collects complete paths for peerless neighborhoods to guarantee they have no peers at all (cf. Sec. 11.3.2), but heuristics could easily be designed to stop complete `traceroute` measurements sooner, based on the TTL distances that are available in the IP dictionary by the end of subnet inference (cf. Sec. 11.3.1).

However, before exploiting data captured with SAGE, the tool must be first assessed on groundtruth topologies to show that its neighborhood-based DAGs are true to the topology. Moreover, SAGE should also be evaluated on the basis of snapshots captured in the wild, notably in order to demonstrate that SAGE can capture comparable snapshots of one target network regardless of the vantage point. Both these forms of assessment are covered in Chapter 12.

CHAPTER 12

# SAGE EVALUATION

This chapter assesses SAGE [54], the topology discovery tool previously described in Chapter 11, by validating it on groundtruth networks and evaluating it in the wild after deploying it on the PlanetLab testbed. This therefore answers research question 3 from Sec. 9.4: as a reminder, the question was to determine whether or not SAGE could capture accurate snapshots of target networks, regardless of the vantage point. Sec. 12.1 first presents the validation of SAGE, conducted on two groundtruth networks using a unique approach to confirm the discovered links are true to the topology. Sec. 12.2 proceeds with the evaluation in the wild, notably discussing the ability of SAGE to capture comparable pictures of a same target network despite changing the vantage point. Sec. 12.3 concludes this chapter by summarizing the perspectives offered by the SAGE methodology.

## 12.1 SAGE validation

This section validates SAGE assisted by two groundtruth networks. Sec. 12.1.1 first presents the methodology used to validate the tool, which relies on a unique way of collaborating with the groundtruth partners to verify the discovered links. Sec. 12.1.2 subsequently provides the results and discusses the merits of SAGE as well as why a few links were not correctly inferred.

### 12.1.1 Validation methodology

To validate SAGE, the groundtruth networks previously introduced in Sec. 8.1 to validate WISE [52, 53] (as well as the concept of neighborhood, see Sec. 10.3.3) were measured again with the former in September 2020. It should be repeated that these groundtruth networks consist of the ULiège network (one /16 IPv4 prefix completed by two other /24 prefixes for the backbone) and the backbone of a Belgian ISP whose IPv4 prefixes cover hundreds of thousands of attributable IP addresses [1].

[1]As always, the details of each groundtruth network and the collected data will not be made public for security.

Validating a neighborhood-based DAG (as defined in Sec. 11.1) might, at first glance, seem to be a challenging task. Not only is it necessary to confirm that the overall shape of the graph is true to the topology, i.e., the successor/predecessor relationships accounted by the edges are accurate, but the subnets mapped to each edge need to be checked to ensure they are also correct. In other words, a complete verification of a neighborhood-based DAG requires audits at both the router-level and the subnet-level, and requesting exhaustive data at these levels could be considered as too intruisive by a network operator.

Hopefully, a strategy to thoroughly verify the links discovered by `SAGE` while minimizing the time spent on network audits by the groundtruth partners could be designed. The key idea is to take advantage of the output of the `show ip ro IP` command (where `IP` is a given IP address) available on most routers. Indeed, the output of this command demonstrates whether the provided IP address is located next hop and also provides the subnet prefix corresponding to the outgoing link towards which packets would be routed for this specific target address. Of course, such a command line also has to be run from the correct device. Since neighborhoods discovered by `SAGE` are usually identified by one or several router interfaces previously discovered with `traceroute` probing (cf. Sec. 11.3.3) and alias resolution (cf. Sec. 11.3.5), providing the associated IP address(es) to a groundtruth partner is sufficient for this partner to recognize a specific router and log onto it to run one or several commands. Then, for each neighborhood, a `show ip ro` command can be generated for each outgoing edge, with the IP address completing the command being associated with the next neighborhood.

For example, if the neighborhood-based DAG contains an edge `N32 → N33 via x.y.z.0/30`[2] where `N32` is identified by the IP address `a.b.c.1` and `N33` by `x.y.z.2`, then the network operator has to log onto the router bearing the interface `a.b.c.1` and run the command `show ip ro x.y.z.2`. The network operator's involvement stops here: the output of each command can be requested to subsequently verify whether or not the initial edge is correct. If the output states that the provided IP address is *directly connected*, this means that the edge discovered by `SAGE` corresponds to a real life link. If this output is parsed to extract the subnet prefix associated to the outgoing link, it then becomes possible to compare the subnet mapped to the edge in the DAG discovered by `SAGE` (if any) with the groundtruth prefix. Ideally, the inferred prefix should coincide with the groundtruth prefix, but small differences in prefix length can be considered as an acceptable result depending on how many subnet interfaces are visible to probes (as discussed in Sec. 8.1.2).

Not only does this strategy determine whether the discovered links are true to the topology, but it can also be a way of assessing the discovered convergence points. Indeed, when noticing a router identified by multiple IP addresses in `SAGE` data, the operator can also confirm whether or not the listed addresses are attributed to the same device. The present methodology is therefore capable of verifying the observed properties of a topology at both the router-level and the subnet-level while not being too intrusive, i.e., the groundtruth partners do not have to send an exhaustive description of their respective network as they only confirm observations.

[2]This text format is identical to the actual output of `SAGE` for a direct link (cf. Sec. 11.1.3).

### 12.1.2 Validation results

| | **Metrics** | **ULiège network** | **Belgian ISP backbone** |
|---|---|---|---|
| Subnets | Covered prefixes | 95.93% | 88.95% |
| | Exact inferred prefixes | 57.55% | 46.98% |
| | Differing by a most one bit | 79.94% | 74.09% |
| DAG | Neighborhoods | 51 | 121 |
| | Discovered links | 34 | 113 |
| | Direct links | 29 | 108 |
| | Indirect links | 5 | 5 |
| | Correct links | 34 (100%) | 108 (95.58%) |
| | False direct | 0 (0%) | 3 (2.78%) |
| | False indirect | 0 (0%) | 2 (40%) |
| Subnets as links | Matched | 34 | 108 |
| | Exact prefix | 33 (97.06%) | 93 (82.3%) |
| | Mean prefix difference | 1 | 1.2 |

Table 12.1: Validation of SAGE on the ULiège network and the backbone of a Belgian ISP.

Table 12.1 provides the results of the validation of SAGE on both groundtruth networks, divided into three parts: one for subnet inference, another one for the DAG and a third one for the subnet mappings. The first part of Table 12.1 serves as a quick reminder that the subnets discovered with the WISE methodology (reused by SAGE, cf. Sec. 11.2) are sound but also exhaustive in respect of both target networks. The metrics are the same as in Sec. 8.1.2. The two other parts assess the ability of SAGE to build a consistent neighborhood-based DAG.

The main highlight of Table 12.1 is that SAGE consistently finds accurate links: as shown by the second part of the table, the vast majority of the discovered links were true to the topology for both networks, with 100% of the links discovered on the ULiège network being correct and 95% of the links found on the Belgian ISP backbone being true to the topology as well. As demonstrated by the third part of Table 12.1, the subnets that were mapped to the detected links were also overwhelmingly accurate: only one subnet had a prefix greater by one bit within the ULiège network, while a dozen of subnets mapped to links featured prefixes longer by one bit within the Belgian ISP backbone (only three prefixes were longer by two bits). A careful analysis of the validation results for the Belgian ISP backbone also revealed that all five incorrect links (including two indirect links) corresponded to subnets incorrectly routed because of a BGP configuration error, resulting in incorrect trails (therefore incorrect neighborhoods). Interestingly, the subnets themselves were still matching the groundtruth (though not counted in the metrics). While not mentioned in Table 12.1, all convergence points discovered by SAGE were correct in respect of the topology in both cases. In fact, in the case of the ULiège network, a closer inspection of the data with the friendly help of the SeGI [3] rather showed that SAGE had only missed one convergence point because the collected `traceroute` data did not suggest its existence during peer discovery (cf. Sec. 11.3.4).

[3]For reminders, the SeGI (*Service Général d'Informatique*) is the company maintaining the ULiège network.

An interesting observation to be made is that both measurements included a small number of indirect links (as defined in Sec. 11.1.3), the majority of which were still ruled as true to the topology: only two such links were inaccurate in the case of the Belgian ISP backbone. As a reminder, an indirect link simply accounts for an edge $u \rightarrow v$ whose mapped subnet is not in the vicinity of $u$ (i.e., the neighborhood that is closer to the vantage point), but rather around a third neighborhood. Interestingly, in the case of the ULiège network, all subnets mapped to the indirect links bordered the two neighborhoods that should have been merged due to the (missed) convergence point: each neighborhod had a few outgoing edges mapped with a subnet from the other half (and vice versa). This observation suggested that both neighborhoods were related, and could be used to design a post-processing of the DAG. However, the correct indirect links found within the Belgian ISP backbone did not suggest a convergence point was missed, and in fact supports the intuition (mentioned in Sec. 11.1.3) that subnets can occasionally be routed (i.e., reached with direct probes) and crossed (i.e., the probes pass through them) differently. Interestingly, such links can reveal adjacencies that are not directly observed via (Paris) `traceroute` probing, especially if the mapped subnets are large enough to bind together multiple routers (this will be discussed in greater detail in Chapter 13).

Finally, it should be noted that the graphs provided slightly more neighborhoods than edges: this is due to having a few neighborhoods without peers (often because they were the closest to the vantage point), but also due to having several connected components in the case of the ULiège network. In fact, `SAGE` will rarely discover all the links within a network since it typically runs from a single vantage point. Discovering all links would require probing the same network from different vantage points and combining the results. However, whether or not the neighborhood-based DAG is representative of a target network can be checked by verifying the subnet-level data exhaustively covers the groundtruth data. It is for this reason that Table 12.1 also provides subnet metrics to demonstrate that `SAGE` (via `WISE`) accurately covered most of both networks, with the ratio of inferred prefixes differing by at most one bit being close to the ratio of covered prefixes. Such a ratio accounts for subnets which the (lack of) *live* interfaces prevents finding the exact prefix (cf. Sec. 8.1).

## 12.2 `SAGE` in the wild

This section evaluates `SAGE` in the wild, using data collected from various **A**utonomous **S**ystems (or ASes) from the PlanetLab testbed. Sec. 12.2.1 first briefly describes how `SAGE` has been deployed. Sec. 12.2.2 subsequently discusses the ability of `SAGE` to capture comparable pictures of a same network despite changing the vantage point at each new measurement.

### 12.2.1 Deployment on PlanetLab

The first complete implementation of `SAGE` [4] was deployed by the end of 2019 on the PlanetLab testbed. Before going any further, it is important to note that the termination of the PlanetLab testbed

[4]The (open source) code can be found on GitHub: `https://github.com/JefGrailet/SAGE`.

was planned for May 2020 [99], and that very few nodes were still usable by Fall 2019. In fact, only two or three dozens of PLE (**P**lanet**L**ab **E**urope) nodes, i.e. all located in Europe, could be used to deploy WISE or SAGE at the time. Hopefully, the PlanetLab testbed got a spiritual successor in the EdgeNet cluster [108], which became usable starting from Spring 2020 and with which more SAGE snapshots have been captured. How they have been performed will be discussed in Sec. 13.2.1.

Due to the state of the PlanetLab testbed as of early 2020, SAGE has been initially deployed in the same manner as WISE (cf. Sec. 8.2.1), i.e., SAGE was run each time from a single vantage point to probe a single Autonomous System and capture a snapshot of it (as defined in Sec. 6.2.1). To optimize the usage of the remaining PLE nodes, SAGE measurements were also scheduled with vantage point rotation (again, see Sec. 6.2.1), i.e., each target AS was measured once from each node from a subset of the available PLE nodes at the time. Hopefully, this strategy made it possible to study the differences between the neighborhood-based DAGs built by SAGE from each vantage point, as will be discussed in Sec. 12.2.2. Finally, Table 12.2 lists the ASes measured with SAGE in early 2020 from the PlanetLab testbed. Several target ASes already probed with WISE in 2019 were kept (cf. Table 8.2), with a few additions and extractions. These 12 ASes were measured by two campaigns: the first one beginning on December 29th, 2019 and ending on January 15th, 2020 and the second one during March 2020, more precisely from March 6th to March 20th.

| **ASN** | **Name** | **Origin** | **Category** |
|---|---|---|---|
| 12956 | Telefonica Global Solutions | Spain | Tier-1 |
| 3257 | GTT Communications Inc. | USA | Tier-1 |
| 6453 | TATA Communications | USA | Tier-1 |
| 6762 | Sparkle | Italy | Tier-1 |
| 13789 | Internap Corporation | USA | Transit |
| 286 | KPN B.V. | Netherlands | Transit |
| 50673 | Serverius Holding B.V. | Netherlands | Transit |
| 6939 | Hurricane Electric LLC | USA | Transit |
| 8928 | Interoute Communications | UK | Transit |
| 9198 | JSC Kazahtelecom | Kazakhstan | Transit |
| 1241 | Forthnet | Greece | Stub |
| 224 | UNINETT | Norway | Stub |

Table 12.2: Summary of the 12 Autonomous Systems probed with SAGE in early 2020.

### 12.2.2 Graph isomorphism

While the results of the validation of SAGE on groundtruth networks are promising (cf. Sec. 12.1), both involved groundtruth networks primarily consisting of end systems. Moreover, the measurements themselves were not affected to any great extent by traffic engineering, though multiple convergence points could be detected, notably within the Belgian ISP backbone. In the wild, not all subnets or routers of a specific AS will be visible to probes (e.g., because of filtering policies), meaning the graphs built by SAGE will not always be as complete as with our two groundtruth networks.

Hopefully, SAGE can be assessed in the wild by comparing the graphs captured on one target network but on different days and from distinct vantage points. As a reminder, all the data captured by SAGE for a given target network (usually an entire AS) on a given date and from a given (and single) vantage point is called a ***snapshot*** (cf. Sec. 6.2.1). If two snapshots from one AS captured from distinct vantage points on separate dates provide comparable neighborhood-based DAGs, in the sense that they provide similar or identical vertices and edges, then these DAGs are ***isomorphic*** to some extent. As a reminder, in graph theory, two graphs are isomorphic when they share the same number of vertices and if these vertices are connected in the same manner in both graphs. Finding isomorphic components in two DAGs as captured by SAGE from distinct vantage points is a way to show its hop-level inference is consistent regardless of the vantage point. Of course, two snapshots will likely never provide strictly identical DAGs (e.g., because the target AS may be routed differently depending on the vantage point), but a large amount of similarities between a pair or a collection of snapshots would imply that the same architecture was captured each time with a few different vertices and edges because of the change of vantage point.

The similarities between neighborhood-based DAGs can be quantified as follows. A set of snapshots collected for a given AS such that each snapshot was captured on a different date and from a different vantage point can be first selected. Then, the very first snapshot can be picked as a reference that will be compared with each subsequent snapshot. Second, for each pair, how many neighborhoods appear in both snapshots can be quantified by comparing their respective labels, i.e., how each neighborhood is uniquely identified. The non-best effort neighborhoods, also called ***regular*** neighborhoods, deserves particular attention as they are clearly identified by one or several router interfaces. Best effort neighborhoods (cf. Sec. 10.3.2), on the other hand, are less likely to always be identified in the same way due to differences in TTL distances (cf. Sec. 11.3.3) and routing.

The number of neighborhoods found identically in both snapshots (e.g., because they are identified by the same router interface(s)) can be divided by the total number of neighborhoods present in the reference snapshot to obtain a ***redundant vertex ratio (RVR)***. This ratio can be subdivided into two metrics by excluding or including best effort neighborhoods. The redundant vertex ratio without best effort neighborhoods will be simply denoted as ***RVR***, while the same ratio including best effort neighborhoods will be expressed as ***RVR with BE*** (**B**est **E**ffort). The former should be usually slightly higher than the latter, as best effort neighborhoods are less likely to appear identically in two snapshots of a same network. When the redundant vertex ratio (with or without best effort) is close to 1, this means that both snapshots provide a large number of identical vertices. Conversely, if the ratio is close to 0, then both snapshots differ greatly from each other. This can notably occur when one of both snapshots was captured from a vantage point with poor reachability.

The redundant vertex ratio(s) can be completemented by an additional metric, the ***intersection ratio (IR)***, which is computed by dividing the number of regular neighborhoods that appear in all selected snapshots without exception by the number of regular neighborhoods found in the reference snapshot. Having a high intersection ratio means that a large share of the (regular) neighborhoods identified in the reference snapshot always appear regardless of the vantage point.

When two snapshots share many neighborhoods in common, it becomes possible to quantify how many edges can exist in both snapshots: for any edge $u \rightarrow v$ found in the reference snapshot, if neighborhoods $u$ and $v$ exist in both snapshots, then $u \rightarrow v$ *can* exist in the second snapshot as well. The same edge is said to be redundant when it does appear in the second snapshot, i.e., the edge exists in both snapshots. The total of redundant edges can be divided by the total of edges that can appear in both snapshots to obtain a ***redundant edge ratio (RER)***. Therefore, a redundant edge ratio close to 1 expresses the fact that, for any edge that *can* exist in both snapshots, this edge is most likely redundant. Consequently, if two snapshots of a same target network captured from distinct vantage points feature a redundant vertex ratio and a redundant edge ratio that are both close to 1, then these snapshots provide neighborhood-based DAGs that are isomorphic to a great extent.

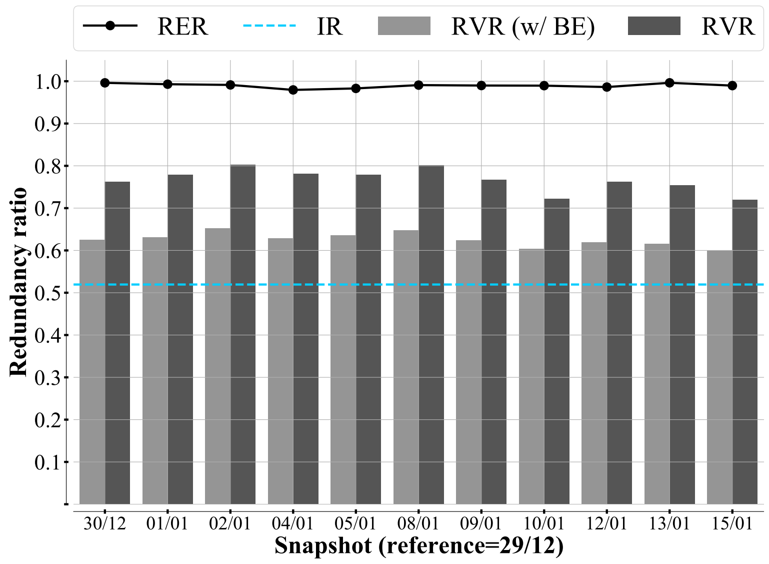


Figure 12.1: Graph isomorphism on AS6453 (December 29th, 2019 to January 15th, 2020).

Fig. 12.1 quantifies graph isomorphism for AS6453, using 12 snapshots collected from the PlanetLab testbed between December 29th, 2019 and January 15, 2020th, each snapshot being collected from a different vantage point. AS6453 (TATA Communications Inc.) was selected because it is a major Tier-1 AS found in the top 10 of CAIDA's AS ranking [2] that could also be measured on a daily basis. In the figure, the black line shows the redundant edge ratio of each comparison, i.e., the edges that exist in both the reference snapshot and the snapshot of the X-axis. Light gray bars show the redundant vertex ratios with best effort neighborhoods while the dark gray bars depict the same ratios but excluding best effort neighborhoods. Finally, the blue dashed line represents the intersection ratio, i.e., the ratio of regular neighborhoods that appear in all snapshots.

In Fig. 12.1, more than 70% of the regular neighborhoods appear in both the reference and the compared snapshot, while the redundant edge ratios are close to 1 (above 0.95) for all pairs of snapshots. When best effort neighborhoods are included, the redundant vertex ratios remain above or around 0.6, while the intersection ratio is slighty greater than 0.5. These results clearly suggest

SAGE captured a similar hop-level topology in each snapshot of AS6453, despite being a Tier-1 AS.

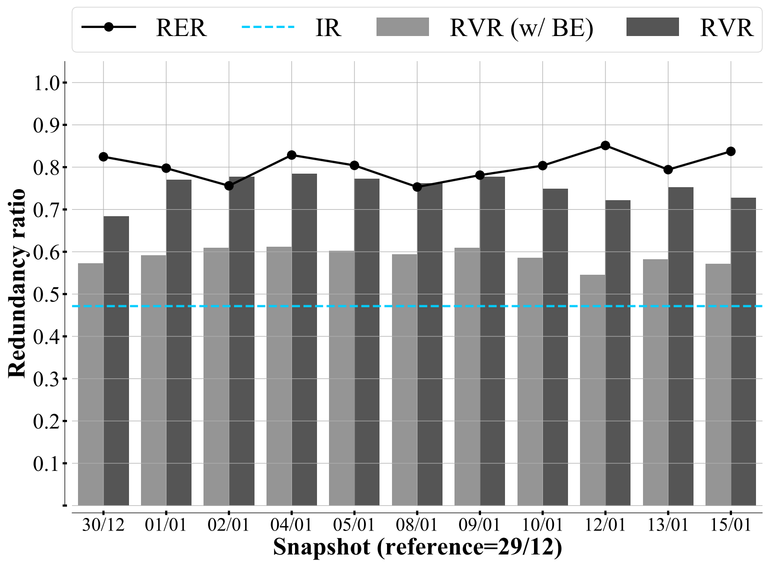


Figure 12.2: Graph isomorphism on AS3257 (December 29th, 2019 to January 15th, 2020).

To complement Fig. 12.1, Fig. 12.2 provides a comparable analysis (i.e., same methodology) for AS3257 (GTT Communications, Inc.), again using snapshots captured from PlanetLab in January 2020. Again, the redundant vertex ratios are good, with more than 70% of common regular neighborhoods for all pairs except for the comparison between the reference and the snapshot from December 30th, 2019. The redundant vertex ratios are also clearly above 0.5 if including best effort neighborhoods, thouh only a few snapshots feature around 60% of redundant vertices in this scenario. The redundant edge ratios are satisfying too but worse than previously: most pairs of snapshots provide a ratio above 0.8, while three pairs provide ratios slightly higher than 0.75. Such a difference to Fig. 12.1 could be explained by the fact that AS3257 is both a Tier-1 AS but also a larger target AS for SAGE: in fact, its IPv4 prefixes cover more than two millions of attributable IP addresses, while the IPv4 prefixes associated with AS6453 cover only slightly more than half a million of attributable IP addresses.

Finally, Fig. 12.3 provides the same visual tools as Fig. 12.1 and Fig. 12.2 but for AS224 (UNINETT), using the same methodology and dataset as before. This last figure provides better metrics overall, as only the redundant edge ratios are slightly worse than those depicted in Fig. 12.1 though they remain close to 1. In particular, the intersection ratio is excellent, as more than 70% of the regular neighborhoods from the reference snapshot reappeared in all subsequent snapshots. While excluding the best effort neighborhoods, the redundant vertex ratios are above 0.9 in several instances. These metrics should however not be unexpected, considering AS224 is a Stub AS (though it covers approximately one million of attributable IP addresses) and considering that both previous examples illustrated metrics for two Tier-1 ASes, which usually have more complex topologies.

All three previously discussed figures demonstrate that SAGE is able to capture comparable graphs despite changing the vantage point, even when considering complex networks. Interestingly, however, there is always a noticeably amount of vertices and edges that are not redundant. In particular, the

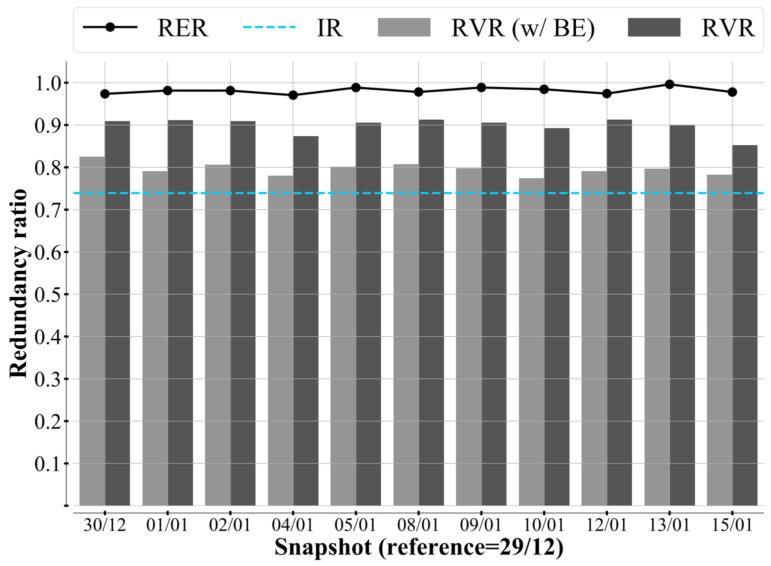


Figure 12.3: Graph isomorphism on AS224 (December 29th, 2019 to January 15th, 2020).

redundant edge ratios in Fig. 12.2 are clearly worse than in both other figures, being around 0.8 rather than above 0.95. As already discussed in Sec. 12.1.2, this could motivate combining several DAGs collected from different vantage points together: as comparable vertices could be found between two snapshots, it should be possible to merge the vertices and edges of two snapshots (and possibly other parts of the data, such as the inferred subnet-level) to produce a more exhaustive neighborhood-based DAG. Such a strategy is left for future work. It should be noted that this section only discussed the results for three ASes, but figures for other ASes listed in Table 12.2 can be browsed online at the SAGE public GitHub repository [5].

Finally, it should also be noted that redundant vertices quantified by the previous figures also include convergence points, though there is often only a small number of common convergence points between two snapshots: in the data discussed above, there is usually between 10 and 20% of convergence points from the reference snapshot that reappear identically in any other snapshot in the case of Transit/Tier-1 ASes, and sometimes up to 60% with Stub ASes. Such a result is not particularly surprising: because of the change of vantage point, there is little chance that the probes sent towards the target AS get routed exactly in the same way, therefore little chance they go through exactly the same load balanced paths. As a consequence, the observed convergence points are more likely to differ from one snapshot to another than other regular neighborhoods. This observation also suggests that the share of non-redundant vertices observed for each pair of snapshots might consist of unique paths/access points to the target network, while the redundant vertices constitute a kind of backbone which can be – in theory – observed from any vantage point. If multiple snapshots were to be combined into one unique DAG to get a comprehensive graph of the target domain, the convergence points observed in each snapshot should also be comprehensively checked and possibly merged by relying on alias transitivity (cf. Sec. 2.2).

[5] Cf. `https://github.com/JefGrailet/SAGE/tree/master/Python/Isomorphism/Figures`

## 12.3 Closing comments: on the potential of neighborhood-based DAGs

In this chapter, the consistency of the neighborhood-based DAGs built by SAGE [54] has been demonstrated. One the one hand, SAGE could be reliably validated on two groundtruth networks (Sec. 12.1). This validation demonstrated that both the discovered links and the subnets mapped to them were overwhelmingly true to the topology, though a small number of links were not correct for a variety of reasons, such as unexpected trails for a few subnets in the Belgian ISP backbone (because of a BGP configuration error). On the other hand, a large collection of snapshots captured on 12 ASes from the PlanetLab testbed in early 2020 showed that SAGE can discover comparable neighborhood-based DAGs for one target network regardless of the vantage point (cf. Sec. 12.2). In particular, the vast majority of graph edges that could exist in two snapshots of a pair (because the associated vertices were present in both) usually appeared in both, more than 95% of such edges being redundant in snapshots collected on AS6453 and AS224, respectively Tier-1 and Stub ASes (cf. Sec. 12.2.2).

Observations in the collected data also showed that SAGE regularly discovered convergence points through alias resolution, even in simple topologies such as the ULiège network (cf. Sec. 12.1.2). In fact, the present evaluation tends to suggest that SAGE is more likely to miss convergence points than to incorrectly infer them, since convergence points are an observed consequence of load balancing, which will vary from one vantage point to another, as already indicated while evaluating subnet inference challenges in a previous chapter (cf. Sec. 6.2.2). It is worth noting that additional practical results with SAGE also show that convergence points can usually be quite small in respect of the total number of alias candidates considered at once (cf. Sec. A.3.4).

All these results support the idea that neighborhood-based DAGs are a good paradigm to study the hop-level of intra-domain topologies, especially if alias resolution is eventually performed on each individual neighborhood to have a glance at the router-level (cf. Sec. 9.3.1 and Sec. 11.4). Moreover, while SAGE is designed so that it could easily run from a single vantage point, it could be potentially deployed from multiple vantage points to capture more exhaustive maps of some target domain by comparing and combining the collected snapshots, as already suggested in Sec. 12.2.2.

While neighborhood-based DAGs are very interesting from a measurement perspective, they may not, however, be the best way to model a target domain and study its topological properties. For instance, since several edges in a DAG can be mapped with the same subnet (e.g., if the subnet is a large one, such as a /25 subnet), the DAG itself is not the ideal way to account for the role played by the subnet-level, despite accounting accurately for the hop-level adjacencies. However, a neighborhood-based DAG can be easily translated into a bipartite graph (cf. Sec. 9.3.2): indeed, because neighborhoods are never connected directly but rather with a subnet in between, a bipartite graph where the $\top$ vertices correspond to neighborhoods while $\perp$ vertices account for subnets can be built. Moreover, if neighborhoods are further analyzed through alias resolution, it becomes possible to replace them with routers and Layer-2 devices, resulting in a tripartite graph that re-uses the ideas introduced by Tarissan et al. in 2013 [114]. The conversion of neighborhood-based DAGs into bipartite/tripartite graphs and what can be learned from them is discussed in Chapter 13.

# Chapter 13

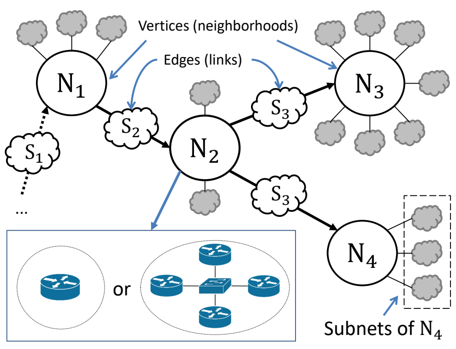
# Modeling networks with bipartite graphs

This final chapter analyzes the data collected by SAGE [54] with bipartite graphs to address research question 4 from Sec. 9.4. The question was to determine whether or not bipartite formalisms could be suitable to study the topology of the target networks measured by SAGE. While SAGE natively builds a comprehensive map of a target network as a neighborhood-based DAG (cf. Chapter 11), this model could be reinterpreted with bipartite formalisms to highlight topological features more easily. Sec. 13.1 presents two possible bipartite formalisms that could be applied on SAGE data to study the properties of the measured networks, while Sec. 13.2 summarizes how SAGE data was collected during 2020 and 2021 and subsequently reinterpreted with the proposed formalisms. Sec. 13.3 discusses the topological features that can be extracted from the first formalism, followed by a similar discussion with the second proposed model in Sec. 13.4. Finally, Sec. 13.5 concludes both this chapter and the third part of this thesis by summarizing its main contributions and discussing, as future work, ways to improve the SAGE methodology and its bipartite analysis.

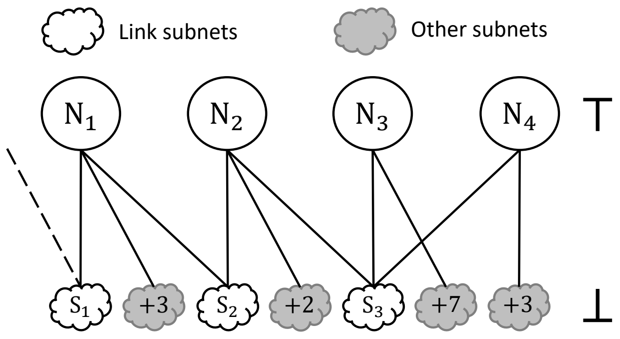
## 13.1 Bipartite models for SAGE data

Despite offering a comprehensive map of a target domain when annotated with subnet mappings (cf. Sec. 11.1.3), a neighborhood-based DAG as built by SAGE only accounts for the hop-level (cf. Sec. 9.3.1) when it comes to its structure. As such, it can provide insight on the hop-level via classical graph metrics, but requires additional processing to compute, for instance, additional metrics on the subnet-level. For reminders, the subnet mappings inferred by SAGE allow a same subnet to be mapped to multiple edges: indeed, it is possible a larger subnet (e.g., a /27 subnet) acts as a hub between several (meshes of) routers, therefore several neighborhoods. Moreover, it is also possible a subnet gets mapped at the same time to both direct or indirect links (cf. Sec. 11.1.3), i.e., the subnet mapped to an edge might not be in the vicinity of the neighborhoods connected through said edge. As a consequence, identifying the exact amount of neighborhoods a subnet is connecting together requires a careful reading of the neighborhood-based DAG, and discovering other topological features (such as structural cycles) involving subnets can require further analysis of the DAG.

Hopefully, studying the topological properties tied to the subnet-level can be achieved more easily by relying on a bipartite formalism (cf. Sec. 9.3.2). Indeed, a neighborhood-based DAG features two main components: neighborhoods and subnets. Moreover, by design, two adjacent neighborhoods should be linked together via an intermediate subnet. Likewise, two subnets cannot be directly connected and must have at least one intermediate (Layer-3) device between them, as a subnet delimits, by definition, a set of devices that can reach other directly at the datalink layer (cf. Sec. 5.1.1). The intermediate device(s) can be abstracted by a neighborhood. Therefore, neighborhood-based DAGs can be easily reinterpreted as neigborhood ($\top$) – subnet ($\perp$) bipartite graphs.



(a) Initial DAG



(b) As a bipartite graph (+X accounts for X subnets)

Figure 13.1: Translating a neighborhood-based DAG into a bipartite graph.

Figure 13.1 shows a toy example of a neighborhood-based DAG which is reinterpreted as a neighborhood ($\top$) – subnet ($\perp$) bipartite graph. Fig. 13.1a shows the initial DAG [1] while Fig. 13.1b depicts its bipartite equivalent. Not only this example demonstrates the original DAG can be easily reinterpreted as a bipartite graph without changing anything in the initial data, but it also shows that such a model can also reveal new adjacencies that were not initially visible in the `traceroute` data collected by `SAGE`. Indeed, in Fig. 13.1a, the neighborhoods $N_3$ and $N_4$ share no common edge, but the edges going from $N_2$ to $N_3$ and $N_4$ are mapped to the same subnet $S_3$. As a consequence, it can be inferred that $S_3$ is a hub between $N_2$, $N_3$, and $N_4$, which means $N_3$ and $N_4$ are adjacent too. A strict reading of the IP-level DAG arising from the initial `traceroute` data collected by `SAGE` would not have suggested this at first. Because $S_3$ is modeled by a single vertex in Fig. 13.1b, the bipartite graph naturally accounts for the adjacency of $N_3$ and $N_4$ without modifying the initial data.

Not only the neighborhood – subnet bipartite formalism can reveal new adjacencies, but it is also a convenient tool to get back to simpler graphs thanks to the mecanism of bipartite projection (cf. Sec. 9.3.2). For example, a projection on the neighborhoods ($\top$), also called $\top$***-projection***, can

[1]This figure is actually identical to Fig. 11.1. It is reused here to allow a convenient comparison.

be performed to generate a hop-level graph which accounts more accurately for the hop-level adjacencies than the initial neighborhood-based DAG: indeed, the new adjacencies revealed in the bipartite graph are also taken account of in the projection. As a result, this bipartite formalism makes it possible to study two distinct definitions of the ***neighborhood degree***: in a bipartite graph, this degree simply amounts to the number of neighboring subnets, therefore the number of edges shared with $\perp$ vertices, while the degree in the projection amounts to total number of adjacent neighborhoods. Finally, software libraries such as the NetworkX library for Python [34] provides many convenient tools to extract topological features from bipartite graphs and to visualize them.

However, the neighborhood – subnet bipartite model is not the only one that can be applied to `SAGE` data. For reminders, neighborhoods themselves can hide a single router as well as a mesh of routers, possibly involving Layer-2 equipment. Furthermore, `SAGE` itself ends its execution with a final alias resolution step (cf. Sec. 11.4) that is in practice very similar to the `TreeNET` alias resolution stept [51], i.e., it builds lists of alias candidates on the basis of each neighborhood (see also Sec. 4.1.1) and subsequently resolve them. Moreover, previous research already highlighted the practical effects of Layer-2 devices on the router degree [90] (cf. Sec. 9.3.1) and explored the possibility of including Layer-2 devices in a bipartite formalism [114] (cf. Sec. 9.3.2). The neighborhood – subnet paradigm can therefore be extended as follows: first, the neighborhoods themselves can be replaced with the routers they contain, using alias resolution results previously collected by `SAGE`. Second, the router – subnet bipartite graph can be completed by a third party that represent (hypothetical) Layer-2 devices. This additional party accounts for the connections that exist between routers of a same neighborhood, hinting a full mesh rather than a single device.

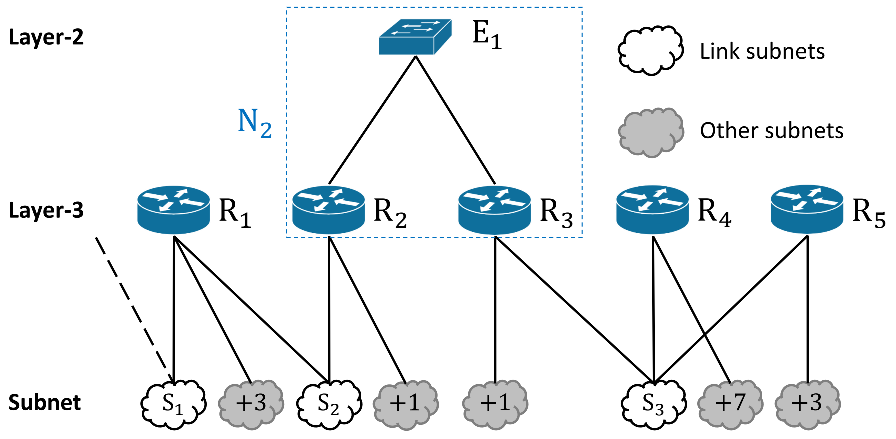


Figure 13.2: The DAG from Fig. 13.1a reinterpreted as a tripartite graph. Note the dashed square, symbolizing that $E_1$, $R_2$ and $R_3$ constitute a mesh initially identified as neighborhood $N_2$.

Figure 13.2 depicts a possible tripartite graph that could be derived from Fig. 13.1a. In this new example, it is assumed that $N_1$, $N_3$, and $N_4$ have all been identified as being single routers (rather than meshes) while the alias resolution (hypothetically) discovered two routers within $N_2$. There are therefore five routers: $R_1$ ($N_1$), $R_2$ and $R_3$ ($N_2$), $R_4$ ($N_3$), $R_5$ ($N_4$). To account for $R_2$ and $R_3$ being found in the same neighborhood, an hypothetical Layer-2 device $E_1$ (for **E**thernet switch, a common type of Layer-2 equipment) is added to the tripartite graph (top of Figure 13.2) to connect both routers together, Figure 13.2 highlighting the original neighborhood $N_3$ to ease comparison with Fig. 13.1b. As a result, the final Layer-2 – Layer-3 – subnet graph provides the same adjacencies as the original DAG, but includes the router-level knowledge gained by SAGE during its final step.

The main advantage of the tripartite model is its suitability to study and compare the router-level with the hop-level (see also Sec. 9.3.1). Indeed, in this model, the router degree is at best equal to the neighborhood degree (bipartite definition), i.e., the degree of each individual router is equal to the degree of the encompassing neighborhood if it turns out – via alias resolution – to consist of a single device. When a same neighborhood features multiple routers, the presence of Layer-2 equipment may also be inferred by assessing whether or not it would be advantageous. For instance, identifying a dozen of routers within the same neighborhood strongly suggests there might be Layer-2 equipment, because having a single data link layer connection with a Layer-2 device to obtain a full mesh is much simpler than having a dozen additional connections for each router. It should be noted that Layer-2 components are also used in practice to handle data link connections within medium- to large-size subnets, but this does not induce a change between the subnet perceived prefix and its actual prefix, while Layer-2 devices interconnecting routers into meshes have a significant impact on their perceived one-hop adjacencies. This latter issue is what the tripartite model is meant to study, hence why Layer-2 devices, in this model, only account for devices involved in meshes of routers.

Both models are discussed in this chapter. Sec. 13.3 reviews the neighborhood – subnet model, while Sec. 13.4 comments the Layer-2 – Layer-3 – subnet model. In both cases, how the neighborhood-based DAGs can be translated in practice into each model will be explained, and various results obtained from snapshots collected by SAGE from the EdgeNet cluster will presented and discussed. The practical details of the deployment of SAGE on EdgeNet are provided in the next section.

## 13.2 Getting and handling SAGE data for bipartite analysis

Due to the planned termination of the PlanetLab testbed by May 2020 [99], data collection with SAGE had to be moved to the EdgeNet cluster [108], a new system dedicated to network research created by the former PlanetLab Europe staff. Moreover, to analyze SAGE data easily while minimizing the implementation work, a dedicated library in Python involving the NetworkX library [34] has been designed in 2020 and progressively extended since then. Sec. 13.2.1 briefly comments on how the EdgeNet cluster works and how SAGE has been deployed from it. Sec. 13.2.2 introduces INSIGHT, the custom Python library used to manipulate and analyze SAGE data with the help of NetworkX.

### 13.2.1 SAGE deployment on the EdgeNet cluster

Created early in the early 2000's, the PlanetLab testbed has been a very useful tool for network research as a whole as well as this thesis: in particular, it has been used to deploy TreeNET [51, 55] and WISE [52, 53] as well as to conduct the very first measurements with SAGE [54] (cf. Sec. 12.2.1).

However, as already hinted several times in this work (see notably Sec. 6.2.1), the PlanetLab testbed was very cumbersome to maintain: dedicated machines had to be installed around the world and were maintained by centralized authorities. Because of the vast amount of machines [2] and centralized management, nodes were updated at a slow pace. Late in the 2010 decade, many PLC nodes (**P**lanet**L**ab **C**entral) were still running with the Fedora 8 operating system, a Linux distribution which was released on November 8th, 2007 and whose support ended on January 7th, 2009. The termination of the PlanetLab testbed was eventually announced for May 2020 [99].

Of course, since the early 2000's, several other platforms have been proposed for network monitoring and/or topology mapping, such as CAIDA's Archipelago [1], RIPE Atlas [9] or MONROE [7]. However, these platforms are not suitable to deploy `WISE/SAGE` because they have either specific goals (e.g., the MONROE project is focused on mobile networks) or restrictions when it comes to the tools that can be deployed (e.g., RIPE Atlas [9] only allow basic tools, such as `traceroute` [119] and `ping`). To offer the same degree of freedom as PlanetLab while modernizing its infrastructure, the late PlanetLab Europe staff created the EdgeNet cluster [108]. The EdgeNet cluster has been designed with the state-of-the-art technologies in cloud computing, as it relies on Docker [3] to containerize applications and Kubernetes [6] to deploy them. As such, it allows any programming environment on any EdgeNet node, the nodes themselves being simpler to manage than past PlanetLab nodes.

However, the EdgeNet cluster is still in its infancy, having around 50 available nodes at the time of writing these lines (to compare with the hundreds of PlanetLab nodes). Moreover, due to Kubernetes adding resource management in the equation, node availability also depends on the current node usage. Because of this, but also because EdgeNet features more working nodes than PlanetLab as of early 2020, `SAGE` measurements from the EdgeNet cluster no longer involves vantage point rotation (as first defined in Sec. 6.2.1). Instead, a dynamic scheduling strategy was designed so that nodes that appear to have limited resources are avoided while each target AS gets measured from as many different vantage points as possible. This strategy, implemented with a mix of Bash and Python scripting, allows one to perform frequent measurements from a large variety of vantage points in an semi-automated manner. The scripts are publicly available on the SAGE GitHub repository [3].

Though the measurement strategy slightly changed with respect to Sec. 12.2.1, the target Autonomous Systems remained more or less the same, as shown by Table 13.1. The main changes in Table 13.1 with respect to Table 12.2 are the removal of AS8928 (Interoute Communications) and the addition of two Transit networks, AS1273 (Vodafone Group) and AS4637 (Telstra International), and one additional Tier-1 network: AS6461 (Zayo Group). AS8928 was removed from the measurement campaign due to most EdgeNet nodes having poor reachability towards this specific network, while the three additional ASes were selected with the same methodology as before, i.e., using CAIDA's AS ranking [2] and the Hurricane Electric BGP toolkit [4] to select major networks whose IPv4 prefixes could be fully probed by `SAGE` in a few hours.

[2]PlanetLab had 1353 nodes in 717 sites at its peak [8].

[3]`https://github.com/JefGrailet/SAGE/tree/master/Dataset/Scripts`

| ASN | Name | Origin | Category |
|---|---|---|---|
| 12956 | Telefonica Global Solutions | Spain | Tier-1 |
| 3257 | GTT Communications Inc. | USA | Tier-1 |
| 6453 | TATA Communications | USA | Tier-1 |
| 6461 | Zayo Group | USA | Tier-1 |
| 6762 | Sparkle | Italy | Tier-1 |
| 1273 | Vodafone Group PLC | European Union | Transit |
| 13789 | Internap Corporation | USA | Transit |
| 286 | KPN B.V. | Netherlands | Transit |
| 4637 | Telstra International Limited | Hong Kong | Transit |
| 50673 | Serverius Holding B.V. | Netherlands | Transit |
| 6939 | Hurricane Electric LLC | USA | Transit |
| 9198 | JSC Kazahtelecom | Kazakhstan | Transit |
| 1241 | Forthnet | Greece | Stub |
| 224 | UNINETT | Norway | Stub |

Table 13.1: Summary of the 14 ASes probed with `SAGE` from the EdgeNet cluster in 2021.

### 13.2.2 Analyzing `SAGE` data with `INSIGHT`

In order to easily transform and analyse the data captured by `SAGE` from either the PlanetLab testbed or the EdgeNet cluster, a custom Python library has been designed. The choice of Python as a programming language is motivated not only because the language features many useful libraries for plotting figures, but also by the possibility to take advantage of the NetworkX library [34]. The NetworkX library is indeed a very useful tool to analyze graphs of all kinds, providing both a large variety of metrics and tools for rendering a graph as a 2D PDF figure. Of course, the data that should be fed to NetworkX graph building utilities must be first pre-processed, or at least managed so that the final graphs account for all adjacencies observed directly in the `SAGE` data or inferred from it (cf. Sec. 13.1. Moreover, additional details on subnets or discovered alias pairs/lists (i.e., discovered during the final step of `SAGE`, see Sec. 11.4) should also be extracted from a snapshot to perform additional verifications or transformations for consistency's sake.

The `INSIGHT` library (**I**nvestigate **N**etworks from **S**ubnet **I**nference to **G**rap**H** **T**ransformations) combines all at once the NetworkX library, the plotting utilities of Python (such as `pyplot`) and custom modules to parse and manage `SAGE` data so that `SAGE` snapshots can be quickly translated then analyzed as bipartite graphs. Moreover, having `INSIGHT` as a side library ensures the way `SAGE` maps a target network is completely decoupled from how the resulting the data is interpreted. Just like `SAGE` itself, `INSIGHT` is open source: in fact, it is provided in the same GitHub repository due to being inseparable from `SAGE` itself [4]. The intricate implementation details of `INSIGHT` are out of the scope of this document; however, guidelines for processing the raw data (i.e., as provided by `SAGE`) and turning it into bipartite graphs are subsequently described in Sec. 13.3.1 and Sec. 13.4.1.

[4] `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT`

## 13.3 A simple bipartite model: neighborhood (⊤) – subnet (⊥)

This section discusses in details the first bipartite model mentioned in Sec. 13.1, i.e., the neighborhood (⊤) – subnet (⊥) bipartite formalism. Sec. 13.3.1 first provides a few guidelines for translating a neighborhood-based DAG built by `SAGE` into this first bipartite model, while Sec. 13.3.2 presents interesting observations made in the wild on various target networks.

### 13.3.1 Guidelines for building a simple bipartite graph

Building a neighborhood – subnet bipartite graph is not a hard task: the vertices of the original neighborhood-based DAG are re-used "as is" as the set of ⊤ vertices, while the ⊥ vertices can be easily computed by reading the subnet mappings while not creating duplicate vertices when a subnet turns out to be mapped more than once to some edge in the initial DAG. Moreover, direct links (as defined in Sec. 11.1.3) are immediate to account for in the bipartite graph: once the corresponding vertices (two neighborhoods and one subnet) are identified, all it takes to account for the adjacency of both neighborhoods and the intermediate subnet is to insert the corresponding edges. I.e., if the neighborhood-based DAG announces `N2 → N3 via 10.0.8.0/30`, assuming both neighborhoods keep the same labels ($N_2$ and $N_3$) while 10.0.8.0/30 is labeled as $S_2$, then the edges $N_2 \leftrightarrow S_2$ and $N_3 \leftrightarrow S_2$ can be simply added to the bipartite graph to account for this link [5].

Of course, other types of edges (as defined in Sec. 11.1.3) have to be handled with more care. Each type of edge, other than direct links, will be subsequently denoted as a ***(translation) scenario***. Three such scenarios have to be addressed:

1. indirect links with a subnet mapping,

2. indirect links without a subnet mapping, also called ***incomplete links***,

3. remote links.

Scenario 1 can be considered as a variant of the translation of a direct link. I.e., the same bipartite edges will be created, but a third additional edge should be created to connect the subnet with the neighborhood it borders, as long as this edge does not exist already [6]. For example, if the initial DAG provides the indirect link `N2 → N3 via 10.0.8.0/28 (From N5)`, assuming all three neighborhoods keep the same labels ($N_2$, $N_3$ and $N_5$) while 10.0.8.0/28 is labeled as $S_2$, then the indirect link can be accounted for by inserting the three following edges: $N_2 \leftrightarrow S_2$, $N_3 \leftrightarrow S_2$ and $N_5 \leftrightarrow S_2$. Of course, the edge $N_5 \leftrightarrow S_2$ has to be inserted only if it does not exist yet, unless the implementation automatically ignores insertions of duplicate edges.

Scenarios 2 and 3 can be handled in a similar manner. In particular, in scenario 2, the subnet connecting the neighborhoods could not be found in the subnet-level data, either because the second

[5] The single direction found in `SAGE` data disappears, as network links are not directed.

[6] It is worth noting duplicate edges are not an issue with NetworkX [34], as it automatically ignores duplicate edges.

neighborhood is a best effort one (cf. Sec. 10.3.2), meaning it is not clearly identified by one or several router interface(s), either because the connecting subnet was invisible to probes. Scenario 3 is a generalization of this issue, because there are not one but several unknown subnets and at least one unknown neighborhood. A simple way to account for both scenarios consists of using ***hypothetical subnets***, i.e., subnets that are not part of the data but should exist in the topology. In `INSIGHT`, such subnets are labeled differently: while regular (real) subnet vertices are named $S_x$, subnets used for incomplete links are named $T_y$ and subnets accounting for remote links are named $U_z$. All three $x$, $y$, and $z$ are unique integers, either in general either with respect to their own category of subnet (e.g., $S_1$ and $T_1$ can coexist because the prefix letter already distinguishes them).

For each edge (in the DAG) corresponding to either Scenario 2 or Scenario 3, an hypothetical subnet can be created then linked to both neighborhoods connected by the edge. Not only these hypothetical subnets account for the adjacencies found in the initial DAG, but labeling them so they are clearly distinguished from regular subnets (like in `INSIGHT`) allows to include or omit them in function of the subsequent operations. For instance, while computing the distribution of the subnet degree (i.e., how many $\top$ vertices a subnet is connected to), hypothetical subnets should ideally be omitted from the distribution to focus on inferred subnets, especially since hypothetical subnets always have a degree of two if one such subnet is created for each instance of Scenarios 2 and 3.

It is important to note that, after fully processing the neighborhood-based DAG and before computing any metrics from the bipartite graph, any subnet found in the `SAGE` snapshot that is not mentioned among the subnet mappings from the DAG should also be added to the bipartite graph. These subnets, which can be denoted as ***degree-1 subnets***, are important to compute accurate metrics for $\top$ vertices, i.e. neighborhoods, such as their final degree, i.e., how many subnets they are connected to (which typically includes the subnets mapped to incident edges and the bordering subnets). However, the bipartite graph without degree-1 subnets can be very convenient to create visual renders: because degree-1 subnets add little information, a visual render only shows the subnets that act as the links between neighborhoods. Moreover, by colouring subnets depending on whether they are real or hypothetical, the one-hop adjacencies can be clearly differentiated from multi-hop adjacencies while reviewing the render. Fig. 13.3 depicts the topology of AS13789 (a small Transit AS) as a neighborhood – subnet bipartite graph, as rendered by `INSIGHT` using a snapshot captured on March 14th, 2021 from the EdgeNet cluster. The hypothetical subnets accounting for remote links are green in color while other hypothetical subnets are light red. Note that Fig. 13.3 only shows the main connected component, other small components being visible in the full render [7].

### 13.3.2 Observations in the wild

A wide collection of snapshots of the ASes listed in Table 13.1 have been captured by `SAGE` from both the PlanetLab testbed and the EdgeNet cluster in 2020 and from the EdgeNet cluster only during Spring 2021. These snapshots have been subsequently processed into simple bipartite graphs

[7] `https://github.com/JefGrailet/SAGE/blob/master/Python/INSIGHT/Results/AS13789/2021/03/14/`

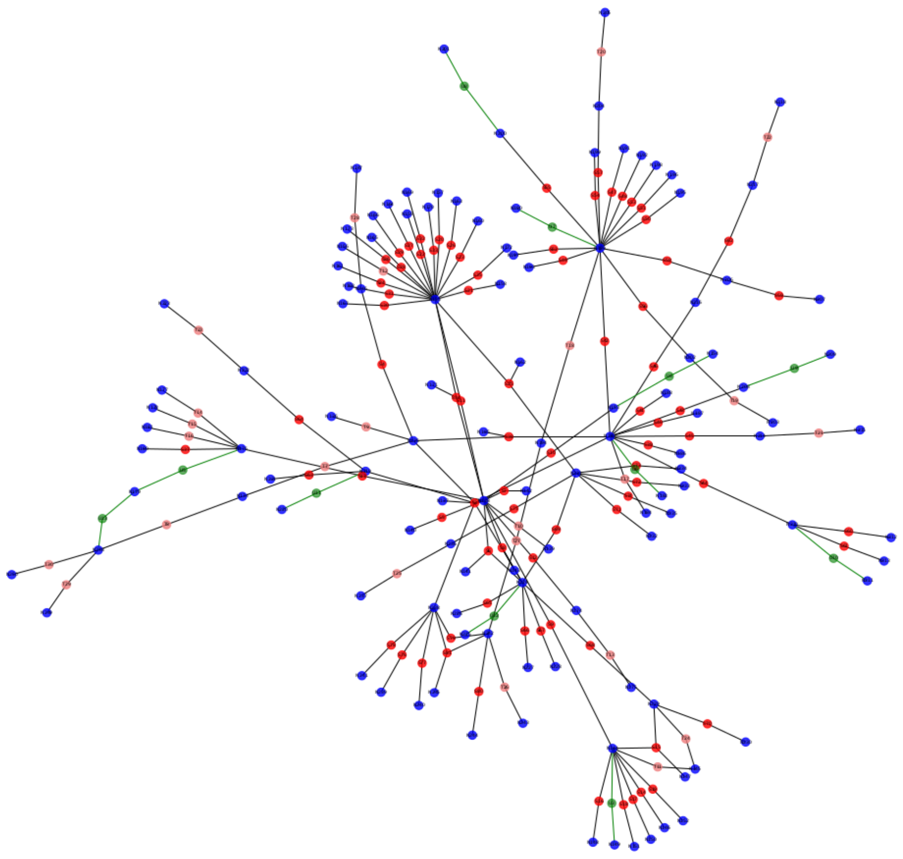


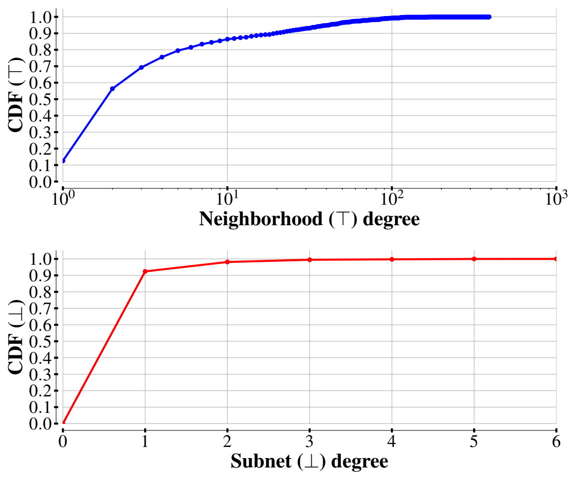
Figure 13.3: Render of AS13789 as a bipartite graph (March 14th, 2021; EdgeNet). Blue and red vertices respectively model neighborhoods and subnets (light red vertices are hypothetical). Green vertices and edges account for remote links (multiple hops paths that could not be identified by SAGE probes).

(as described in Sec. 13.1) with INSIGHT, and their analysis led to several interesting observations. Sec. 13.3.2.1 first discusses the degree distribution of both ⊤ and ⊥ vertices (i.e., neighborhoods and subnets, respectively) and the local density of the former, and how both may be linked to the role played by the modeled network in the Internet. Sec. 13.3.2.2 then comments on the structural cycles discovered within the bipartite graphs and what they imply regarding the Internet. Sec. 13.3.2.3 subsequently compares both definitions of the neighborhood degree (cf. Sec. 13.1), i.e., in the initial bipartite graphs and their ⊤-projections, and explains how the distributions corroborates previous claims in the field of topology discovery. Finally, Sec. 13.3.2.4 shortly comments some residual issues observed in the graphs, and how they could be prevented or mitigated. Note that all results presented in this section constitute a small part of the figures and metrics generated by INSIGHT, which can be all browsed at the SAGE GitHub repository, along with visual renders of the graphs [8], which are provided as additional information.

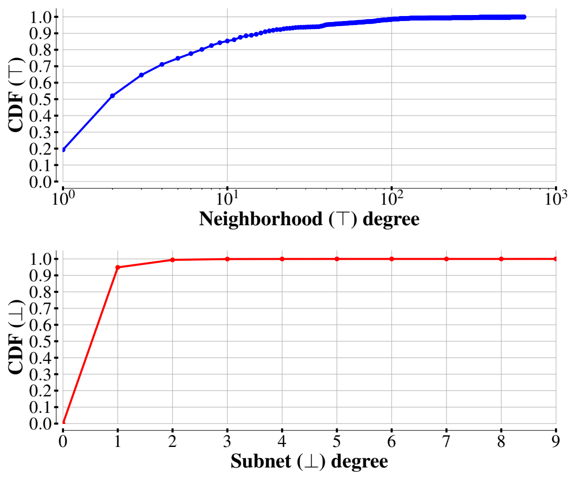
[8] `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT/Results`

#### 13.3.2.1 Degree distribution (⊤ and ⊥) and local density (⊤)

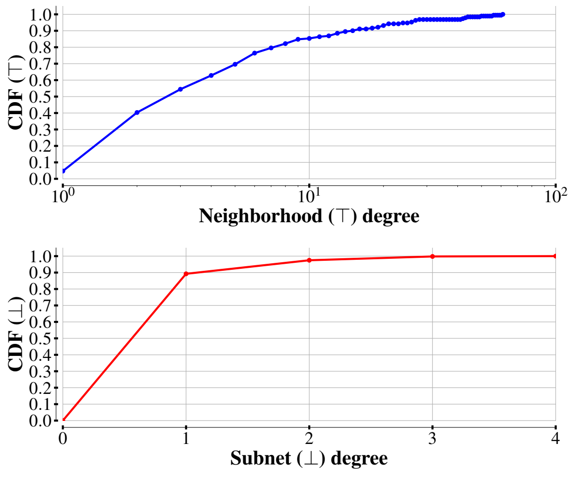
One of the key advantages of the neighborhood – subnet bipartite formalism is its suitability to study the ***subnet degree***. In this context, the subnet degree is defined as the total number of neighborhoods, i.e. hop-level nodes, a same subnet binds together in the topology. Typically, a /30 or /31 subnet acting as a point-to-point link between two neighborhoods has a degree or 2, while a larger subnet (e.g., /29) acting as a hub between four distinct neighborhoods has a degree of 4. Obviously, this degree is strictly equivalent to the total number of edges to which a ⊥ vertex (therefore a subnet) is connected in the bipartite graph. This property arises naturally from the bipartite conversion itself (cf. Sec. 13.3.1), as a same subnet being mapped to multiple edges in a neighborhood-based DAG ends up sharing an edge with each neighborhood it connects in the initial neighborhood-based DAG built by `SAGE`. Analyzing both the neighborhood degree (in the bipartite graph) and the subnet degree can shed light on the measured topology and its role in the Internet as a whole.



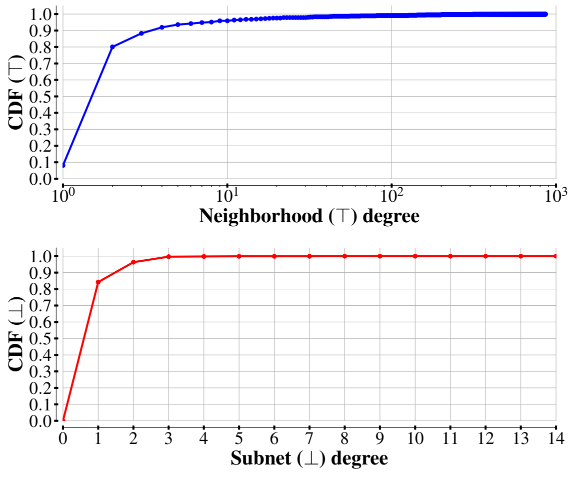
(a) AS6453 (TATA Com.), September 20th, 2020



(b) AS6939 (H.E. LLC), September 20th, 2020



(c) AS286 (KPN B.V.), September 20th, 2020



(d) AS1241 (Forthnet), September 12th, 2020

Figure 13.4: ⊤ and ⊥ degree distribution for various Autonomous Systems.

Figure 13.4 provides **c**umulative **d**ensity **f**unctions (CDFs) for both the neighborhood (⊤) degree and the subnet (⊥) degree of bipartite graphs obtained from four snapshots of distinct Autonomous Systems: AS6453 (TATA Communications; Tier-1), AS6939 (Hurricane Electric LLC; Transit), AS286 (KPN B.V.; Transit) and AS1241 (Forthnet; Stub). All four snapshots were captured in September 2020 from the EdgeNet cluster. While the provided figures relate to four specific snapshots, it should be noted that the depicted distributions (and other topological features) are representative of the snapshots captured over the same period for the same ASes: variations can be observed when confronting distributions from Figure 13.4 with those obtained on other snapshots, but the orders of magnitude and overall shapes remain comparable [9].

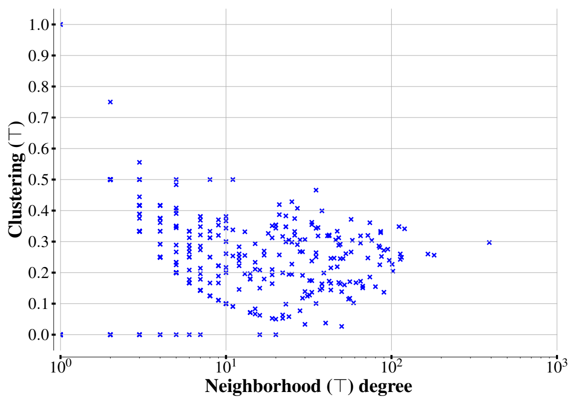
Within Figure 13.4, the CDFs of AS1241 (Fig. 13.4d) stand out. Indeed, the CDF for ⊥ vertices (subnets) shows slightly more than 15% of the discovered subnets (3,052 in this snapshot) had a degree of two or more, with a maximum of 14 (corresponding to an inferred /20 subnet). Meanwhile, the CDF for ⊤ vertices (neighborhoods) suggests there were a few very high degree neighborhoods, most neighborhoods (i.e., above 90%) having a degree smaller than or equal to 10. The CDFs for AS6453 (Fig. 13.4a) and AS6939 (Fig. 13.4b) rather show their respective neighborhoods had higher degrees overall, but smaller maxima. Likewise, their subnets had smaller maxima too, and the share of subnets having a degree of two or more was noticeably smaller, though they hid dozens of such subnets in practice due to the large number of discovered subnets featured in both snapshots (8,188 for AS6453 and 10,645 for AS6939). Such observations can be explained by the roles played by these ASes: AS1241 is indeed operated by a Greek ISP, while AS6453 and AS6939 are respectively Tier-1 and Transit, both being featured in the top 10 of CAIDA's AS ranking [2]. As such, the latter ASes might rely more often – on average – on point-to-point links, perhaps involved in traffic engineering schemes, while large subnets might be used within AS1241 to connect the most important (meshes of) routers with other parts of the network more easily [10].

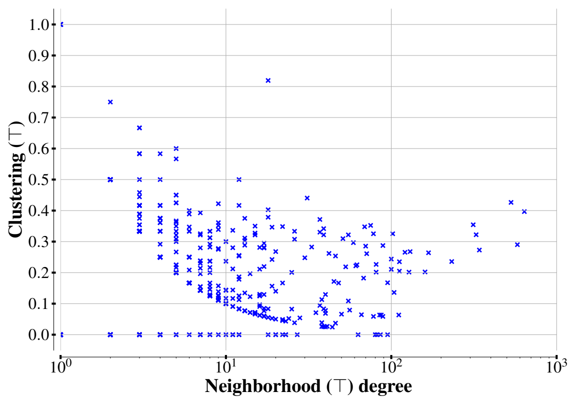
Interestingly, AS6453 had smaller maxima than both AS6939 and AS1241 despite being a Tier-1 AS. However, the smallest maxima were found in AS286 (Fig. 13.4c), though its ⊥ distribution was reminescent of the one observed for AS6453 (Fig. 13.4a) while its ⊤ CDF was comparable to that of AS6939 (Fig. 13.4b). The lower maxima can be explained by the fact that much less subnets were discovered in AS286 at the time: on September 20th, 2020, only 995 subnets were discovered and covered roughly 7% of the IPv4 prefixes announced for this AS. Naturally, other snapshots from the same period provide comparable numbers. Such a small picture of the target domain might account for a backbone, the rest of the network being shielded from `SAGE` probes via filtering policies. It is nevertheless interesting to see that, even in such a small snapshot, the distributions of both ⊤ and ⊥ vertices were comparable to those observed on much larger snapshots for Tier-1 and Transit ASes and very different from those of a Stub AS such as AS1241.

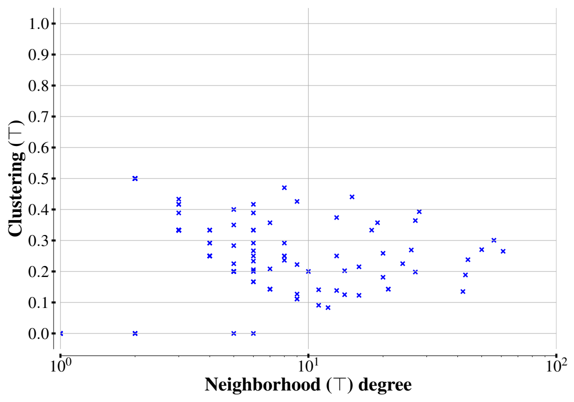
To complete the distributions depicted in Figure 13.4, Figure 13.5 provides, for the same bipartite

[9]Given the evaluation provided in Sec. 12.2.2, this should not come as a surprise.

[10]This is actually visible in the associated visual render as red vertices with multiple edges towards blue ones, cf. `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT/Results/AS1241/2020/09/12/`.


(a) AS6453 (TATA Com.), September 20th, 2020


(b) AS6939 (H.E. LLC), September 20th, 2020


(c) AS286 (KPN B.V.), September 20th, 2020

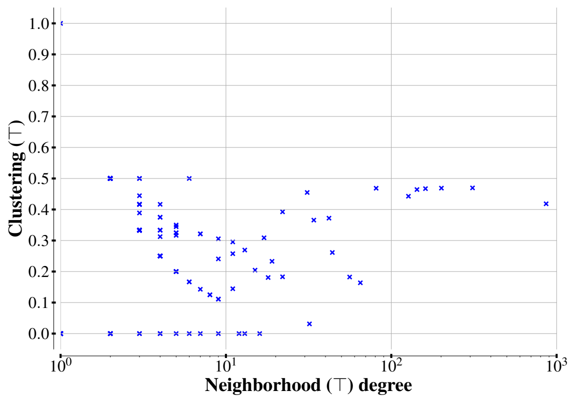

(d) AS1241 (Forthnet), September 12th, 2020

Figure 13.5: Clustering coefficients of ⊤ vertices in function of their degree.

graphs built from the same snapshots, the clustering coefficients of the neighborhoods (Y axis) in function of their degree (X axis). For reminders, the clustering coefficient is a classical graph metric evaluating the probability that two vertices sharing a common neighbor vertex also shares a common edge. It is typically used to assess the local density of vertices, i.e., the tendency of vertices to cluster themselves together in a graph. In the case of Figure 13.5, the coefficients were computed with a formula specifically designed by Latapy et al. in 2008 for bipartite graphs [77]. More precisely, the *min-clustering* formula was used: this version of Latapy's bipartite clustering coefficient was designed for cases where the vertices whose coefficients are being computed tend to have higher degrees than vertices from the other party. This is typically the case of ⊤ vertices in a neighborhood – subnet bipartite graph, as already demonstrated by Figure 13.4.

Figure 13.5 notably highlights that the clustering coefficients can vary considerably for a same neighborhood degree (bipartite definition). In particular, results for AS6453 (Fig. 13.5a) and AS6939 (Fig. 13.5b) took the appearance of point clouds. The same could be said for AS286 (Fig. 13.5c) and AS1241 (Fig. 13.5d), though there are fewer neighborhoods to consider, resulting in less striking figures. Another result is that high degree neighborhoods did not necessarily feature a high clustering coefficient: for instance, in the case of AS286 (Fig. 13.5c), the coefficients of the highest degree neighborhoods were around or under 0.3. The same could be said about AS6453 (Fig. 13.5a), where the highest degree neighborhood had a lower coefficient than many nodes whose degrees were comprised between 10 and 100. Interestingly, however, the highest degree neighborhoods from the bipartite graph generated for AS1241 (Fig. 13.5d) featured high clustering coefficients in respect of the large majority of ⊤ vertices, reaching almost 0.5 for some. Such metrics suggests AS1241 consists of a few very important (meshes of) routers with high local density [11], as already hinted by Fig. 13.4d. Finally, clustering coefficients for AS6939 (Fig. 13.5b) suggest the highest degree ⊤ vertices also tended to have a higher local density than most vertices, though they were comparatively lower with respect to those of AS1241, in addition to being contrasted by a large number of neighborhoods with small clustering coefficients. In comparison, the coefficients for AS6453 appeared to be more uniformly distributed. This further reinforces the idea that the observed topology of AS6453 was the most distributed of all, while AS1241 definitely standed out as a topology due to featuring a few very high degree neighborhoods with high local density. Both observations could be linked to the role assumed by each AS in the Internet.

#### 13.3.2.2 Topological cycles

Simple bipartite graphs can also constitute tools to study ***topological cycles***. In this context, a topological cycle simply refers to a graph cycle upon considering a graph mapping the topology of a target network, and as such, should not be confused with cycles experienced during `traceroute` probing: while the latter might be caused by the former, previous research demonstrated `traceroute` cycles could also be experienced due to traffic engineering or MPLS tunnels [85]. What also makes bipartite graphs interesting for this purpose is that they are non-directed: indeed, neighborhood-based DAGs are directed by design to better reflect the `traceroute` data and do not feature cycles unless `traceroute` problems induce them [12]. Moreover, simple bipartite graphs not only are non-directed but also provide new adjacencies by design (cf. Sec. 13.1). Therefore, they are well suited to highlight topological cycles, even though such cycles take a specific shape: indeed, each cycle features an even amount of vertices as well as an alternation of neighborhoods and subnets due to the bipartite formalism. This section will also focus on ***base cycles***, i.e., cycles that cannot be decomposed into simpler cycles, notably because the NetworkX library [34] used by `INSIGHT` already provides all the tools to extract those cycles from a given bipartite graph.

Before going any further, it should be noted that actual instances of cycles in bipartite graphs could be observed and subsequently assessed with `INSIGHT`. In particular, the `SAGE` snapshot of the Belgian ISP backbone discussed in Sec. 12.1 contained three topological cycles after bipartite conversion. These cycles could be validated at the hop-level, i.e., the neighborhoods involved in the cycles were correct and did feature the adjacencies enumerated by each cycle. However, the subnets involved in these cycles were not entirely correct, as each involved one of the six subnets which

[11] This is made even more obvious by the visual render of the bipartite graph: `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT/Results/AS1241/2020/09/12/`.

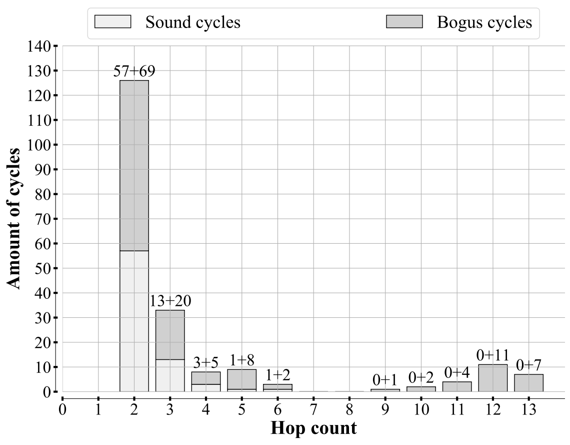
[12] Some were observed with complex topologies in the later development stages of `SAGE`, but this is uncommon.

were attributed to the wrong neighborhood because of a BGP configuration error (cf. Sec. 12.1.2). Nevertheless, the neighborhoods they connected still shared a common link in the groundtruth.

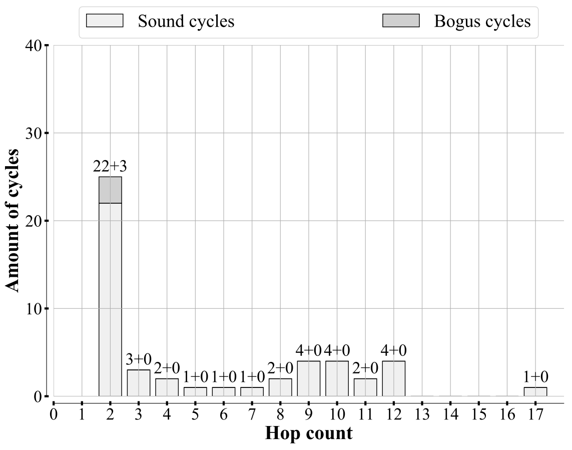
However, looking for cycles with the `SAGE` snapshot of the ULiège network (cf. Sec. 12.1) also resulted in the detection of two topological cycles, which both involved the two neighborhoods which corresponded to the undetected convergence point (cf. Sec. 12.1.2). As a consequence, topological cycles detected by `INSIGHT` (with the help of NetworkX) should be handled with special care, because they can result from measurement artifacts such as undetected convergence points, routing issues or overgrown subnets. One way to detect potential artifacts consists of checking whether the subnets involved in each cycle are large enough for their respective degree. E.g., a degree of 10 for a /29 subnet is obviously not possible, because such a subnet can only host up to 8 interfaces and therefore connect at most 8 devices (considering it uses the prefix and broadcast addresses too, despite subnetting conventions [92]). Cycles featuring subnets with inconsistent degree can therefore be ruled as ***bogus cycles***, i.e., they might be the consequence of measurement artifacts. Conversely, cycles where each subnet features a consistent degree can be deemed as ***sound cycles***. For instance, the two cycles found in the ULiège topology are deemed as bogus cycles by `INSIGHT`. It shoud also be noted that the so-called bogus cycles may not necessarily be erroneous in reality: they can also be the result of undergrown subnets, in which case said cycles might be true to the topology anyway.

Figure 13.6 shows the distributions of both sound and bogus cycles found in bipartite graphs resulting from the same four snapshots as in Figure 13.4 and Figure 13.5. Each bar stacks the number of bogus cycles on top of the number of sound cycles, and is annotated with the sum of the raw numbers, i.e., $X + Y$ means there were $X$ sound cycles and $Y$ bogus cycles. The cycles are also quantified by their number of hops rather than their actual length in the bipartite graph: indeed, due to the bipartite formalism, their actual length is always an even number due to the alternation of neighborhoods and subnets. Dividing this actual length by two therefore gives how many hops are needed to complete the cycles. Because a cycle is at least two hops long, no distribution advertises topological cycles for $x = 1$. Two hops cycles are also a particular case, as they correspond to going back and forth between two neighborhoods: as such, they can be considered to be back-up links.

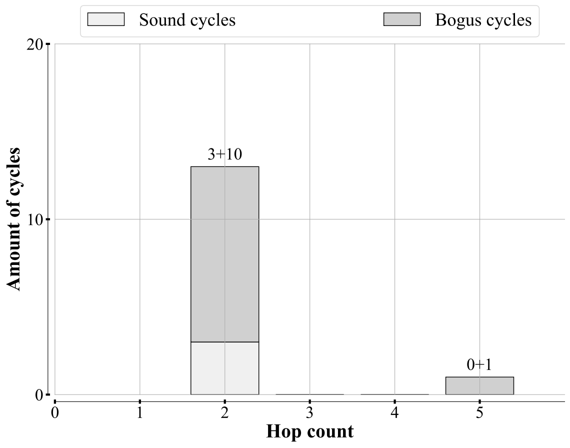
Figure 13.6 provides two results. First, potential back-up links tend to be the most common occurrence of topological cycles (regardless of being sound or bogus). Second, there can be a large variety of cycles within the same topology regardless of the nature of the network. In particular, the bipartite graph generated for AS1241 (Fig. 13.6d) includes numerous sound topological cycles, the longest being eight hops long. The noticeable share of subnets featuring a large degree within this specific topology (cf. Fig. 13.4d) partially explains why topological cycles appeared to be more common than in other measured topologies, though AS6939 also appeared to feature many consistent cycles but comparatively less than AS1241 (Fig. 13.6b). Conversely, the bipartite graph for AS6453 included several cycles deemed as bogus (Fig. 13.6a): due to the nature of this particular network, it is likely many of these problematic cycles resulted from missing one or several convergence points, though they could also have resulted from undergrown subnets.



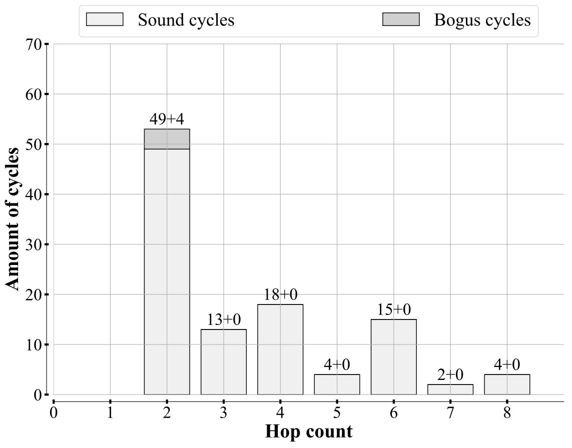
(a) AS6453 (TATA Com.), September 20th, 2020



(b) AS6939 (H.E. LLC), September 20th, 2020



(c) AS286 (KPN B.V.), September 20th, 2020



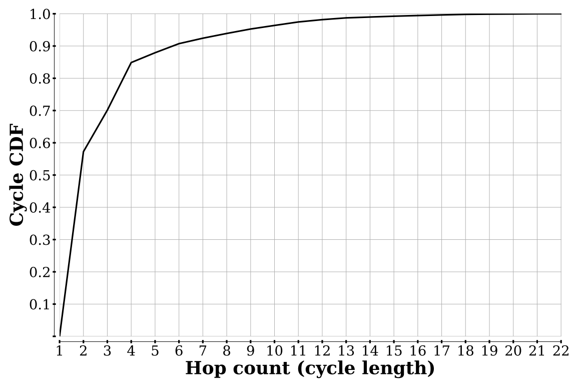
(d) AS1241 (Forthnet), September 12th, 2020

Figure 13.6: Topological cycles for various Autonomous Systems.

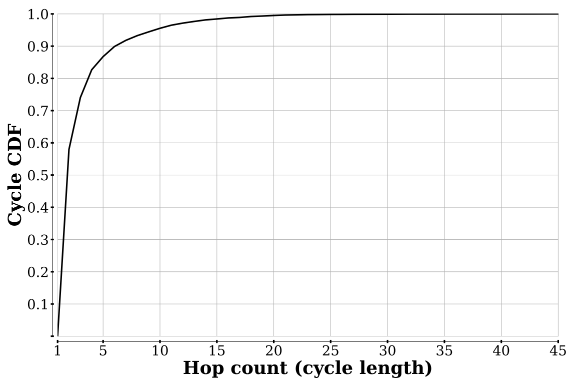
To better assess the frequency of back-up links, the topological cycles can be viewed at a larger scale, i.e., across several snapshots. Figure 13.7 shows, as CDFs, the distribution of all sound cycles found within two collections of snapshots captured by SAGE during the year 2020 (Fig. 13.7a) and during Spring 2021, i.e., from March to May included (Fig. 13.7b). Both Fig. 13.7a and Fig. 13.7b shows strikingly similar distributions: in both cases, slightly less than 60% of all sound cycles were back-up links, while slightly less than 90% of the same cycles were 5 hops long or less. This suggests, at the very least, that topological cycles did not result from capturing the snapshots from specific vantage points nor from targetting specific networks: a large share of 2020 snapshots were still collected from the PlanetLab testbed (mostly European nodes at the time), while all Spring 2021 snapshots were captured from the EdgeNet cluster (most EdgeNet nodes being located in the USA at the time). Moreover, the target ASes in 2021 differed from 2020 due to the addition of new target ASes but also a few considerable changes in the advertised IPv4 prefixes [13].

There is however one major difference between Fig. 13.7a and Fig. 13.7b: the largest hop counts

[13] E.g., AS286 now features larger advertised prefixes; as a result, its 2021 snapshots are much larger than those of 2020.



(a) Year 2020 (25,095 cycles in 345 snapshots)



(b) Spring 2021 (36,180 cycles in 371 snapshots)

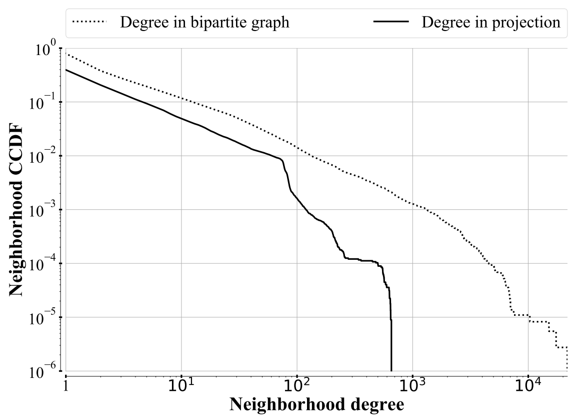
Figure 13.7: Distribution of structural cycles across large sets of snapshots.

drastically differ, with 2020's longest observed cycle being 22 hops long (found in a snapshot of AS3257) while 2021's longest cycle is 45 hops long (found in a snapshot of AS286). However, this latter cycle might be an artefact, as it was captured in its target AS only once, most topological cycles for the same target AS being included between 2 and 9 hops. Interestingly, in both 2020's CDF and 2021's CDF, the curve noticeably rises between 10 and 20 hops, suggesting long topological cycles are not that rare. In fact, long topological cycles may hint at more complex network constructions, such as load balancers, which may initially appear in SAGE data as diamonds [14]. A better characterization of topological cycles is left for future work.

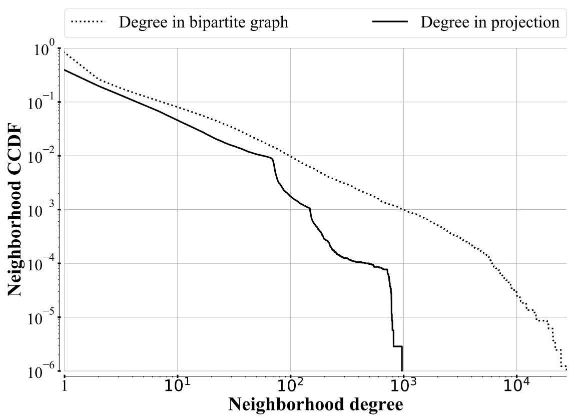
#### 13.3.2.3 Neighborhood degree in bipartite graphs and in ⊤-projections

Thanks to the mecanism of ⊤-projection, a neighborhood – subnet bipartite graph can be used to get a more accurate hop-level map of some target network by accounting for the new adjacencies revealed by the bipartite conversion itself (cf. Sec. 13.1). Comparing the neighborhood degrees in both types of graphs can be a way to reconcile old and new observations made on the router-level of the Internet (cf. Sec. 9.3.1): for reminders, an old but popular claim on the router-level states the router degree (i.e., how many routers a given router is adjacent to in the topology) naturally follows a power law shape [40]. However, this claim has been challenged multiple times [26, 28, 76, 78, 89], and IGMP probing (cf. Sec. 9.2) could be used to explain why power laws tend to appear: inference of Layer-2 equipment with `mrinfo` [90] was used to demonstrate the router degree might be overestimated because of the effects of Layer-2 devices (e.g., Ethernet switches).

Because a neighborhood abstracts one node at the hop-level, it can consist of either a single router or a full mesh with or without Layer-2 equipment. Moreover, neighborhoods are a tool for alias resolution (cf. Chapter 4), which means they can be used "as is", like in this section, or can be combined with alias resolution to study the router-level more precisely, as will be discussed in Sec. 13.4. If we stick to previous observations, the distribution of the neighborhood degree in the Internet should follow a power law, but such a property would not be mutually exclusive with a thorough study of the router-level that corroborates more recent research.



(a) Year 2020 (345 snapshots)



(b) Spring 2021 (371 snapshots)

Figure 13.8: Distribution of the neighborhood degree in bipartite graphs and their ⊤-projections.

Figure 13.8 plots the distribution of both definitions for the neighborhood degree, i.e., in bipartite graphs and their ⊤-projections (as performed by the NetworkX library [34]), respectively. To better view the power law property, both axes are in logarithmic scale, and the distribution is plotted as a **c**omplementary **c**umulative **d**istribution **f**unction (or CCDF). Much like Figure 13.6, Figure 13.8 accounts for large collections of snapshots to be more representative of the Internet as a whole, with Fig. 13.8a being based on 2020's snapshots and Fig. 13.8b resulting from Spring 2021's snapshots.

In both Fig. 13.8a and Fig. 13.8b, the neighborhood degree in bipartite graphs follows almost exactly a power law until a degree of $10^3$, while the neighborhood degree in ⊤-projections follows closely the former distribution until shortly before a degree of $10^2$. In both cases, upon diverging from the power law shape, the curves already account for more than 99% of the discovered neighborhoods. Such observations corroborate aforementioned claims on the Internet [40, 90] and further reinforce the idea that neighborhoods can be a credible tool for topology discovery, especially given that the datasets used to compute the distributions are quite different (as already argued in Sec. 13.3.2.2). Of course, Figure 13.8 also shows that the neighborhood inference as performed by `SAGE` (cf. Chapter 11) is still prone to produce artifacts: in both 2020 and 2021, some snapshots resulted in bipartite graphs where one neighborhood featured a degree higher than 10,000, which seems unrealistic. An in-depth look at the data led to the conclusion that such a neighborhood was a measurement artifact, being observed only once on the associated target AS across all snapshots. Various phenomena can explain such an outlier: large numbers of undergrown subnets, unusually large alias resolution scenarios that were processed in a too tolerant way [14] or inaccurate `traceroute` probing. Nevertheless, in the vast majority of the measurements, neighborhoods appear to provide a credible picture of the hop-level.

#### 13.3.2.4 Residual issues

While this section has presented many interesting results, measurement artifacts or incoherent observations have been mentioned multiple times, and show that, while `SAGE` is a quite accurate

[14] I.e., accepting alias pairs/lists during neighborhood inference with methods other than `Ally`-like/`iffinder`-like.

tool (as discussed in Chapter 12), more accurate subnet inference or convergence point detection would be needed to refine the observations made with the help of `INSIGHT`. In particular, overgrown subnets can be a problem to accurately infer the edges of a simple bipartite graph: overgrown subnets tend to replace multiple links at once, leading to unusually large subnet degrees. A large number of snapshots of AS224 captured during 2020 [15] are very good examples: in multiple snapshots, one large /21 subnet appears to have a large subnet degree, as if it was a key hub in the topology. However, a deeper look in the data shows the associated inferred subnet features very few interfaces, but many router interfaces detected via `traceroute` probing during various stages of `SAGE` fall into the address range of this subnet. The problem is actually quite simple: many interfaces in the address range do not reply to ICMP `echo-request` probes typically used during subnet inference (cf. Chapter 7), but do reply to `traceroute` probes when the TTL value expires. As a consequence, the /21 subnet can be ruled as a measurement artifact that might actually cover many smaller subnets (e.g., /30 subnets acting as point-to-point links). One way to fix this would consist of modifying the `WISE` methodology so that it also considers router interfaces exclusively observed via `traceroute` probing that are part of the initial target IPv4 prefixes.

Adjusting the target IPv4 prefixes provided to `SAGE` at start may be another way to improve the snapshots, and in particular, to reduce the amount of remote links (cf. Sec. 11.1.3). Indeed, the intermediate paths due to which `SAGE` resorts to remote links can sometimes include legitimate router interfaces that were just not included in the initial target prefixes, therefore not included in any inferred subnet/neighborhood. Including additional prefixes and re-running a measurement could "*fill in the blanks*", resulting in more complete snapshots. Interestingly, remote links are not uniformly distributed in DAGs and tend to be clustered around the same neighborhoods. In particular, in visual renders of bipartite graphs produced by `INSIGHT`, they typically appear as small cluster of green vertices [16]. Such observations may hint that sections of a network may be hidden, may be isolated and accessed through intermediate ASes or may consist of various sorts of tunnels.

Finally, accurate convergence point inference is another critical point. As discussed in Sec. 13.3.2.2, missing convergence points may exaggerate the number of topological cycles (as notably shown by Fig. 13.6a). Since previous or contemporary research provided algorithmical approachs for enumerating load balanced paths [14, 65, 122], `SAGE` may include some of these approachs to improve its peer discovery and subsequent peer aggregation steps which result in convergence point detection (cf. Chapter 11). Obviously, the accuracy of alias resolution is critical too: as mentioned in Sec. 13.3.2.3, running `SAGE` with a tolerant alias resolution (i.e., tolerating alias pairs/lists not produced by the `Ally`-like/`iffinder`-like methods) may result in unnecessarily large convergence points clustering too many components of the network at the same node in the graph. More generally, an accurate and exhaustive alias resolution scheme is capital for correctly inferring routers, which are critical for the tripartite model discussed in the next section.

Hopefully, all these issues arise from the measurement methodology and are not tied to the

[15]Cf. `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT/Results/AS224/2020`

[16]E.g. `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT/Results/AS6453/2020/09/`

formalism itself. On the contrary, the neighborhood – subnet model shows promising results for accurately assessing the importance of the subnet-level and more generally the network links it corresponds to. Moreover, the hop-level modeled by the neighborhoods and their adjacencies act as a first approximation of the router-level and may be used to study the internal routing of a target network without requiring exhaustive router-level discovery. The next section will nevertheless introduces one additional model designed to more accurately account for the router-level and to evaluate the likelihood of Layer-2 equipment being used.

## 13.4 A tripartite model: Layer-2 – Layer-3 – subnet

This section reviews the tripartite model first described in Sec. 13.1, i.e., the Layer-2 – Layer-3 – subnet tripartite formalism. First, Sec. 13.4.1 reviews how a `SAGE` snapshot can be turned into such a graph, and second, Sec. 13.4.2 discusses the router degree, how it compares with the neighborhood degree (bipartite definition) and how it can hint the presence of Layer-2 equipment.

### 13.4.1 Guidelines for building a tripartite graph

While the construction of a neighborhood – subnet simple bipartite graph is almost immediate (cf. Sec. 13.3.1), building a tripartite graph requires more care: indeed, in the final graph, only the subnets will remain unmodified with respect to the initial snapshot, while each neighborhood will be replaced by either one router, either several routers connected with (hypothetical) Layer-2 devices. Moreover, the routers resulting from this transformation needs to be connected with the right bordering subnets, based on the IP interfaces being part of their respective alias pair/list: e.g., given an alias list $\{A, B, C\}$, the associated router will be connected to a subnet $S_a$ if $S_a$ encompasses $A$. The combination of the underlying (inferred) routers and their associated subnets will be subsequently denoted as a ***post-neighborhood***. In fact, `INSIGHT` builds tripartite graphs by first converting the initial DAG vertices into post-neighborhoods, using the alias resolution data collected during the final step of `SAGE` (cf. Sec. 11.4), then by reading the neighborhood-based DAG to account for the edges that connect the initial neighborhoods together.

Hopefully, post-processing neighborhoods can be a trivial operation. If the final alias resolution step resulted in discovering exactly one alias pair/list for a given neighborhood, it can be modeled by a single router connected to all neighboring subnets due to having no hint for additional routers. Occasionally, there can be no alias pair/list at all: this notably occurs with best effort neighborhoods whose surrounding subnets feature no contra-pivot interface to consider for an alias resolution scenario (cf. Sec. 4.1.1). In this case, the neighborhood is modeled by a single ***hypothetical router*** as a best effort strategy. Like hypothetical subnets, hypothetical routers account for devices which should theoretically exist but which could not be detected during a measurement with `SAGE`.

When multiple alias pairs/lists are associated to a same neighborhood, multiple routers have to be created. For each alias pair/list, a new router can be created then associated with any neighboring subnet that encompasses one of its IP interfaces. Indeed, because alias pairs/lists typically include the contra-pivot interfaces of the neighboring subnets, the IPv4 ranges covered by each subnet can be relied on to identify which one the router provides access to. Of course, this is only possible if a subnet features one or several contra-pivot interfaces. This is why all neighboring subnets that cannot be mapped to any known alias pair/list (therefore any inferred router) should be mapped to one or several hypothetical router(s), again as a best effort strategy to ensure each subnet is eventually mapped to a router in the final graph. In the case of `INSIGHT`, a single hypothetical router is created at most for each post-neighborhood to minimize the total number of vertices in the final graph.

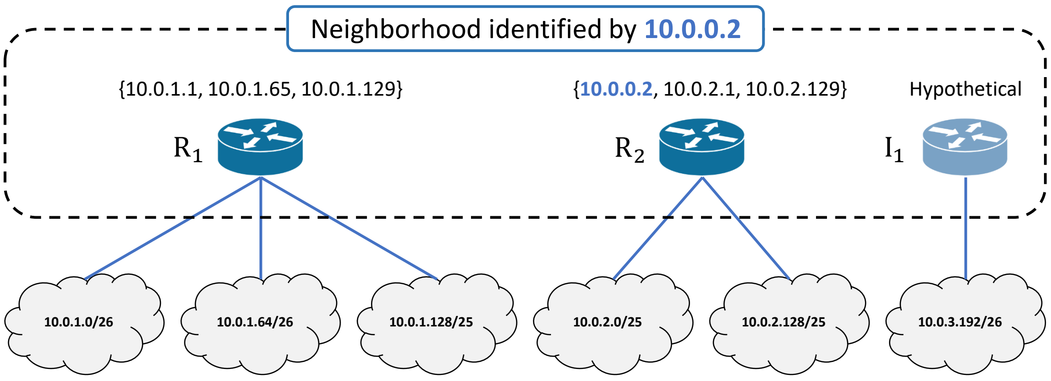


Figure 13.9: A toy example of post-neighborhood.

Figure 13.9 depicts a toy example of post-neighborhood. In the figure, the initial neighborhood is identified by `10.0.0.2` and bordered by six different subnets. Two alias lists are discovered on it, with the second including the router interface `10.0.0.2` itself. The first alias list is mapped to the three first subnets (all prefixed with `10.0.1`) while the second alias list is mapped to both subnets whose prefix address starts with `10.0.2`. Finally, because `10.0.3.192/26` does not encompass any aliased interface (because it does not feature any contra-pivot interface), said subnet is mapped to an hypothetical router $I_1$ ($I$ for I̲maginary) since it cannot be mapped to neither $R_1$ or $R_2$. While not shown in Figure 13.9, an hypothetical Layer-2 vertex will be eventually included in the final tripartite graph to glue $R_1$, $R_2$ and $I_1$ together to keep track of the fact they correspond to the same neighborhood/hop-level node in the topology.

Once all neighborhoods have been processed into post-neighborhoods, the tripartite graph can be finalized by integrating the adjacencies accounted for by the initial neighborhood-based DAG as additional router – subnet edges. The process can actually be very similar to the one used to build neighborhood – subnet bipartite graphs (cf. Sec. 13.3.1), and in particular, hypothetical subnets will also be used to model remote links and edges without a mapping. However, to do so, there is one specific requirement: establishing an ***entry router*** for each neighborhood. An entry router corresponds to the router whose alias pair/list includes the router interfaces that first identified the parent neighborhood: for instance, in Figure 13.9, $R_2$ would be the entry router. When a neighborhood is post-processed into a single router, it also becomes its entry router. In the case of a best effort neighborhood, i.e. which is not identified by observed router interface(s), the entry router is either an hypothetical router, either the first router listed for the associated post-neighborhood.

An entry router not only accounts for the router interfaces identifying the initial neighborhood, but also acts as a default solution when no router can be clearly identified to add a router – subnet edge corresponding to a neighborhood adjacency. Since it accounts for the IP-level by including router interfaces discovered by `traceroute` probing (if any are visible), the entry router is used to connect the post-neighborhood to any hypothetical subnet. Doing so, any adjacency found in the initial neighborhood-based DAG can be present in the final tripartite graph as well.

In practice, because all subnets are already connected to a router vertex as a result of generating post-neighborhoods, each type of edge in `SAGE` data (cf. Sec. 11.1.3) can be processed as follows.

- In the case of a **direct link**, an additional router – subnet edge can be added to the graph to connect the subnet associated to the link with the entry router of the second neighborhood. E.g., given a direct link `N2 → N3 via v.w.x.y/z`, an edge connecting `v.w.x.y/z` to the entry router of `N3` is inserted.

- For an **indirect link mapped to a subnet**, two router – subnet edges will be inserted, each one connecting the subnet with the respective entry router of each neighborhood. Contrary to simple bipartite conversion (cf. Sec. 13.3.1), there is no need to create a third edge since the edge between the subnet and a router of the neighborhood it borders already exists. E.g., given an indirect link `N2 → N3 via v.w.x.y/z (from N7)`, two router – subnet edges are inserted: one between `v.w.x.y/z` and the entry router of `N3` and a second one between `v.w.x.y/z` and the entry router of `N2`.

- The process is identical for an **indirect link without a mapping**, except that a new hypothetical subnet is created to replaced the missing subnet.

- The same goes for a **remote link**, except that the hypothetical subnet should be labelled so that it can be clearly distinguished from hypothetical subnets modeling one-hop adjacencies, just like with the simple bipartite formalism (cf. Sec. 13.3.1).

Once the adjacencies found in neighborhood-based DAG have all been inserted, following the aforementioned guidelines, the Layer-2 – Layer-3 – subnet tripartite graph is complete. If a visual render of it is required, however, all degree-1 components should be removed from the graph to begin with. Indeed, all subnets are accounted for during the post-neighborhood creation, but degree-1 subnets are useless in a visual render and can only make it unnecessarily difficult to read. For readability's sake, hypothetical Layer-2 components and hypothetical routers should also be colored differently: in `INSIGHT`, the former are colored in yellow and the latter in light blue.

Figure 13.10 provides a visual render of a tripartite graph based on a snapshot of AS13789, collected on April 10th, 2021 from the EdgeNet cluster. In this render, red nodes represent subnets, blue nodes account for routers and yellow nodes model (hypothetical) Layer-2 devices, with green

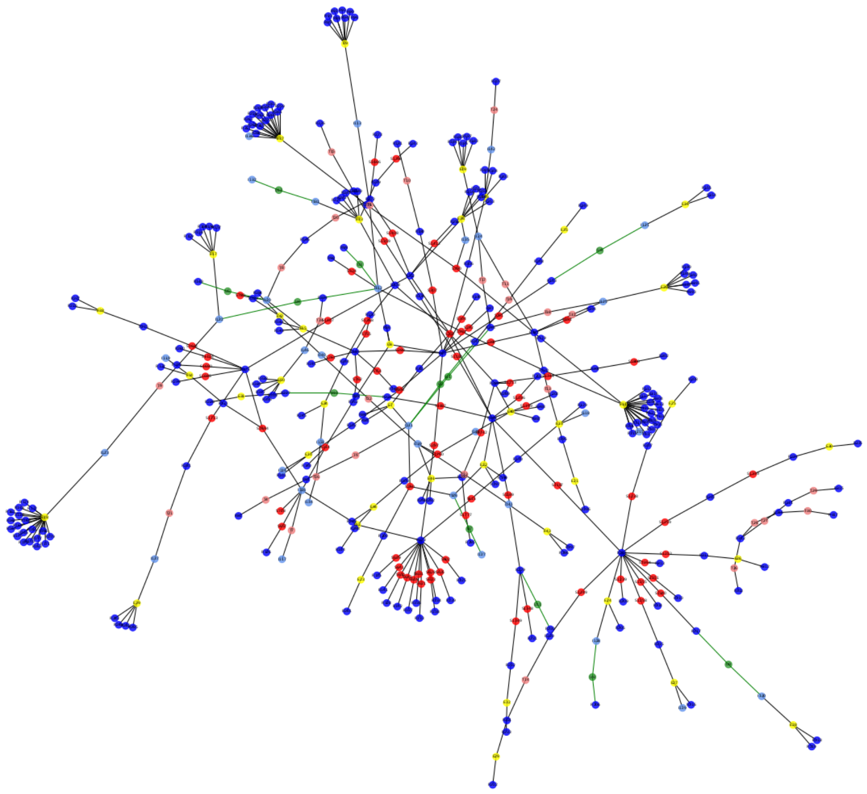


Figure 13.10: Render of AS13789 as a tripartite graph (April 10th, 2021; EdgeNet). The color code extends the one in Fig. 13.3, with light blue for hypothetical neighborhoods and yellow for Layer-2.

nodes representing multi-hop adjacencies. Naturally, Figure 13.10 only depicts the main connected components: a few small connected components are omitted, but can be viewed in the full render [17]. Note the visual similarities with Figure 13.3, but also the differences induced by the addition of the Layer-2/Layer-3 components, such as the small aggregates of routers connected with a same hypothetical Layer-2 component (aggregates of blue vertices with a single yellow vertex). Whether these aggregates hint actual Layer-2 components is discussed in the next section.

### 13.4.2 Router degree in the wild

As already discussed in Sec. 9.3.1, the router degree, i.e., how many routers a given router is adjacent to, has been studied and commented multiple times in the literature. In particular, many studies discussed the distribution of this degree in the Internet: early studies compared it to a power law [40], but ulterior research suggested the router degree may be overestimated because of other network components (Layer-2 equipment, in particular) [90]. However, as discussed in Sec. 13.3.2.3, the neighborhood degree does follow a power law shape while neighborhoods themselves can consist of either a single router or a mesh. As a consequence, the degree of the actual router(s) found within a neighborhood should always be equal or inferior to the degree of this neighborhood, and the distributions of both types of degree might hint the presence of Layer-2 equipment.

[17] `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT/Results/AS13789/2021/04/10/`

However, defining precisely the router degree in the context of a tripartite graph requires some nuance. While each neighborhood consisting of multiple routers will be translated in the graph as a virtual Layer-2 device connecting these routers by design, the existence of this device in practice is not necessarily likely. It is indeed possible, in some circumstances, that routers are directly connected together at the datalink layer without intermediate Layer-2 equipment. Moreover, because the total number of routers will be greater or equal to the total number of neighborhoods, a distribution of the router degree versus a distribution of the neighborhood degree may not reflect accurately the difference between both kinds of degree.

This is why this section will subsequently use four different definitions of the router degree. Let us first define the ***base router degree*** as the total number of adjacent routers a router can reach through its adjacent subnets in the tripartite graph. E.g., if a router has three adjacent subnets with degrees 2, 2, and 3 (as defined in Sec. 13.3.2.1), its base degree will be equal to 4 because four routers are adjacent via the three adjacent subnets [18]. With this base degree in mind, four different router degrees can be defined as follows.

- ***Experienced degree*** (`router-exp`)**:** denotes the degree of the encompassing neighborhood. It is equal to the sum of the base degrees of all routers it encompasses, or to put it another way, to the sum of the base degrees of all routers connected through the same virtual Layer-2 device in the tripartite graph. By definition, the experienced degree of all routers of a same neighborhood is identical.

- ***Minimal degree*** (`router-min`)**:** denotes the router degree if all virtual Layer-2 devices in the tripartite graph are considered as plausible. It is therefore equal to the base degree incremented once (to account for the additional connection to the Layer-2 equipment) if the encompassing neighborhood consists of multiple routers, and to the base degree if the neighborhood consists of only one router.

- ***Maximal degree*** (`router-max`)**:** denotes the router degree if all virtual Layer-2 devices in the tripartite graph are considered as implausible. I.e., all meshes are implemented by connecting each router to every other router in the neighborhood without intermediate equipment. It is therefore equal to the base degree plus the total number of routers found in the encompassing neighborhood minus one (to not account for the router itself).

- ***Conditional degree*** (`router-mix`)**:** denotes the router degree if all virtual Layer-2 devices in the tripartite graph are considered as plausible above a certain number of routers per neighborhood, and implausible otherwise. Given a threshold, it is therefore equivalent to the minimal degree if the total number of routers in the encompassing neighborhood is greater than the threshold, and equivalent to the maximal degree otherwise.

[18] For simplicity's sake, potential back-up links will be considered negligible at the scale of a snapshot.

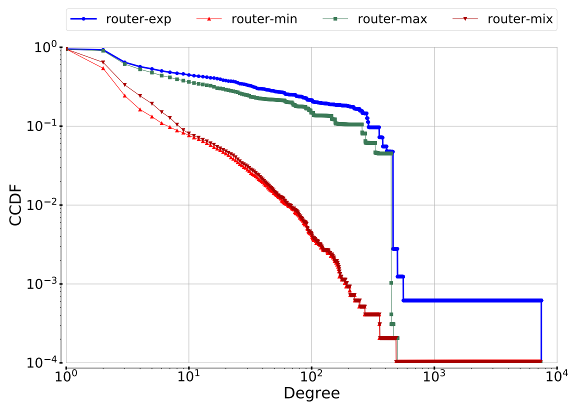
With these four definitions, the degree of each inferred router can be evaluated depending on its encompassing neighborhood and on the hypothetical presence of Layer-2 equipment. It should be noted, too, that the four distinct degrees should fulfill the inequality `router-min ≤ router-mix ≤ router-max ≤ router-exp`: at worst, if the neighborhood consists of a single router, all degrees will be strictly equal. Otherwise, the minimal degree should be inferior to the maximal degree, the conditional degree being equal to one of both depending on the total amount of routers in a same neighborhood and on the selected threshold. Finally, at worst, the maximal degree should be equal to the experienced degree: since the degree is incremented for each additional router in the neighborhood, the maximal degree is equal to the experienced degree if and only if all routers in the neighborhood have a base degree of 1.

As a toy example, let us review the degrees of Figure 13.9. Assuming each subnet has a degree of 2, the base degrees should be: 3 for $R_1$, 2 for $R_2$ and 1 for $I_1$. Their (respective) experienced degree should therefore be equal to 6. The minimal degrees are simply the base degrees but incremented once, therefore: 4 for $R_1$, 3 for $R_2$, and 2 for $I_1$. The maximal degree is obtained by summing 2 (total of routers in the neighborhood minus one) to the base degrees, resulting in: 5 for $R_1$, 4 for $R_2$, and 3 for $I_1$. Finally, the conditional degrees will be equal to the respective maximal degrees unless the threshold for considering Layer-2 equipment plausible is equal to 2, in which case the conditional degrees are equal to the minimal degrees.

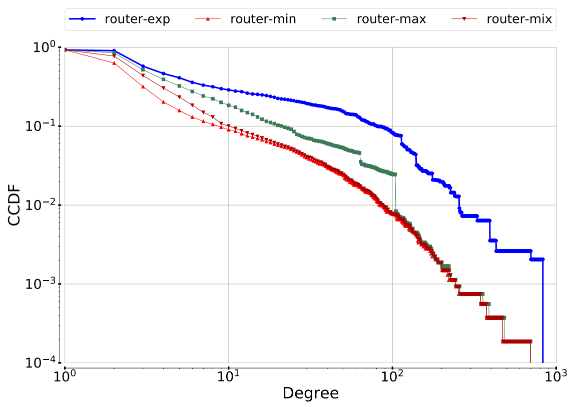
Figure 13.11 provides the distribution of all four different degrees based on the tripartite graphs generated from snapshots of four distinct networks: AS3257 (GTT Communications; Tier-1), AS6461 (Zayo Group; Tier-1), AS1273 (Vodafone; Transit) and AS224 (UNINETT; Stub). All snapshots were captured from the EdgeNet cluster in May 2021. Just like for figures discussed in Sec. 13.3.2, the distributions depicted in Figure 13.11 are representative and do not constitute outliers among snapshots captured for the same target networks, though they are some variations depending on the snapshot. Note also that the threshold used for the conditional degree is 8 inferred routers in Figure 13.11; i.e., Layer-2 equipment is considered plausible starting from 9 inferred routers. Finally, just like with Sec. 13.3.2, many more comparable figures (including for other Autonomous Systems) can be browsed online at the `SAGE` public GitHub repository [19].

Figure 13.11 shows, overall, that the experienced degree is always significantly higher than other definitions of the router degree. Without a surprise, the distribution of the minimal degree was always the lowest, as in all figures, 99% of the inferred routers had a minimal degree below 100. At the same time, it was not uncommon for the experienced degree of the same proportion of inferred routers to reach maxima of several hundreds: only the distribution for AS224 (Fig. 13.11d) went only slightly above one hundred. Interestingly, the conditional degree influenced mostly small degree routers, as it tended to equate the minimal degree for higher degrees. The maximal degree, on the other hand, could shift the degree close to the experienced degree, but this really depended on the measured target AS. In fact, AS3257 (Fig. 13.11a) and AS1273 (Fig. 13.11c) provided comparable distributions for

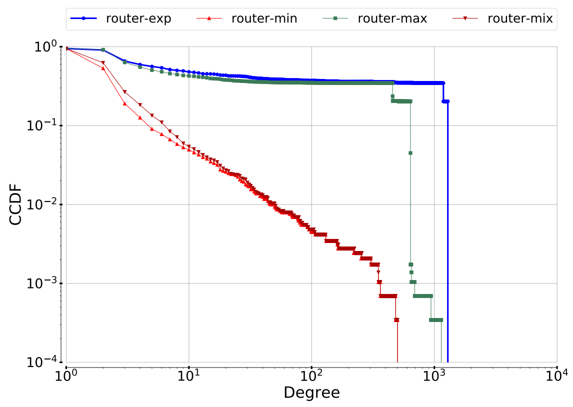
[19]Cf. `https://github.com/JefGrailet/SAGE/tree/master/Python/INSIGHT/Results/`.



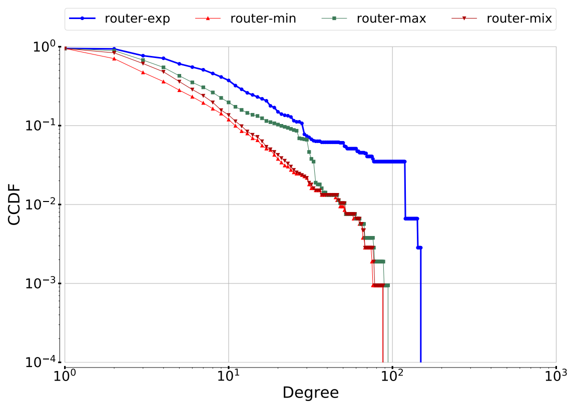
(a) AS3257 (GTT Com.), May 19th, 2021



(b) AS6461 (Zayo Group), May 15th, 2021



(c) AS1273 (Vodafone), May 15th, 2021



(d) AS224 (UNINETT), May 17th, 2021

Figure 13.11: Router degree distribution for various Autonomous Systems.

the maximal and experienced degrees. Similar observations could be drawn for AS6461 (Fig. 13.11b) and AS224 (Fig. 13.11d), though the former provided higher maxima, which should not be too surprising considering its Tier-1 nature. These differences must however be nuanced with the results of the alias resolution itself, which struggled with some difficult scenarios.

| **Metrics** | **AS3257** | **AS6461** | **AS1273** | **AS224** |
|---|---|---|---|---|
| #neighborhoods | 5,194 | 3,410 | 1,384 | 622 |
| #inferred routers | 11,038 | 5,783 | 3,159 | 1,178 |
| #hypothetical routers | 1,048 (9.5%) | 816 (14.1%) | 275 (8.7%) | 181 (15.4%) |
| #single routers | 4,093 (78.8%) | 2,462 (72.2%) | 1,069 (77.2%) | 387 (62.2%) |
| #single alias pair/list | 1,875 (36.1%) | 11,89 (34.9%) | 574 (41.5%) | 216 (34.7%) |
| Max #routers per neighborhood | 445 | 105 | 639 | 34 |
| Corresponding `router-exp` | 461 | 114 | 1,289 | 18 |
| Average (multiple routers) | 6.3 | 3.5 | 6.63 | 3.37 |

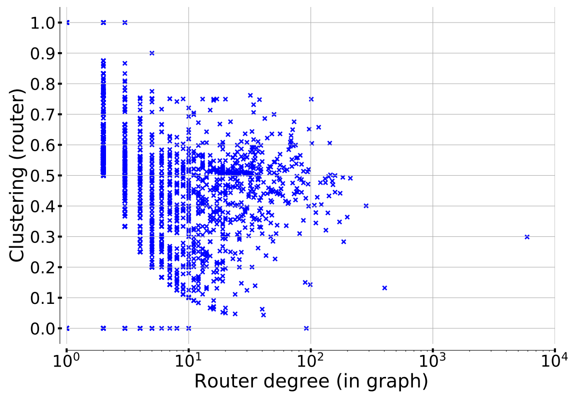
Table 13.2: Additional metrics for the snapshots of Figure 13.11.

Table 13.2 provides various metrics about the routers inferred for all snapshots mentioned in Figure 13.11. In particular, the first part of the table compares the total number of neighborhoods and inferred routers, with additional metrics quantifiying the number of hypothetical routers (cf. Sec. 13.4.1), the number of neighborhoods eventually replaced by a single router, and finally the number of neighborhoods whose single router corresponds to an actual alias pair/list. This last metric is provided notably because numerous best effort neighborhoods (regardless of the snapshot) results in a single or no IP interface to alias, both cases being replaced by a single router following the conventions of the tripartite conversion in `INSIGHT` (Cf. Sec. 13.4.1). The second part provides the largest aggregates of routers (i.e. maximum number of routers replacing a same neighborhood), the corresponding experienced degree and the average number of routers per neighborhood (not counting the neighborhoods replaced by single routers).

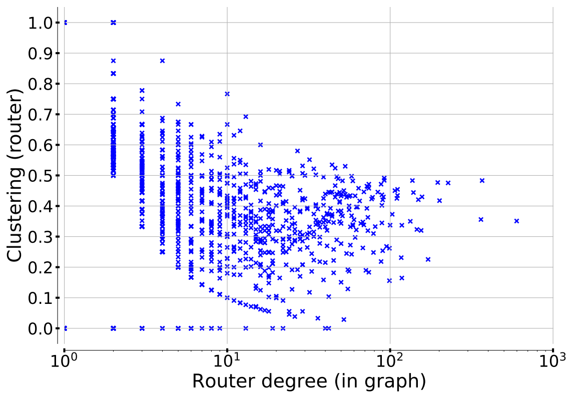
Table 13.2 shows that, in all snapshots, a majority of neighborhoods were eventually replaced by a single router in the tripartite graph, around half of them resulting from alias resolution results rather than a lack of identified alias candidates. However, the metrics in the second part of Table 13.2 suggests that some neighborhoods were replaced by a surprisingly large number of inferred routers. Interestingly, the neighborhoods these extreme values correspond to are not the highest degree neighborhoods in the selected snapshots, but rather cases where the method picked by the alias resolution framework first developed in Chapter 3 was not suitable. In all cases, the problem is related to IP-ID-based alias resolution. For reminders, this type of alias resolution consists of collecting sequences of IP identifier values (the IP identifier being a field in the IP header [102]) on each IP interface to alias and to compare them: if the sequences can be interleaved so that they are consistently increasing, or if their behaviours (e.g., speed, monotonicity) can be compared soundly, then the associated interfaces can be considered as belonging to the same devices. Indeed, a same router typically uses the same IP ID counter for all its interfaces. Several tools are based on this approach [16, 73, 113] (see also Sec. 2.3).

In the case of AS3257 and AS1273, the largest aggregates of inferred routers result from a large number of alias candidates being fingerprinted as compatible with the `Ally`-like method, but resolving simulatenously so many candidates proved to be too ambitious for the framework (first described in Chapter 3) as configured by default in `SAGE`. Using other alias resolution parameters and/or post-processing the alias resolution data provided by each snapshot can however lead to the discovery of several aliases, though the best approach to tackle these scenarios would be to create an improved data collection mechanism which is suitable for both small- and medium-sized alias resolution scenario (for reminders, the framework used by `SAGE` is tailored for small scenarios). Interestingly, there were several profiles of fingerprints observed in the initial neighborhoods replaced by these aggregates, hinting that were would have been several routers anyway – something which should not be too surprising considering the associated experienced degree.

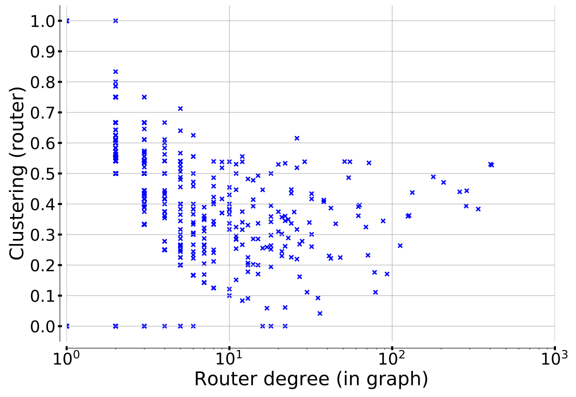
However, the largest aggregates of routers for both AS6461 and AS224, also encompassing router interfaces deemed as compatible with the `Ally`-like method, are likely the consequence of the alias resolution method itself being not suitable. In the case of AS6461, the interfaces involved in the largest aggregate all appear to have their own IP ID counter, which may suggest they are separate routers, or more simply, that the parent router simply implements a separate counter for each interface. The latter scenario actually holds true for some new routers of the ULiège network which were installed during the course of 2021 and which implement a different IP ID counter per interface, therefore impairing the `Ally`-like method provided by the framework from Chapter 3. When it comes to AS224, the problem is similar but results from a different issue: IP ID values emitted by the alias candidates appeared to be constant for a short period of time, and therefore could not be interpreted as increasing sequences. In both cases, the `Ally`-like method cannot be used "as is": an updated characterization of the IP ID behaviour, alternatives or additional probing would be needed to achieve sound discovery of alias pairs/lists in today's Internet.



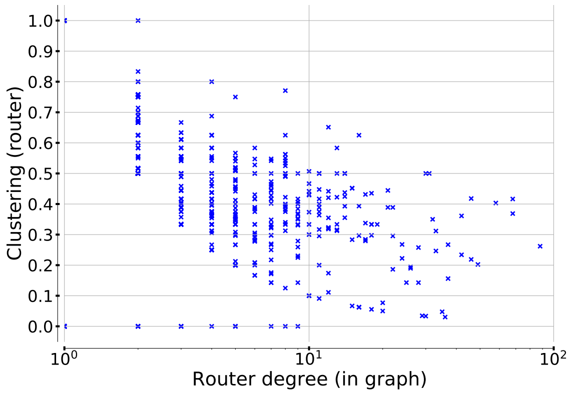
(a) AS3257 (GTT Com.), May 19th, 2021



(b) AS6461 (Zayo Group), May 15th, 2021



(c) AS1273 (Vodafone), May 15th, 2021



(d) AS224 (UNINETT), May 17th, 2021

Figure 13.12: Local density (clustering) of routers in function of their degree (`router-min`).

Hopefully, even considering large aggregates resulting from difficult alias resolution scenarios, the average number of inferred routers within neighborhoods turned into multiple routers remain reasonably small but high enough to suggest the presence of Layer-2 equipment within the largest neighborhoods, even though the distributions for AS3257 (Fig. 13.11a) and AS1273 (Fig. 13.11c) are skewed because of some aggregates of inferred routers increasing considerably the total number of inferred routers. In particular, the distributions of the minimal and maximal degree for AS6461 (Fig. 13.11b) shows clearly Layer-2 equipment would be advantageous for a small ratio of neighborhoods. Finally, it should be noted that the inability to use classical alias resolution methods in some cases led to the inference of a few very large alias lists on the basis of fingerprints alone: this is why the distributions in Fig. 13.11a are meeting by the bottom of the figure. Overall, these observations show the tripartite model is suitable for studying the router-level while considering the potential presence of Layer-2 equipment within a target network, though a more up-to-date and accurate alias resolution framework is needed to build and study such graphs thoroughly.

Finally, to complement the previous degree distributions, Figure 13.12 shows the clustering coefficients of the inferred routers found in the tripartite graphs generated for the same snapshots as before. Due to the construction of the tripartite graphs in `INSIGHT` (cf. 13.12), these coefficients are based by design on the minimal router degree, and naturally also includes the splitting of some large neighborhoods in many (small degree) inferred routers because of alias resolution issues. Nevertheless, the point clouds depicted by Figure 13.12 suggest, just like Figure 13.5, that the local density of the inferred routers was just as varied as the local density of their initial neighborhoods. In fact, it tended to be slightly higher, though this might be influenced by the existence of the hypothetical Layer-2 vertices used to build the tripartite graphs. Again, the local density of higher degree vertices did not seem to be synonymous of a high clustering coefficient: in fact, with the exception of AS1273 (Fig. 13.12c), there seemed to be considerably more inferred routers having a higher clustering coefficient than the highest degree vertex than in Figure 13.5, and this seemed especially true for both Tier-1 ASes, i.e. AS3257 (Fig. 13.12a) and AS6461 (Fig. 13.12b). Though this observation should be taken cautiously, this suggests that both these networks featured a more distributed topology, which may be correlated with their classification as Tier-1 ASes. A deepened analysis of the degree and the local density of routers in tripartite graphs is left for future work, along with an updated alias resolution framework solving the aforementioned issues [20].

## 13.5 Closing comments on network graph inference and modeling

This third and final part of this thesis provides three significant contributions.

1. A novel approach to take advantage of `traceroute` data was discussed and evaluated in Chapter 10, therefore answering research question 1 (from Sec. 9.4), which was about re-evaluating how `traceroute` paths should be interpreted for topology mapping. This novel approach takes advantage of existing subnet-level data to discover neighborhoods, which correspond to single nodes at the hop-level, and relies on the final hops of `traceroute` paths to locate such neighborhoods with respect to each other. Doing so, `traceroute` probing can be minimized and it becomes possible to build new maps of the Internet without suffering from typical `traceroute` issues which impair tools such as `TreeNET` [51, 55].

[20]For reminders, the framework used in `SAGE` (cf. Chapter 3) was first implemented in `TreeNET` around 2016-2017 [51].

2. To address research question 2, which consisted of designing a systematic topology mapping technique based on the aforementioned new interpretation of `traceroute` paths, a new topology discovery tool called `SAGE` [54] has been introduced and thoroughly described in Chapter 11. Re-using the `WISE` methodology [52, 53], `SAGE` collects subnet-level data then maps the network in the form of a neighborhood-based **d**irected **a**cyclic **g**raph (DAG). As such, it provides a systematic network mapping scheme based on the concepts elaborated in Chapter 10, while also addressing several challenges, such as the detection of convergence points (cf. Sec. 11.1.1). Measurements with `SAGE` on groundtruth networks and in the wild could also be analyzed to answer research question 3, on determining whether `SAGE` could capture accurate pictures of networks regardless of the vantage point, in Chapter 12. Doing so, it has been demonstrated that `SAGE` offers very accurate link discovery and is also capable of capturing comparable snapshots of a same target network regardless of the vantage point.

3. Finally, an answer to research question 4, on assessing the potential of bipartite formalisms for studying the Internet, was provided in this very chapter: two different modeling formalisms have been described in Sec. 13.1 to transform then analyze the data first captured by `SAGE`. A large collection of snapshots collected from both the PlanetLab testbed (early 2020) and the EdgeNet cluster (second half of 2020 and 2021) were transformed via a custom Python library, `INSIGHT`, to study topological properties of the measured networks. In particular, the neighborhood – subnet bipartite model turned out to be very promising: not only it is very easy to obtain from `SAGE` data, but it also allows to study the roles played by subnets in today's Internet, notably by assessing their degree (cf. Sec. 13.3.2.1) and finding and analyzing the topological cycles in which they appear (Sec. 13.3.2.2). Finally, an analysis of the neighborhood degree in the wild showed `SAGE` observations are consistent with previous claims regarding the structure of the Internet (Sec. 13.3.2.3).

Contribution 3 also encompasses some early observations involving the tripartite Layer-2 – Layer-3 – subnet model. More true to the actual components of the Internet than the neighborhood – subnet formalism, such a model could be re-used in the future to study the router-level accurately while assessing the potential presence of Layer-2 equipment. The current state of the research with this model is however partially impaired by alias resolution issues, in addition to the few problems already influencing simple bipartite graphs (cf. Sec. 13.3.2.4).

Nevertheless, the `SAGE` methodology has the potential to make bipartite and tripartite modeling of the Internet more accessible. For reminders, a bipartite model for the Internet has already been proposed in the past [114] but relied on data collected with IGMP probing [33], a topology discovery scheme which is now outdated (cf. Sec. 9.2). Since `SAGE` relies on much more common probing techniques (i.e., mostly ICMP [101] and `traceroute` [119] probing), its core ideas could be easily reused by the research community to study intra-domain topologies and how they use routers, subnets and Layer-2 equipment with the help of bipartite/tripartite modeling.

Of course, before considering these modeling opportunities, the SAGE methodology may also be improved first. For instance, the detection of convergence points could be refined, as graphs built by SAGE occasionally shows signs of missed convergence points (cf. Sec. 12.1.2 and Sec. 13.3.2.2). As already argued in Sec. 13.4.2, the alias resolution scheme needs an update to take account of new challenges but also to scale better with medium-sized alias resolution scenarios (i.e., several hundreds of IP interfaces to resolve). And while this chapter only discussed the comparison of several definitions of the router degree as a way to infer Layer-2 devices, other topological properties observed in the graphs could be exploited to refine the detection of these elusive devices.

Finally, the network graph inference methodology provided by SAGE constitutes another example of holistic topology discovery, as it connects the problem of mapping the hop- and router-levels of a target network with both alias resolution and subnet inference. On the one hand, alias resolution (via the framework from Chapter 3) is required to verify whether or not the peers of a given neighborhood, when it features several peers, correspond to the same neighborhood (see Sec. 11.1.1). Failing to take this possibility into account may lead to a graph featuring more vertices and edges than there are neighborhoods and links in the real network. On the other hand, the neighborhoods as discovered by SAGE rely on subnets first inferred with the WISE methodology, and the same subnets are later used to identify the network links (cf. Sec. 11.1.3), this latter use of subnet-level data being the first step to build the bipartite/tripartite models discussed in this chapter (cf. Sec. 13.1).

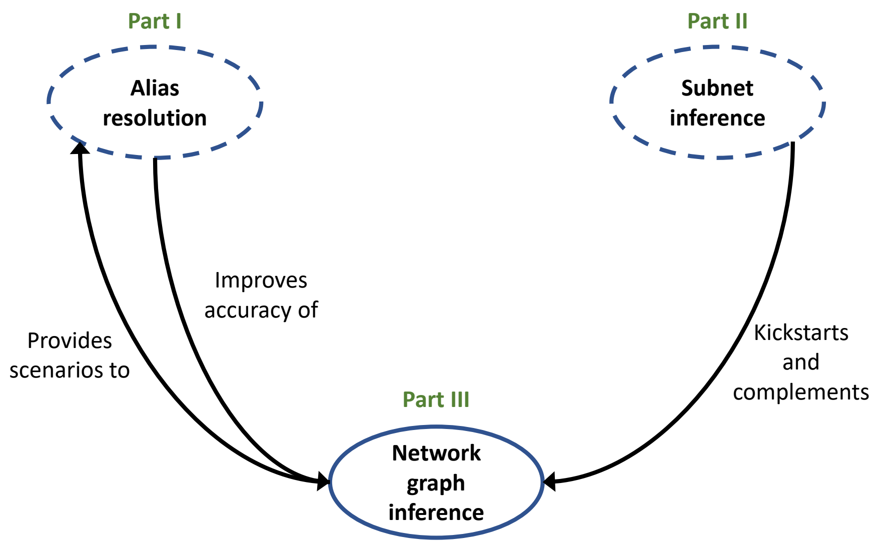


Figure 13.13: Interactions of the SAGE graph inference methodology in a holistic approach.

In fact, if we also consider that the WISE methodology embedded in SAGE to perform subnet inference also itself embeds alias resolution (again, via the framework presented in Chapter 3), then SAGE is the most complete example of holistic topology discovery described in this thesis. Figure 13.13 summarizes the interactions between network graph inference and other topology discovery challenges as presented in this third part of the thesis, that is, without the interactions between alias resolution and subnet inference that the WISE methodology takes advantage of.

# Chapter 14

# Conclusion

The Internet is a vast and complex network of computers that makes possible the transmission of large amounts of data over the entire globe in a matter of (milli)seconds. Such a technological achievement was made possible not only by the packet switch paradigm [15, 63, 75], but also by the open network paradigm elaborated by Kahn in 1972 [69], which led to the creation of the first versions of the IP [102] and TCP [103] protocols in 1974 [25]. Thanks to their design, new computers can join the Internet and exchange packets without any preliminary knowledge of its architecture. This also ensures the Internet cannot be controlled in its entirety by a central authority.

However, this design also implies that the complete and exact architecture of the Internet remains elusive. At the highest level, the Internet works as a large number of **A**utomonous **S**ystems (ASes) which are, in practice, individual networks each operated by the same organization or company, sometimes over a specific region of the globe. Typically, each major national ISP possesses its own AS (e.g., Proximus, a major Belgian ISP, operates AS5432). Each AS is nevertheless capable of forwarding packets to any part of the globe by sending them to one of its (direct) neighboring ASes, each AS advertising the routes it provides to its neighbors with the help of the BGP protocol [79].

As a consequence, while a network operator may know all the technical and architectural details of its own network, the same operator may have no idea of what the architecture of the neighboring ASes consists of, even though client/provider relationships between ASes may hint at what type of architecture might be used. While this is certainly an advantage in terms of security and flexibility, this has led to some major issues with the Internet in the past: for instance, in June 2015, a Malaysian AS (Telekom Malaysia – AS4788) advertised a very large number of routes to its neighbor, a major Tier-1 AS (Level3 – AS3549), which induced the latter into forwarding a large amount of its traffic towards the former. Because the Malaysian AS did not have the architecture to handle such a large amount of traffic, the entire Internet slowed down [115].

The fact that the full architecture of the Internet cannot be known led the scientific community into creating an entire research topic known as Internet topology discovery. Topology discovery aims not only at examining and preventing issues such as the global slowdown of June 2015 [115], but also at capturing realistic data on various networks to develop new technologies (such as new communication protocols) or to re-assess existing ones. To do so, the research community developped multiple methods to study the Internet from various perspectives, among which three have been discussed in this thesis:

1. the **router-level**, i.e., how individual routers (the computers designed to route packets) are interconnected,

2. the **subnet-level**, i.e., how the address prefixes managed by a same network are organized in subnets (short for subnetworks), which are sets of computers that can exchange packets with each other without any intermediate router,

3. the **hop-level**, i.e., which accounts for how packets are transmitted from one location to another in respect of the forwarding observed via (Paris) `traceroute` [11, 119], each location consisting of a single router or of a mesh of routers, as the modern Internet also includes Layer-2 devices that connect individual routers into meshes that behave as single hops in respect of packet forwarding [90].

One of the initial purposes of this thesis was to advance the state-of-the-art of topology discovery in respect of all three aforementioned levels. In particular, each level has been studied through the lens of a specific sub-domain of topology discovery which is already well documented in the literature. The discovery of the router-level was addressed by re-evaluating well known alias resolution methods and by creating a framework which takes advantage of IP fingerprinting [121] to evaluate which methods are the most suitable to a given scenario, each scenario being (ideally) determined through space search reduction [51]. The problem of subnet inference (i.e., mapping the subnet-level) was extensively studied by re-evaluating the traditional hypotheses of this domain, which resulted in the creation of the `WISE` (**W**ide and l**I**near **S**ubnet inferenc**E**) subnet inference tool [52, 53]. Last but not least, the discovery of the hop-level was carefully elaborated by reviewing how network paths discovered via `traceroute` probing [119] could be best used for inferring the hop-level of a target network, which motivated the design of the topology discovery tool `SAGE` (**S**ubnet **AG**gr**E**gation) [54].

Despite the aforementioned research topics appearing to have little connection with each other, this thesis aimed at building bridges between them, notably by identifying the challenges they have in common. In particular, the new solutions designed to discover the hop- and subnet-levels were elaborated with load balancing in mind. In the modern Internet, many networks resort to implementing complex architectures featuring multiple paths to handle increasingly large amounts of traffic, rather than increasing the capacity of a single path, the former approach being usually more economic. However, due to these load balancing constructions, network paths can vary considerably, even over short timespans [12, 82].

Therefore, a first major contribution of this thesis is to advance the state-of-the-art of **load balancing aware topology discovery** (a research direction first mentioned in Sec. 1.1.2). The detection of the convergence points was notably discussed, a convergence point being a router where multiple (possibly load balanced) network paths rejoin and which can appear as multiple differing router interfaces in `traceroute` paths. Correctly detecting these convergence points with the help of alias resolution is critical to achieve both consistent subnet inference and accurate hop-level mapping of a target network. Moreover, this thesis also frequently discussed the issue of observing `traceroute` paths of varying length towards the same destination, which can constitute a challenge for both aforementioned topics. Therefore, both `WISE` and `SAGE` implemented algorithmical solutions to be able to achieve their respective tasks while dealing with the effects of load balancing. Measurements in the wild with both `WISE` and `SAGE` demonstrated that they were capable of discovering very similar data on the same target network regardless of the date (over the span of weeks) and the vantage point, while their respective validation on two groundtruth networks showed they could capture very accurate pictures of these networks [52–54].

This thesis has also extensively explored the topic of **subnet-based topology discovery**. Initially, the research community considered the discovery of subnets to be a way to complement router-level maps [58, 116, 118], but did not see subnets as a new tool to investigate the Internet, or at least not before the introduction of `TreeNET` [55]. As discussed many times in this thesis, subnets can be a powerful tool for topology discovery: in the context of alias resolution, aggregates of subnets located one hop away from each other, i.e., neighborhoods, can be used to compute sets of alias candidates that can only be aliased with each other. This form of space search reduction was used to complement the alias resolution framework elaborated for this thesis, resulting in efficient alias resolution [51]. Research involving `WISE` was later used to thoroughly assess the concept of neighborhood and to discuss its potential for building network graphs [53], which was eventually concretized with `SAGE` [54]. `SAGE` not only builds these graphs but also maps the links they contain to the discovered subnets, which not only complements the initial data but also opens **new modeling opportunities**. In particular, a neighborhood – subnet bipartite formalism was proposed, and the potential of such a formalism was commented on with the help of data captured by `SAGE` and subsequently processed into bipartite graphs [54], therefore contributing to the research direction announced in Sec. 1.1.3. Overall, this thesis implicated subnets in all of its contributions.

More broadly speaking, as first announced in Sec. 1.1.1, this thesis has all along elaborated applications of **holistic topology discovery**. Instead of considering all topics of topology discovery as separate problems with dedicated solutions, all new tools introduced by this thesis, namely `WISE` [52, 53] and `SAGE` [54], take account of the relationships that exist between them to achieve their respective purpose. Though the idea of combining several topology discovery methods is not entirely new, as it has been discussed several times in the context of alias resolution [74, 87, 113, 117], many works from the state-of-the-art focus exclusively on specific tasks or rely on unique (and sometimes obsolete) probing strategies. This holds true for all topics discussed in this thesis: alias resolution [16, 73, 86], subnet inference [116, 118] and network graph mapping [90, 91, 97, 111, 112]. In this thesis, it was demonstrated that subnet inference can benefit from alias resolution and vice versa [51–53], that subnet inference can not only kickstart but also complement network graph inference [53, 54], and that network graph inference can benefit from alias resolution but also induce new alias resolution scenarios in turn [51, 54].

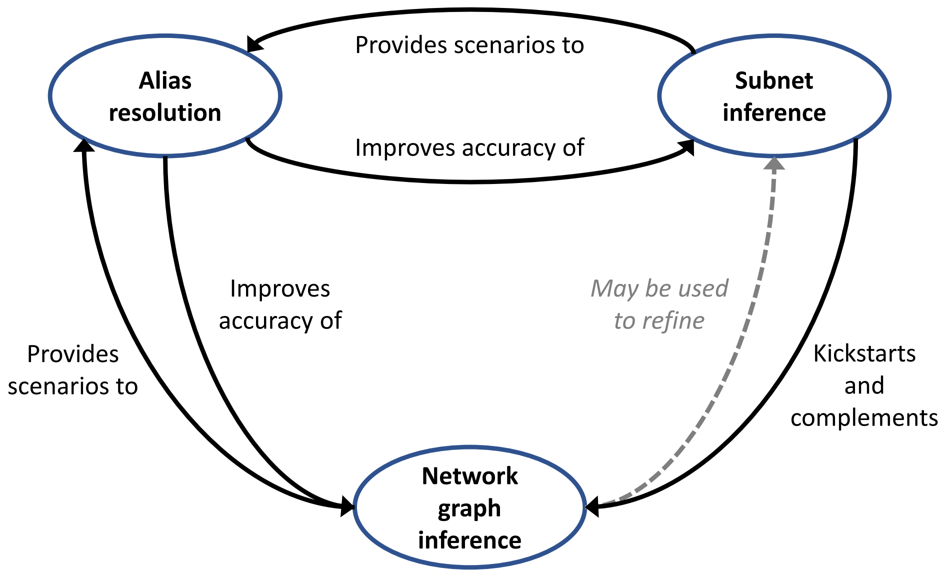


Figure 14.1: A holistic approach to topology discovery.

In fact, there is only one interaction between the three main topics of this thesis that has not been explored: using the results of network graph inference to refine subnet inference. This specific interaction has been left for potential future work (cf. Sec. 14.1), mostly because it would require adding some kind of additional post-processing step in a tool like SAGE, which already features a complex workflow. Otherwise, the current implementation of SAGE features almost all interactions between the three main topics discussed throughout this document. These interactions are summarized in Figure 14.1, where the dashed arrow symbolizes the sole unexplored interaction.

Last but not least, this thesis has also put a lot of emphasis on **designing efficient topology mapping solutions**, notably by carefully designing their methodology. In particular, the first implementation of the fingerprint-based alias resolution framework in TreeNET is significantly cheaper and faster than MIDAR [73] (a well-known state-of-the-art tool) when it comes to alias resolution, and remains cheaper even when the space search reduction is taken into account [51]. The WISE methodology, which separates the subnet inference itself from the data collection mechanisms that allow it, outperforms the subnet inference approach of TreeNET [55], which itself involves ExploreNET [118], in respect of both accuracy and total runtime [52, 53]. Finally, the deployment of both WISE and SAGE demonstrated both tools were capable of capturing complete snapshots of large networks (e.g., Tier-1 ASes covering one or more millions of attributable IP addresses) fast enough to reproduce the measurements on a quasi-daily basis. In particular, these regular measurements allowed to assess their ability to capture similar data regardless of the vantage point [52–54]. While other contemporary works have explored fast `traceroute` probing at the scale of the Internet [18, 65, 122], `WISE` and `SAGE` requires more thorough exploration of the IPv4 prefixes of a target network to perform multiple tasks (subnet inference, hop-level mapping, alias resolution) and are, at the time of writing, some of the most exhaustive topology discovery tools [52–54].

Overall, the defining characteristic of this thesis is how it approached its subject: rather than being a continuation of the state-of-the-art, it considered each of the three main topics under a new perspective. The topic of alias resolution was not tackled by finding other network protocol loopholes to create new methods, but by elaborating a generic methodology which is able to re-use state-of-the-art methods as if they constituted a toolbox [51]. Subnet inference was refreshed with `WISE` [52, 53] by decoupling the probing from the inference, which allowed for a better characterization and handling of its various challenges. Finally, the network graph inference methodology offered by `SAGE` [54] constitute a completely new, holistic approach to the problem, taking advantage of alias resolution and subnet inference to discover accurate snapshots of the hop- and router-levels of a target network. In summary, **envisioning new ways to look at a well-known problem should always be an option**, even if a whole literature already covers it.

## 14.1 Future work and research directions

Though most of the research work described in this thesis has been conceived to be as rigorous as possible, there are several research directions that could be further explored to improve or refresh the topology discovery methods that were previously presented.

First of all, the fingerprint-based framework needs to be properly updated in at least two ways. As already discussed at the end of Sec. 13.4.2, the framework as it is currently implemented in `SAGE` [48] struggles with larger scenarios of alias resolution, i.e., when there are several hundreds of alias candidates at the same time. A better scheduling should therefore be designed to tackle these scenarios, therefore ensuring the framework can handle both small- and medium-scale problems. One way to improve the scheduling could consist of the following: the initial scenario could be split in several sub-scenarios, and the alias pairs/lists discovered in these sub-scenarios could be merged by re-resolving a few of their respective IP addresses. The alias pairs/lists obtained during the latter resolution could then be used to merge the pairs/lists discovered via the sub-scenarios thanks to alias transitivity, i.e., the idea that for three IP interfaces $A$, $B$ and $C$, if $A$ and $B$ are alias of each other and if $B$ and $C$ are alias too, then $A$ and $C$ are alias of each other as well (cf. Sec. 2.2).

The second way to update the framework would simply consist of re-assessing the provided alias resolution methods. As the framework was, for the most part, designed around 2016 [51], its methods may not be as relevant in 2021's Internet. In particular, it became more and more apparent over time that some routers could no longer be inferred through alias resolution methods based on the IP identification field of the IP protocol [102]. As a reminder, these methods are essentially based on the idea that a router will generate IP identifiers (16-bit positive integers) with the help of a counter shared by all its interfaces (16-bit positive integer, incremented at each new value), which can then be probed each to verify if the IP identifiers appear to have been generated by the same counter [16, 73, 113]. However, more recent devices sometimes implement a separate counter for each of their interfaces rather than using the same counter for all of them. In fact, such routers are now found within the ULiège network. Therefore, a properly updated framework should plan heuristics to detect these difficult scenarios in order to test other methods when possible. The methods based on the IP identification field are not the only ones that need an update: for example, the DNS-based method could take advantage of regexes generated with the help of machine learning to spare the effort of tuning the regexes depending on the target network [80].

One last possible improvement for the fingerprint-based alias resolution framework would simply consist of updating the fingerprints themselves. Other properties tied to router interfaces could be explored, and new values could be used for the IP identifier counter type, i.e., a value provided by each fingerprint that characterizes the behaviour of the IP identifiers generated by the associated interface (cf. Sec. 3.1.2). For example, it has been observed, from time to time, that some router interfaces appear to generate a fixed IP identifier over a short timespan instead of generating an ascending sequence. Updated fingerprints could account for this behaviour, and could be subsequently used to re-assess the permeability of the Internet to alias resolution methods based on IP identifiers.

When it comes to subnet inference and, more broadly speaking, subnet-based topology discovery, a major research direction that has not been explored (or barely) by this thesis lies in comparing and merging measurements from multiple vantage points to obtain a more exhaustive picture of the target network. For example, the subnets of each snapshot of a target network could be compared in order to only keep the soundest results overall. Likewise, the neighborhood-based DAGs as built by `SAGE` could be compared so that the unique links observed in each snapshot could be inserted in a final, more complete graph, which would be more true to the target network than each individual snapshot (as hinted at in Sec. 12.2.2).

There are two reasons why this possibility has not been explored yet: one the one hand, a major part of the assessment of both `WISE` and `SAGE` consisted of finding similarities between snapshots collected from separate vantage points [52–54], and on the other hand, both tools have been designed with the PlanetLab testbed in mind [8]. Because of the number of working PlanetLab nodes decreased as the research work of this thesis progressed, measurement campaigns with either `WISE` or `SAGE` had to make the most of the available nodes. In hindsight, the vantage point rotation scheme could have been taken advantage of for this purpose, but it would not have provided as much vantage points as mentioned in previous works involving multiple monitors. For instance, the Cheleby system, used for Internet-scale subnet inference in 2012, involved several hundreds of PlanetLab nodes at once [70], while only one or two dozens could be used at the time of deploying `WISE` (i.e., Fall 2018).

The fact that `WISE`/`SAGE` are no longer constrained by the PlanetLab testbed could also motivate some more practical improvements over the present work. [1] Indeed, with the EdgeNet cluster [108],

[1] Both `WISE` and `SAGE` are 32-bit C/C++ applications whose source code is available for free on GitHub [48, 49].

which notably offers containerization via Docker [3], the algorithms of `WISE` and `SAGE` could be easily implemented and deployed with other languages or strategies. For example, they could be implemented as several programs: the active probing steps of `WISE` may constitute a stand-alone software separated from the subnet inference itself, since this latter step is entirely offline. While this strategy would probably have proven cumbersome with the PlanetLab testbed, it may be more convenient on the EdgeNet cluster or comparable systems.

Chunking the `SAGE` methodology into several pieces of software, potentially interacting with each other to stay true to the concept of holistic topology discovery, might also be useful to take advantage of network graph inference to improve the discovered subnets, an opportunity which is hinted at by Figure 14.1. A possible heuristic to achieve this would consist of detecting patterns among the subnets bordering a given neighborhood, notably based on the distribution of the prefix length of the bordering subnets. For example, if a large but dubious `/24` subnet is found around a neighborhood which is otherwise only bordered by `/30` subnets, this suspicious subnet may be split into `/30` subnets in order to be consistent with the other inferred subnets.

Another unexplored research direction lies in applying the methods described in this thesis to IPv6. As a reminder, both `WISE` and `SAGE` are designed with IPv4 in mind, mostly because IPv6 subnets are vastly larger than in IPv4. While subnet inference tools can easily exhaustively explore the address space of a single subnet in IPv4, the cost of doing the same task in IPv6 is prohibitive. Hopefully, the `WISE` methodology might be more suitable to inspire an IPv6 equivalent than previous work [116, 118], because the only hurdle would be, in theory, the detection of responsive IP addresses within IPv6 prefixes, a topic which is currently being investigated by the research community [19, 106]. In practice, the vastly larger IPv6 subnets might violate some of the assumptions that justify the linear complexity of some `WISE` algorithms (cf. Sec. A.2), but algorithmical adaptations could solve this hypothetical issue. Finally, the algorithmical steps that are proper to `SAGE` (i.e., regarding network graph inference) can, in theory, be re-used *as is* in IPv6. In other words, once a practical solution for IPv6 subnet inference is available, all the tools described in this thesis could be updated for IPv6.

Finally, it goes without saying that there is plenty of work to be done with the bipartite and tripartite formalisms. The formalisms described in Chapter 13 are merely opening doors for more research: research with `INSIGHT` essentially showed what types of topological properties could be studied, but these properties need to be better characterized. For instance, topological cycles, i.e., cycles that are not experienced at the time of probing but appear within bipartite graphs, could be used to quantify back-up links but also to detect and characterize load balancers. If load balancers were to be studied more precisely, `SAGE` may also be updated to incorporate some of the ideas elaborated earlier by Augustin et al. to enumerate load balanced paths while conducting (Paris) `traceroute` probing [14]. Last but not least, the tripartite model may be used in the future to study in detail the relationships between routers, subnets, and hypothetical Layer-2 vertices, and in particular, how much the router-level differs from the hop-level accounted for by the neighborhood-based DAGs built by `SAGE` and the associated neighborhood – subnet bipartite graphs.

# Appendix A

# Addenda on algorithms

This appendix provides additional content for all the unique algorithms that were presented in the main body of this document. In particular, the time complexity of each is analyzed in details. Moreover, the practical effects of specific algorithms that have not been evaluated in the main body of this thesis are quantified with the help of empirical data, either to further support the time complexity analyses or to provide insight on them. This side chapter is organized as follows: Sec. A.1 reviews the alias resolution framework from Chapter 3, Sec. A.2 comments on each algorithmic step of `WISE` [52, 53] (cf. Chapter 7), and Sec. A.3 provides the same commentary for each algorithmic (sub-)step of `SAGE` [54] (cf. Chapter 11) that is not already in `WISE`.

## A.1 Addenda on the alias resolution framework

This section analyzes the complexity of the alias resolution framework described in Chapter 3. Given that the actual alias resolution methods (cf. Sec. 3.3) are all based on the state-of-the-art (cf. Chapter 2), this section will only focus on the probing scheme (cf. Sec. 3.2). First, Sec. A.1.1 analyzes the time complexity of the main algorithm, and second, Sec. A.1.2 discusses the impact of the correction heuristic (Algorithm 2 from Sec. 3.2.3) on this complexity using empirical data.

### A.1.1 Time complexity of the probing scheme

The total number of alias candidates can be first expressed as $N$. The probing scheme formalized in Algorithm 1 from Sec. 3.2.3 should have a time complexity of strictly $\mathcal{O}(N)$ without the correction heuristic, and should stay very close to $\mathcal{O}(N)$ if the heuristic is used, as will be discussed in Sec. A.1.2.

As already discussed in the beginning of Sec. 3.2.2, only a small but constant number of probes will be sent to each of the $N$ alias candidates. The part of Algorithm 1 that collects IP IDs for all $N$

candidates should be proportional at worst to $\mathcal{O}(N)$ as well. Each wave of probes will send a single ICMP probe towards each of the $N$ candidates and store the resulting tuples (no tuple being produced for unresponsive targets). If using suitable programming techniques or paradigms (in this case, the object-oriented paradigm), storing the tuples can be done in constant time regardless of the target candidate ($\mathcal{O}(1)$). Of course, as the algorithm should collect $k$ IP IDs per alias candidate, there will be a total of $k$ waves. Nevertheless, $k$ can be considered negligible: it should be recommended to try to avoid probing each alias candidate for IP IDs more than necessary, both to avoid triggering security mechanisms at the side of the target domain and to reduce the overall probing cost. Typical values of $k$ would be below 10, starting ideally at four IP IDs per alias candidate (cf. Sec. 3.1.2).

### A.1.2 Impact of the correction heuristic

If the total number of alias candidates is still expressed as $N$, the correction heuristic detailed by Algorithm 2 in Sec. 3.2.3 should have a worst case scenario of $\mathcal{O}(N^2)$ due to the two nested loops, with the main loop processing up to $N$ tuples (no tuple is recorded for unresponsive candidates) and the nested loop processing at worst an number of tuples proportional to the current iteration. However, this worst case is highly unlikely: this would require all IP IDs to come in the order opposite to the tokens, in addition to having no *echoed* IP IDs at all. In practice, the nested loop will only start running for a small subset of pairs, and should only result in small displacements of a few tokens. The heuristic as a whole is therefore expected to have a negligible effect on the overall time complexity of the probing scheme (cf. Sec. A.1.1).

To assess this, five measurements with SAGE [54] were run from a server located at the Montefiore Institute [1] in April 2021. SAGE is a topology discovery tool developed in the context of this thesis (and presented in Chapter 11) that embeds an implementation of the alias resolution framework presented in Chapter 3 and evaluated in Chapter 4. The five target networks of the measurements were the network of the ULiège itself and four autonomous systems (or ASes) of varying sizes: AS12956 (Telefonica Global Solutions, Tier-1), AS13789 (Internap Holding, Stub), AS224 (UNINETT, Stub), AS6453 (TATA Communications, Tier-1). While these measurements were performed, SAGE was set in a *slightly verbose* mode, i.e., a mode in which the tool prints more details about its algorithmic steps in the console output. These additional details include the number of token reorderings induced by the correction heuristic. Using some Python scripting to process the console output, it became possible to quantify the frequency of the token reorderings and how many tokens, on average, were rearranged when reordering occurred.

| **Metrics** | **ULiège** | **AS12956** | **AS13789** | **AS224** | **AS6453** |
|---|---|---|---|---|---|
| Scenarios with token reorderings | 4.84% | 7.58% | 2.23% | 3.93% | 8.06% |
| Mean number of reordered tokens | 0.6666 | 8.44 | 0.6875 | 3.6634 | 1.0287 |
| Mean number of alias candidates | 15.6666 | 38.1733 | 30.5 | 8.4231 | 18.6781 |

Table A.1: Quantifying the correction heuristic by measuring five target networks with SAGE.

[1] planetlab1.montefiore.ulg.ac.be, a.k.a. *Montefiore-Fry*.

Table A.1 shows the results of the heuristic assessment for the five measurements. The first metric quantifies how many alias resolution scenarios evaluated by SAGE had to reorder tokens in order to mitigate the effects of random network delays. The second and third metrics quantify, among the scenarios where reorderings occurred, how many tokens where reordered per wave and how many alias candidates were involved on average. While there were a sizeable number of scenarios where reorderings occurred, with up to 8% while probing AS6453, the number of tokens being reordered appeared to be negligible or small with respect to the total number of alias candidates (since there will be as many token/ID pairs as there are candidates). This was especially striking with AS6453, which featured the highest ratio of scenarios with reorderings, yet only had one rearranged token in a wave targeting 18 alias candidates on average. The worst case in Table A.1 appeared to be the measurement of AS12956, where approximately 8 tokens were rearranged while probing an average of 38 candidates, i.e., 21% of the token/ID pairs in slightly fewer than 8% of all alias resolution scenarios. This shows that token reordering was only occasionnally necessary, even when network delays were experienced on a regular basis while probing.

As a consequence, the practical time complexity of the heuristic presented at Algorithm 2 in Sec. 3.2.3 should be very close to $\mathcal{O}(N)$ despite a worst case of $\mathcal{O}(N^2)$. As a result, the overall probing scheme still scales linearly with the number of alias candidates while using the correction heuristic.

## A.2 Addenda on `WISE` algorithms

This section reviews the time complexity of each algorithmic step of WISE [52, 53] as presented in Chapter 7, i.e.: target pre-scanning (Sec. A.2.1), target scanning (Sec. A.2.2), detection of problematic trails (cf. A.2.3), subnet inference (Sec. A.2.4) and subnet post-processing (Sec. A.2.5). Some of these sections also provide comments on the heuristics they involve when necessary. Finally, Sec. A.2.6 discusses how subnet post-processing is actually tuned in the implementation of WISE [49] and assesses its practical effects.

### A.2.1 Time complexity of target pre-scanning

Assuming the target IPv4 prefixes provided for WISE encompass up to $N$ attributable IP addresses and that the target pre-scanning sends a bounded number of probes towards each target IP address (two to three probes with the public implementation of WISE), the time complexity is $\mathcal{O}(N)$ , i.e., it is linear in respect of the total number of target IP addresses. In practice, multithreading is crucial to reduce the total execution time. Reducing the initial timeout delay can also speed up the pre-scanning, but this comes at a the cost of potentially discovering less responsive IP addresses.

### A.2.2 Time complexity of target scanning and probing heuristic assessment

The time complexity of the target scanning obviously depends on the number of responsive IP addresses discovered during pre-scanning (cf. Sec. 7.3.1). This value can be expressed as $N$. For an individual target IP address, the very worst case would be to perform the `traceroute`-like forward probing starting with an initial TTL value of 1 and increasing to the maximum TTL value defined by the IP protocol [102], i.e. 255, or a maximum TTL value allowed by the program (e.g., 64; this value is used in `WISE`). This worst case corresponds to an IP address that was responsive during pre-scanning, but no longer replies during target scanning (or is located unusually far if using a lower maximum TTL value than 255), and for which no previous TTL distance was available to benefit from the probing reduction heuristic. But even in this (rare) worst case scenario, the total number of operations (probes) is bounded by a constant. As the operations for a given target IP address do not depend on $N$, it is safe to assume that scanning a single IP address is in constant time ($\mathcal{O}(1)$) and that the time complexity of target scanning is therefore $\mathcal{O}(N)$, i.e., in linear time.

In practice, target scanning can be a cumbersome process if performed without the probing reduction heuristic and without multithreading. While the benefits of multithreading are immediate and obvious (given $M$ threads, each thread can scan $N/M$ targets), the benefits of the probing reduction heuristic should be assessed. To do so, another look at the test measurements conducted with `SAGE` [54] in April 2021 will be taken (they were already discussed in Sec. A.1.2). Indeed, `SAGE` (cf. Chapter 11) is built on top of `WISE`, and therefore starts with the same sequence of algorithm steps as `WISE` itself, therefore including a target scanning step. Moreover, with its *slightly verbose* mode (also mentioned in Sec. A.1.2), `SAGE` prints the (partial or full) `traceroute` paths collected for each scanned target IP address. By comparing the length of the collected (partial or full) paths with the TTL distance of each target IP address, it becomes possible to quantify how many probes have been spared. Indeed, the TTL distance can be used to infer the total number of probes that would have been sent if full `traceroute` measurements had been performed instead, while the length of the collected paths gives an estimation of how many probes were actually used.

| **Metrics** | **ULiège** | **AS12956** | **AS13789** | **AS224** | **AS6453** |
|---|---|---|---|---|---|
| Scanned IP addresses | 3,175 | 94,249 | 7,772 | 46,130 | 89,418 |
| Partial routes | 2,146 | 93,973 | 7,528 | 45,873 | 85,242 |
| Ratio of partial routes | 67.59% | 99.71% | 96.86% | 99.44% | 95.33% |
| Average TTL distance | 4.61 | 15.73 | 15.88 | 18.77 | 17.28 |
| Theoretical #probes | 14,646 | 1,482,962 | 123,385 | 865,980 | 1,545,412 |
| Actual #probes | 10,913 | 306,470 | 27,615 | 145,044 | 519,228 |
| Ratio of spared probes | 25.49% | 79.33% | 77.62% | 83.25% | 66.40% |

Table A.2: Quantifying the probing reduction heuristic with `SAGE` (April 2021).

Table A.2 shows the results of the probing reduction heuristic assessment for the five measurements conducted in early April 2021 from the Montefiore Institute, i.e., the same as discussed in Sec. A.1.2. The first part of the table provides the total number of responsive IP addresses that could be successfully scanned, along with the number and ratio of partial `traceroute` records. These first metrics are completed with the average TTL distance, as an indication of how many probes would have been necessary, on average, to complete a `traceroute` measurement towards an IP address of the given target network. The second part of the table presents how many probes would have been needed to collect complete `traceroute` paths towards all responsive IP addresses, based on their respective TTL distances, and how many probes were actually sent during target scanning. Finally, the last line gives the ratio of theoretical probes that were not actually sent.

Table A.2 clearly demonstrates that the probing reduction heuristic spares a considerable amount of probes, even when the vantage point is close to or within the target network: indeed, the *ULiège* column suggests that 25% of theoretical `traceroute` probes were avoided. All other columns show that much more than half of the probes that would have, in theory, been required have been spared, with more than 75% of them being spared with three of the five measurements. In conclusion, the target scanning as implemented by both `WISE` [49] and `SAGE` [48] can be said to be efficient thanks to the combination of a linear complexity, multithreading and a powerful heuristic.

### A.2.3 Time complexity of problematic trail detection

Each kind of problematic trail can be quantified in $\mathcal{O}(N)$ where $N$ denotes the total number of scanned IP addresses, as long as the access to one entry in the IP dictionnary is in constant time (cf. Sec. 7.2.1). The case of the echoing trails is trivial, since each IP address with an echoing trail can be immediately flagged as such without consulting the data of other IP addresses. Warping trails are a different matter: for each scanned IP address, if its trail is anomaly-free, the entry in the IP dictionary matching the IP address found in the trail must be retrieved. Then, the TTL distance of the trail (i.e., TTL distance of the scanned IP address minus one) must be compared to the known TTL distance of the second entry. If the TTL distances do not match, the second entry can be flagged as warping, and the distance can be recorded. If the second entry has already been flagged as warping, then the only remaining operation consists of recording the new TTL distance (if new). This sequence of operations can be $\mathcal{O}(1)$ for a given scanned IP address as long as checking the IP dictionary is in constant time (as suggested in Sec. 7.2.1), resulting in $\mathcal{O}(N)$ (linear time) for the warping trail detection.

Likewise, identifying flickering IP addresses can be performed in linear time too. Indeed, all operations shown in Algorithm 8 (Sec. 7.3.3) are $\mathcal{O}(1)$ with the exception of the four instructions found before the very last one in the body of the loop (lines 22– 25 in Algorithm 8). The two first instructions are $\mathcal{O}(1)$ if accessing an entry of the IP dictionary is in constant time, while the two last can be $\mathcal{O}(1)$ too assuming the entries are added at the end of the lists (of flickering *peers*) without further processing, e.g. via a simple *push* operation (i.e., like the `push_back()` method in Algorithm 8). Once all flickering trails have been identified, the lists of flickering *peers* can easily be cleaned (i.e., they are sorted and duplicate occurrences are removed) so that each *peer* only occurs once.

### A.2.4 Time complexity of subnet inference

The subnet inference has a linear time complexity in respect of the number of scanned IP addresses it has to aggregate into subnets, which will be denoted by $N$. While assessing the prefix of a growing subnet, the algorithm from Sec. 7.4.1 has to consider and process a certain number of candidate interfaces. This number will be represented by $M$, which is strictly inferior to $N$ (because the inference never considers all $N$ interfaces at the same time by design) and bounded by the size of the current prefix divided by 2 (since the other half has already been evaluated at the previous expansion).

The evaluation of a subnet prefix is split into two main operations: review and diagnosis. During the review, each of the $M$ candidate interfaces is compared with the reference pivot only once, regardless of whether the reference pivot has been updated. All operations within the loop in Algorithm 10 (cf. Sec. 7.4.1) are in $\mathcal{O}(1)$ as long as the looking up of an alias pair/list for a given IP address in an alias set is $\mathcal{O}(1)$ too (as suggested by Sec. 7.2.2), since rules 4 and 5 require an alias set built after discovering flickering trails. In practice, because there are usually a small number of flickering trails, finding an alias in logarithmic time should barely change the overall execution time. Indeed, if the measurements conducted from the Montefiore Institute in early April 2021 with `SAGE` [54] (already discussed in Sec. A.1.2 and Sec. A.2.2) are re-examined, the total of IP addresses found in alias pairs/lists, recorded in the alias set, are negligible for every target network (especially compared to the total of IP addresses to handle). Table A.3 provides the total of successfully aliased flickering trails and the size of the largest alias list for each measurement. The total of scanned IP addresses is provided as well to give an order of magnitude for the number of IP addresses dealt with during each measurement [2]. Therefore, the time complexity of the review should ideally be $\mathcal{O}(M)$, and as close as possible to it in practice if finding an alias pair/list in the alias set is done in logarithmic time.

| **Metrics** | **ULiège** | **AS12956** | **AS13789** | **AS224** | **AS6453** |
|---|---|---|---|---|---|
| Scanned IP addresses | 3,175 | 94,249 | 7,772 | 46,130 | 89,418 |
| Aliased flickering trails | 4 | 70 | 8 | 48 | 22 |
| Largest alias list (#IPs) | 2 | 8 | 2 | 4 | 3 |

Table A.3: Quantifying aliased flickering trails with `SAGE` (April 2021).

The case of the diagnosis formalized in Algorithm 11 in Sec. 7.4.1 is not much more complex: for instance, counting the number of interfaces for each type (i.e., pivot, contra-pivot and outlier) can be done in linear time (i.e., $\mathcal{O}(M)$). The additional tests performed when several potential contra-pivot interfaces are discovered can be $\mathcal{O}(1)$, since they are constant operations involving at worst one or two loops over a bounded number of contra-pivot interfaces (cf. the constant `MAX_CONTRAPIVOTS` used in Algorithm 11). At the very least, these additional verifications can be $\mathcal{O}(M)$, notably if the overgrowth heuristic used by the public implementation of `WISE` is implemented: indeed, this heuristic has a time complexity proportional to $M$, since each candidate is handled at worst once. Therefore, the worst case for the diagnostic is $\mathcal{O}(2M) = \mathcal{O}(M)$, and the evaluation of a single subnet prefix as a whole is $\mathcal{O}(M)$. In an ideal scenario where no subnet shrinking takes place, each of the $M$ interfaces of the evaluation of a given prefix is evaluated by the whole algorithm only once, as these interfaces are removed from the initial list (i.e., $M$ is substracted to $N$). As a consequence, in this ideal scenario, the entire execution is proportional to $N$.

[2]Note that IP addresses found in flickering trails are not necessarily part of the scanned IP addresses.

However, growing and shrinking several subnets consecutively can lead to comparing one IP interface with others twice or more. To check whether or not the complexity of subnet inference is still linear, the two possible worst cases that could induce multiple comparisons of a given scanned IP interface with others will be examined. In the first scenario, the network exclusively consists of small subnets featuring only one interface (e.g., a /32 or a /31 prefix). On expanding the subnet, one or two other IP addresses are encompassed, but since they are not compatible with the current interface, they are quickly put back in the initial list of scanned IP addresses and therefore considered a second or a third time during the inference of the next subnets. Therefore, all IP addresses are compared to other (close) interfaces up to a bounded number of three times, i.e., $\mathcal{O}(3N) = \mathcal{O}(N)$.

In the second worst case scenario, the network consists of consecutive subnets whose prefix length progressively increases (e.g., a /25, then a /26, then a /27, etc. all contained in a /24 prefix). Indeed, the last expansion (assuming it occurs) of the first will encompass all other interfaces, but as they are on different subnets, they will be put back onto the list, after which the same scenario will occur again but on a smaller scale. In the end, the interface(s) of the smallest subnet will be compared as many times as there are subnets. Therefore, the total number of comparisons is bounded by $N + N/2 + N/4 + ... = 2N$. Given that the subnet inference as implemented by `WISE` does not go lower than the /20 prefix, this specific worst case even has an upper bound for the number of processed interfaces. Therefore, the complexity of the subnet inference of `WISE` is in linear time, i.e., $\mathcal{O}(N)$.

### A.2.5 Time complexity of subnet post-processing

The time complexity of the subnet post-processing step in `WISE` is linear in respect of the total number of subnets to post-process, which will be expressed as $N$. To demonstrate this, the number of subnets involved in a merging scenario will be represented by $M$. When a scenario is tested, each subnet involved is evaluated at most once, regardless of $M$, as each subnet is only compared to the initial undergrown subnet when it comes to the third (suggested) merging rule. The merging rules themselves, regardless of what they do, can be tested in constant time ($\mathcal{O}(1)$). Moreover, the last suggested rule can be easily optimized by ensuring each subnet maintains its numbers of pivot/contra-pivot interfaces and outliers to avoid repeatedly counting them, so that the number of interfaces encompassed by each subnet becomes irrelevant to the current problem. As a consequence, evaluating a single scenario is $\mathcal{O}(M)$.

When a merging scenario turns out to be valid, the subnet expansion goes on and another scenario will be evaluated for a larger $M$. Two worst case scenarios should be taken into consideration where several successful scenarios are evaluated, with $M$ eventually growing to be equal to $N$. There are two possibilities: either $M$ doubles each time (because the total of IP addresses encompassed by a prefix doubles when its length is decremented), or $M$ increases by one at each expansion. The first worst case would amount to having $N$ undergrown subnets with the same prefix length and successive in the IP address space (e.g., a /20 partitioned in 2,048 /31 subnets). In the second worst case, each newly overlapped subnet could have a prefix length smaller by one unit than the previously considered subnet, i.e., if we have an undergrown /30, then the first expansion overlaps another /30, then the second expansion encompasses a /29, and the third expansion would absorb a /28, etc.

With the first worst case, if $M$ becomes equal to $N$, there will be $N + N/2 + N/4 + ... = 2N$ subnet evaluations, which is linear. In the case of the second hypothetical worst case, since the total of subnets in the final merging scenario is equal to $N$, there is a total of $N + (N-1) + (N-2) + ... + 1 = N \times (N-1)/2$ subnet evaluations, which is quadratic. However, both cases have an upper bound in practice: indeed, as already stated in Sec. 5.3, /19 subnets have not been documented at the time of this research and `WISE` therefore does not produce subnets beyond the /20 prefix (this also means the largest possible undergrown subnet is a /21). As a result, growing and testing merging scenarios starting from a given undergrown subnet can be considered as being upper bounded, since there can be at most 12 consecutive decrementations of a prefix length. With the first worst case, there can be at most $N = 4,096$ subnets, while there can be at most $N = 12$ subnets with the second one.

Another possible worst case for post-processing could result from overlapping (and therefore evaluating) the same subnets several times by evaluating distinct merging scenarios (i.e., each starting from a different undergrow subnet) that all fail. The worst possible situation would be to consider a large undergrown subnet followed by a set of consecutive undergrown subnets with a progressively increasing prefix length (e.g., a /24, then a /25, etc. all encompassed in the same /23 prefix). An expansion of the first undergrown subnet would encompass all subsequent subnets. If merging all subnets does not produce a sound subnet, all subnets are put back, and the same scenario repeats itself with the second subnet and all subsequent subnets. Under the condition that all the merging scenarios fail, the post-processing evaluates a total of $N + (N-1) + (N-2) + ... + 1 = N \times (N-1)/2$ subnets (where $N$ is the total number of subnets), which is quadratic. However, the total number of failed merging scenarios is bounded by the maximum number of expansions too: there cannot be more than 12 expansions, and therefore, such a scenario can involve at most $N = 12$ subnets. In other words, all worst cases of post-processing are upper bounded. Since expanding an undergrown subnet and considering its merging with other subnets is upper bounded, it can be considered as $\mathcal{O}(1)$, and as a result, subnet post-processing can be assumed to be $\mathcal{O}(N)$ as a whole, with $N$ being the total number of subnets to post-process.

### A.2.6 Practical effects of subnet post-processing

Since `WISE` only starts considering a merging scenario once it has detected an undergrown subnet within the list of inferred subnets, post-processing can be an easy step as long as subnet inference only produced a few undergrown subnets. In practice, because the public implementation of `WISE` adds some flexibility to subnet post-processing (i.e., it can reconsider a type of interface as another one), the process can also aggregate together undergrown subnets that do not fall into the partitioning scenario depicted by Fig. 7.2 in Sec. 7.4.2. This is also because, contrary to subnet inference which processes the IP addresses in decreasing order (with respect to their 32-bit integer equivalents), the post-processing reviews the subnets in ascending order in respect of their prefix.

As a result, the post-processing in WISE not only recovers large partitioned subnets (as depicted in Fig. 7.2 in Sec. 7.4.2), but also aggregates together sets of consecutive undergrown subnets into larger subnets. Because the resulting subnets rarely fulfill all soundness rules, they can be considered as delimiting actual subnets for which the collected data is inconclusive. Whether a subnet inference tool reusing the WISE methodology should do this can be debated. The public implementation of WISE [49] implements this behaviour for practical reasons: most undergrown subnets that eventually get merged together feature very long prefixes and usually encompass one or two observed subnet interfaces, while the resulting merged subnets are usually small subnets. The former can be considered to be measurement artefacts, while the latter can be interpreted as a form of coarse-grained subnet inference. In fact, a majority of subnets resulting from post-processing have a prefix length of 29 or more (i.e., eight attributable IP addresses or less).

| **Metrics** | **ULiège** | **AS12956** | **AS13789** | **AS224** | **AS6453** |
|---|---|---|---|---|---|
| Subnets before post-processing | 353 | 14,943 | 2,608 | 4,526 | 40,990 |
| Subnets after post-processing | 321 | 10,098 | 1,987 | 4,007 | 24,801 |
| Total of merged subnets | 25 | 1,165 | 293 | 286 | 3,808 |
| Average number of merged subnets | 2.28 | 5.159 | 3.119 | 2.815 | 5.251 |
| Median prefix length (merged) | 30 | 29 | 29 | 29 | 29 |

Table A.4: Quantifying effects of subnet post-processing with SAGE (April 2021).

Table A.4 quantifies the effects of subnet post-processing in WISE with the five measurements conducted from the Montefiore Institute with SAGE [54] in April 2021 (already discussed in Sec. A.2.2 and Sec. A.2.4). Table A.4 shows the number of subnets before and after subnet post-processing, how many subnets resulted from it, and what were the average number of merged subnets and the median prefix length. In all measurements, the post-processing merged small amounts of subnets resulting in quite long prefixes, as the median prefix length of merged subnets is 29 for all target networks except for the ULiège network (30), showing that most merged subnets actually aggregate very long prefixes that could be considered as artefacts. As such, the subnet post-processing implemented by WISE both fixed large subnets and reduced the number of measurement artifacts. Making the post-processing stricter could result in more inferred subnets after post-processing, but with a large number of very long prefixes that would be hard to interpret, especially with difficult targets such as AS12956 and AS6453, two Tier-1 ASes for which a lot of traffic engineering is usually observed.

## A.3 Addenda on SAGE algorithms

Just like Sec. A.2 in respect of the WISE algorithms, this section analyzes the time complexity of each algorithmic step of SAGE [54] as presented in Chapter 11, i.e., all algorithms that are unique to SAGE. Most of these dedicated algorithms constitute the sub-steps of its neighborhood-based graph inference (cf. Sec. 11.2): subnet aggregation (Sec. A.3.1), peer discovery (Sec. A.3.2), peer aggregation (Sec. A.3.3), vertex creation (Sec. A.3.5) and edge creation (Sec. A.3.6). In addition, this section also reviews empirical data to demonstrate the benefits of the peer aggregation sub-step (Sec. A.3.4) and analyzes the time complexity of the final alias resolution step of `SAGE` (Sec. A.3.7).

### A.3.1 Time complexity of subnet aggregation

The time complexity of subnet aggregation will depend on the complexity of the look-up operation in the map listing the subnets by aggregation property. The number of subnets to aggregate into neighborhoods will be represented by $N$ while the total number of unique aggregation properties (therefore the number of neighborhoods obtained in the end) is denoted by $M$. If the look-up of a list of subnets based on a property is in constant time ($\mathcal{O}(1)$), then the complexity of subnet aggregation can be expressed as $\mathcal{O}(N+M)$, since there is one loop for iterating the subnets and a second loop for turning each list of subnets into a neighborhood. At worst, one neighborhood will be created per subnet, therefore at worst $N = M$ and the complexity can be re-expressed as $\mathcal{O}(2N) \rightarrow \mathcal{O}(N)$.

If the look-up in the map is $\mathcal{O}(\log M)$ (for instance, because the map is implemented with a tree in practice), then the complexity becomes $\mathcal{O}(N \log N)$. Indeed, the complexity can first be re-expressed as $\mathcal{O}(N \log M + M)$, due to the access to the map in $\mathcal{O}(\log M)$. Since the worst case consists, again, of building one neighborhood for each subnet, then $\mathcal{O}(N \log N + N)$ is obtained, which can be simplified into $\mathcal{O}(N \log N)$. In summary, it is possible to perform this sub-step in linear time (e.g., if the map is implemented with a hash table). However, using a map with a look-up operation in logarithmic time will not significantly worsen the time complexity and can be a good compromise to miminize the amount of memory used by `SAGE`.

### A.3.2 Time complexity of peer discovery

The time complexity of peer discovery mostly depends on the number of partial `traceroute` paths being reviewed for each neighborhood. For a given neighborhood bordered by $S$ subnets, there is a number of paths proportional to $S$: indeed, the number of `traceroute` paths per subnet is bounded by design [3] and is equal to one for small subnets (i.e. /30, /31) which are usually the most common subnets to be found in the Internet (cf. Sec. 8.2.4). Finding the peer offset depends on $S$ due to the size of routes being bounded (255 at the very worst, much less in practice), again by design. Since enumerating peers and miscellaneous hops afterwards consists of looking at the selected offset in each route, this part of the algorithm also scales with $S$.

The enumeration of routes (lines 2– 7 in Algorithm 15 from Sec. 11.3.4) can also be considered to scale with $S$, as long as the routes can be consulted in an object-oriented manner like in Algorithm 15. Indeed, the total number of subnet interfaces within a same subnet is bounded by a relatively small constant: if the smallest subnet prefix is /20 [4], there are, at worst, 4096 interfaces to review (fully populated subnet, prefix and broadcast addresses included). In practice, this part does not significantly slow down the peer discovery because large subnets (e.g. /20, /21 or /22 subnets) are

[3] For reminders, by default, `SAGE` collects up to 5 partial routes per subnet.

[4] For reminders, this is the smallest subnet prefix length allowed in `WISE`, therefore also in `SAGE`.

less frequent (cf. Sec. 8.2.4). Therefore, peer discovery for a single neighborhood is $\mathcal{O}(S)$. In the end, if all neighborhoods are considered, peer discovery can be considered as being $\mathcal{O}(N)$ if $N$ denotes the total number of subnets bordering all discovered neighborhoods (or, in other words, the sum of all $S_i$ if $S_i$ corresponds to the aforementioned $S$ variable for a neighborhood $N_i$).

### A.3.3 Time complexity of peer aggregation

The number of neighborhoods can be expressed as $N$. Because peers are by definition (anomaly-free) trails of subnets, which are used to denote neighborhoods, it can be assumed that the number of peers for a given neighborhood is always strictly inferior to $N$, as it is impossible for a neighborhood to be a peer for itself (the preliminary `traceroute` probing eliminated the risk, cf. Sec. 11.3.2). Moreover, it is very unlikely that a neighborhood will have all other neighborhoods as peers and reciprocally. At worst, all the others can be peers, but they are likely to have few peers (or none) in comparison because of `traceroute` data. Furthermore, previous results with `WISE` showed that most neighborhoods usually have one or few peers [53] (cf. Sec. 10.3.3.3). Therefore, the average number of peers per neighborhood can be considered to be $\log N$, as a close approximation to reality.

The main algorithmic challenge of peer aggregation is the computation of the largest hypothetical alias lists. It means taking all $\log N$ peers of a neighborhood and updating the IP aggregator (or comparable structure) for all of them. Assuming this IP aggregator is implemented with a map which binds an IP address to a list of its potential aliases, since all subnet trails (proportional to $N$) can be present in the map, and assuming there is one list update per peer (due to the content of the map), the update of the map can be $\mathcal{O}(\log N)$ if the look-up operation with the map is $\mathcal{O}(1)$ or $\mathcal{O}(\log^2 N)$ if the same operation is in logarithmic time. Therefore, for $N$ neighborhoods, the complexity of the aggregation itself is either $\mathcal{O}(N \log N)$ or $\mathcal{O}(N \log^2 N)$.

The complexity for the subsequent alias resolution depends on which alias resolution method is being used and how it is implemented. In the case of the public implementation of `SAGE`, the alias resolution is done in linear time with the fingerprint-based framework introduced in Chapter 3 (see also Sec. A.1 for the detailed analysis of this framework). Since the total of alias candidates to resolve is, at worst, the total number of peers, which is itself, at worst, equivalent to $N$ (because neighborhoods are built on the basis of the same IP addresses as peers), then the alias resolution can be done in $\mathcal{O}(N)$ regardless of the number and size of the alias lists to resolve, assuming the alias resolution itself can be done in linear time.

As an aside, blindspot detection and the subsequent improvement of peers should feature time complexities very close to the two previous steps. Indeed, blindspot detection only consists of testing whether (known) miscellaneous hops associated to each neighborhood (if any) match a previously discovered alias pair/list, and it therefore scales with the number of neighborhoods. While the number of miscellaneous hops for a given neighborhood has no relationship with $N$, it can be assumed that such a number is much smaller than the total number of neighborhoods and close to the number of peer addresses, meaning that $\log N$ can be kept as a fair approximation. Depending on how the alias set is implemented, the time complexity is $\mathcal{O}(N \log N)$ (if getting an alias pair/list is in $\mathcal{O}(1)$) or $\mathcal{O}(N \log^2 N)$ (if getting an alias is in logarithmic time). Due to its proximity with peer discovery, the process of improving the peers afterwards can be in linear time too (cf. Sec. A.3.2). In practice, because blindspots are not frequent, detecting them and using this knowledge to improve peers should constitute a light operation, as will be discussed in the next section (Sec. A.3.4).

### A.3.4 Practical results of peer aggregation

To complement the time complexity analysis (cf. Sec. A.3.3) and discuss the soundness of peer aggregation, Table A.5 provides various metrics about peer aggregation based on the measurements from April 2021 conducted with the public implementation of SAGE from the Montefiore Institute (already discussed in Sec. A.1.2 and Sec. A.2.2). For each target network, the table shows – among others – the total number of alias candidates (including both peer addresses and miscellaneous hops) involved in the detection of convergence points, how many IP addresses were listed under the largest hypothetical convergence point and how many IP addresses were included in the largest discovered alias list. Finally, the number of peer lists (equivalent to a number of neighborhoods) featuring blindspots at the end of peer aggregation is also provided.

| **Metrics** | **ULiège** | **AS12956** | **AS13789** | **AS224** | **AS6453** |
|---|---|---|---|---|---|
| Total of alias candidates | 9 | 341 | 20 | 33 | 149 |
| Discovered alias lists | 4 | 40 | 7 | 4 | 29 |
| Largest hypothesis (#IPs) | 3 | 121 | 2 | 3 | 19 |
| Largest alias list (#IPs) | 3 | 40 | 2 | 3 | 8 |
| Aliased IP addresses | 9 (100%) | 214 (62.7%) | 14 (70%) | 9 (27.2%) | 86 (57.7%) |
| Most common method | Group (75%) | Ally (72.5%) | Ally (85.7%) | Ally (50%) | Ally (82.7%) |
| Peer lists w/ blindspots | 1 | 32 | 0 | 1 | 3 |

Table A.5: Quantifying peer aggregation results in SAGE (April 2021).

AS12956 and AS6453 clearly stands out in Table A.5, both being Tier-1 ASes with complex architectures (involving some load balancing), but the metrics still demonstrate that the hypothetical alias lists and actual alias lists were quite small. For instance, at most 19 IP addresses were considered together while probing AS6453, and the largest alias list (obtained with the Ally-like method, cf. Sec. 3.3.1.2) only consisted of 8 router interfaces. The largest hypothetical list with AS12956 encompassed 121 IP addresses, which remained small enough to be resolved with the fingerprint-based framework of Chapter 3. While the largest alias list (40 interfaces) was obtained by grouping interfaces by fingerprint, due to the lack of other exploitable methods, the second largest list (35 interfaces) was obtained with Ally, suggesting that the former might not be as oversized as it first seemed [5]. Finally, the metrics for the ULiège network, AS13789 and AS224 shows there were potential convergence points for all measurements, several of them being confirmed through alias resolution. Though the

[5]It is worth noting that the public implementation of SAGE provides an optional parameter to force peer aggregation to only keep alias pairs/lists discovered via the Ally-like and iffinder-like methods for the sake of accuracy.

alias lists for the ULiège network were built on the basis of fingerprints (again, because other methods could not be used), groundtruth data showed they corresponded to actual routers.

Finally, the last line of Table A.5 shows that there were few neighborhoods with blindspots (cf. Sec. 11.1.2). In fact, during these measurements, no neighborhood obtained better peers through blindspot discovery. However, snapshots of AS12956 captured from the EdgeNet cluster during 2021 included neighborhoods whose peers were occasionally improved through blindspot detection. Overall, Table A.5 shows that peer aggregation is not a computationally difficult task, but also demonstrates its necessity: convergence points can be observed even with (allegedly) simple topologies. Moreover, as mentioned in Sec. A.2.4, `SAGE` already previously found convergence points via flickering trails, not counted in Table A.5 due to the alias transitivity heuristic (cf. Sec. 11.3.1).

### A.3.5 Time complexity of vertex creation

There are two main algorithmic operations during vertex creation: on the one hand, building the peers map, and on the other hand, isolating the termini neighborhoods. Before reviewing the first operation, $N$ will be used to denote the number of neighborhoods. In the same way as for peer aggregation, the average number of peers per each neighborhood can be approximated by $\log N$ for the same reasons as before (cf. Sec. A.3.3). Because the total number of IP addresses present in the alias set should be close to $N$ (i.e., total of peer addresses used to identify neighborhoods with an epsilon accounting for miscellaneous hops), the look-up of one alias within the alias set is $\mathcal{O}(\log N)$ if the map is implemented with a tree and $\mathcal{O}(1)$ if the same map is implemented with a hash table. Knowing this, the time complexity of building the peers map is $\mathcal{O}(N \log N)$ if the look-up in both the map and the alias set is in constant time and $\mathcal{O}(N \log^2 N)$ if the look-up is performed in logarithmic time. Indeed, each iteration of the outer loop reviews each peer address recorded for each neighborhood ($N$ neighborhoods, each with $\log N$ peers). Then, it looks for each peer address in both the peers map and the alias set at worst, which means the complexity of one iteration of the inner loop is $\mathcal{O}(2 * \log N)$ if using tree structures, i.e., $\mathcal{O}(\log N)$ overall and $\mathcal{O}(1)$ if using hash tables.

The complexity for listing the termini neighborhoods is simpler: this consists of a single loop browsing the $N$ neighborhoods, with each iteration performing a single look-up operation in the peers map. While there can be several labels (i.e., trail(s) denoting the neighborhood) for a neighborhood at this stage due to the existence of flickering trails, it is safe to assume that the cost of iterating over such labels is negligible in practice due to flickering trail analysis producing small alias lists (cf. Sec. A.2.4), and this is even more simple if the alias transitivity heuristic (cf. Sec. 11.3.1) is used. In other words, the only constraining factor in an iteration of the loop finding termini neighborhoods is the look-up in the peers map. Therefore, the time complexity of this operation should be $\mathcal{O}(N)$ if the look-up in the map is in constant time (hash table implementation) and in $\mathcal{O}(N \log N)$ if the same operation is in logarithmic time (tree implementation).

Finally, the creation of the vertices consists, in practice, of directly translating the termini neighborhoods into vertices ($\mathcal{O}(T)$ if $T$ is the number of termini neighborhoods) while the other vertices are created with the help of a list of peers and a map associating each peer address with a neighborhood. If $P$ denotes the number of neighborhoods acting as peers, it would be safe to assume this operation scales in the same manner as previous operations, i.e. $\mathcal{O}(P \log P)$ if using a map with constant time look-up and $\mathcal{O}(P \log^2 P)$ if the same operation is in logarithmic time.

### A.3.6 Time complexity of edge creation

The time complexity of edge creation scales with two variables: $N$, the number of neighborhoods, and $S$, the total number of subnets bordering all neighborhoods. Again, it can be considered (for the same reasons as in Sec. A.3.3) that $\log N$ can denote the number of peers for each neighborhood as a fair approximation. It is worth noting that, in practice, there are in fact less peers after creating the final vertices as a consequence of peer aggregation (if alias pairs/lists were discovered), though this should not change the notation. The number of subnets per neighborhood, on the other hand, is more complicated to denote: at worst, there will be one neighborhood per subnet. The opposite worst scenario is a trivial case: if all subnets are under the same neighborhood, then $N = 1$ and there are no edges to create. However, because there are multiple possible subnet distributions in the wild (depending on the architecture of a network), there can be dozens (even hundreds) of subnets for one neighborhood and only one or two in other cases. The average number of subnets per neighborhood will therefore be expressed as $S/N$.

The time complexity of the first part of Algorithm 18 from Sec. 11.3.7 can be expressed as $\mathcal{O}(N * S/N) \rightarrow \mathcal{O}(S)$ since the only constraining factor in an iteration of the loop is the insertion of the $S/N$ subnets into the subnet dictionary, other operations being in $\mathcal{O}(1)$. The second loop (creation of the edges) scales with $N$, because there is at worst a single gate (therefore, $N-1$ vertices to connect, which can be simplified as $N$). Each iteration of this second loop also scales with the number of subnets of $u$ due to the code at least looking for a subnet $S_u$ bordering $u$ encompassing an (anomaly-free) trail identifying $v$ in an attempt to create a direct link between $u$ and $v$ (the list being completely visited only if no subnet can be found). However, the loops iterating over the anomaly-free trails identifying $v$ should also be accounted for (denoted as *labels* in the pseudo-code of Algorithm 18). Because of peer aggregation (cf. Sec. 11.3.5), this number can no longer be considered negligible (though it can stay computationally small, cf. Sec. A.3.4). A safe approximation of the maximum number of (anomaly-free) trails identifying one neighborhood can be $S/N$: at worst, there will be as many trails as there are subnets (assuming neighborhoods with a single subnet have been aggregated). It would in fact mean each subnet bordering $v$ to have its own unique trail in order to have more than $S/N$ trails, which is not a frequent occurrence in the wild. This gives us the complexity $\mathcal{O}(S^2/N^2)$ for an iteration (the worst case consisting in not finding connecting subnet for a direct link). Therefore, if other operations during an iteration are in $\mathcal{O}(1)$ (if the subnet dictionary is designed as recommended), this means the complexity of this operation is $O(N * S^2/N^2) \rightarrow \mathcal{O}(S^2/N)$. This time complexity becomes $\mathcal{O}(S)$ if there is one subnet per neighborhood ($S = N$). If there is only one neighborhood, no operation is actually carried out in practice, since a single neighborhood bordering by all $S$ subnets cannot have peers, meaning the complexity always stays below $\mathcal{O}(S^2)$.

It could then be argued that, because of its $O(1)$ access (if using the recommended approach), using the dictionary to find a subnet for a direct link could be in fact more interesting as long as the subnet object provides the neighborhood it borders. This would lead to $O(S)$ complexity ($\mathcal{O}(N * S/N) \rightarrow O(S)$ where $S/N$ denotes the maximum of subnet trails identifying a neighborhood). In practice, it should be borne in mind that the public implementation of SAGE implements the subnet dictionary with a large array of lists to operate a time/memory trade-off (in the same manner of its IP dictionary, see Sec. 7.2.1). Depending on how large the array is, the maximum size of the lists can still be greater than the number of iterations for a (failed) search of a subnet to create a direct link. In addition, having as many trails as there are subnets will most likely occur with small neighborhoods, and very rarely with large ones, due to such a configuration requiring all subnets to have their own unique trails. As a consequence, Algorithm 18 should, in practice, behave well and should be considered especially if the implementer decides to use another solution for the subnet dictionary to use less memory at the cost of a slightly worse time complexity for finding a subnet.

### A.3.7 Time complexity of the final alias resolution step

The time complexity of the final alias resolution step in SAGE depends on two variables: the number of neighborhoods (or vertices), subsequently denoted by $N$, and the number of subnets, denoted by $S$. Because the number of subnets can vary significantly from one neighborhood to another, the number of subnets per neighborhood can be modeled, again, by the average number of subnets $S/N$ (already used in Sec. A.3.6). For each neighborhood, the operations performed in Algorithm 19 from Sec. 11.4 scales with the number of subnets bordering this neighborhood ($\mathcal{O}(S/N)$). Indeed, each subnet provides a bounded number of contra-pivot interfaces (maximum 5 in WISE/SAGE by default). Moreover, the selection of contra-pivot interfaces among each subnet is in constant time: since the subnet prefix length has a minimum value, the number of interfaces listed for each subnet is bounded. If the implementation chooses to store contra-pivot interfaces in a list separating them for all other known interfaces, then the upper boundary is reduced to the maximum number of contra-pivot interfaces per subnet the tool accepts. While the number of trails used to identify the neighborhood (which are part of the alias candidates) can vary, this number will rarely be greater than the average number of subnets $S/N$ (as already explained in Sec. A.3.6). If all subnets have the maximum of contra-pivot interfaces *max*, then it can be said that there is a maximum of alias candidates $S/N + max * S/N \rightarrow (max + 1) * S/N$, and the time complexity to perform alias resolution on a single neighborhood is $\mathcal{O}(S/N)$. Since the overall complexity for the final alias resolution also scales with $N$ (the number of neighborhoods), it can be expressed as $\mathcal{O}(N * S/N) \rightarrow \mathcal{O}(S)$, but only as long as the alias resolution scales with the number of alias candidates (in the same manner as with the fingerprint-based framework from Chapter 3).

# BIBLIOGRAPHY

[1] *Ark - CAIDA.*
`https://www.caida.org/projects/ark/`.
Accessed: June 18, 2021.

[2] *CAIDA's Ranking of Autonomous Systems (ASRank).*
`https://asrank.caida.org/`.
Accessed: July 7, 2021.

[3] *Empowering App Development for Developers | Docker.*
`https://www.docker.com/`.
Accessed: June 18, 2021.

[4] *Hurricane Electric BGP Toolkit.*
`https://bgp.he.net/`.
Accessed: June 23, 2021.

[5] *Internet world stats: Usage and population statistics.*
`https://www.internetworldstats.com/stats.htm`.
Accessed: July 6, 2021.

[6] *Kubernetes.*
`https://kubernetes.io/`.
Accessed: June 18, 2021.

[7] *MONROE - Measuring Mobile Broadband Networks in Europe.*
`https://www.monroe-project.eu/`.
Accessed: June 18, 2021.

[8] *Planetlab | An open platform for developing, deploying and accessing planetary-scale services.*
`https://planetlab.cs.princeton.edu/`.
Accessed: June 18, 2021.

[9] *What is RIPE Atlas ? | RIPE Atlas.*
`https://atlas.ripe.net/about/`.
Accessed: June 18, 2021.

[10] D. Alderson, L. Li, W. Willinger, and J. C. Doyle, *Understanding Internet Topology: Principles, Models, and Validation*, IEEE/ACM Transactions on Networking, vol. 13, no. 6, pp. 1205–1218, December 2005.

[11] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira, *Avoiding Traceroute Anomalies with Paris traceroute*, in Proc. ACM Internet Measurement Conference (IMC), October 2006.

[12] B. Augustin, T. Friedman, and R. Teixeira, *Measuring Multipath Routing in the Internet*, IEEE/ACM Transactions on Networking, vol. 19, no. 3, pp. 830–840, June 2011.

[13] B. Augustin, B. Krishnamurthy, and W. Willinger, *IXPs: Mapped?*, in Proc. ACM Internet Measurement Conference (IMC), November 2009.

[14] B. Augustin, R. Teixeira, and T. Friedman, *Measuring Load-Balanced Paths in the Internet*, in In Proc. ACM Internet Measurement Conference (IMC), October 2007.

[15] P. Baran, *On distributed communications networks*, IEEE Transactions on Communications Systems, vol. 12, no. 1, pp. 1–9, 1964.

[16] A. Bender, R. Sherwood, and N. Spring, *Fixing Ally's Growing Pains with Velocity Modeling*, in Proc. ACM Internet Measurement Conference (IMC), October 2008.

[17] T. Berners-Lee, R. Fielding, and H. Frystyk, *Hypertext Transfer Protocol – HTTP/1.0*, RFC 1945, Internet Engineering Task Force, May 1996.

[18] R. Beverly, *Yarrp'ing the Internet: Randomized high-speed active topology discovery*, in Proc. ACM Internet Measurement Conference (IMC), November 2016.

[19] R. Beverly, R. Durairajan, D. Plonka, and J. P. Rohrer, *In the IP of the Beholder: Strategies for Active IPv6 Topology Discovery*, in Proc. ACM Internet Measurement Conference (IMC), November 2018.

[20] R. Braden, *Requirements for Internet Hosts – Communication Layers*, RFC 1700, Internet Engineering Task Force, October 1989.

[21] V. Breña-Medina, *University of Bristol Thesis Template*.
`https://fr.overleaf.com/latex/templates/university-of-bristol-thesis-template/kzqrfvyxxcdm/`.
Accessed: February 5, 2021.

[22] CAIDA, *AS relationships dataset*.
`https://data.caida.org/datasets/as-relationships/`.
Accessed: July 7, 2021 (login must be skipped twice).

[23] ——, *AS Core: Visualizing IPv4 Internet Topology at a Macroscopic Scale in 2017.* `https://www.caida.org/projects/cartography/as-core/2017/`, 2017. Accessed: July 6, 2021.

[24] ——, *CAIDA's ipv4 and ipv6 AS Core: Visualizing IPv4 and IPv6 Internet Topology at a Macroscopic Scale in 2020.* `https://www.caida.org/projects/cartography/as-core/2020/`, April 2021. Accessed: July 7, 2021.

[25] V. CERF AND R. KAHN, *A Protocol for Packet Network Intercommunication*, IEEE Transactions on Communications, vol. 22, no. 5, pp. 637–648, May 1974.

[26] Q. CHEN, H. CHANG, R. GOVINDAN, S. JAMIN, S. SHENKER, AND W. WILLINGER, *The Origin of Power Laws in Internet Topologies Revisited*, in Proc. IEEE INFOCOM, June 2002.

[27] CISCO, *Cisco Annual Internet Report 2018–2023.* `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`.

[28] A. CLAUSET AND C. MOORE, *Traceroute Sampling Makes Random Graphs Appear to Have Power Law Degree Distributions*, cond-mat 0312674, arXiv, February 2004.

[29] CREATIVE COMMONS, *Attribution 4.0 International (CC BY 4.0).* `https://creativecommons.org/licenses/by/4.0/deed.en`. Accessed: February 5, 2021.

[30] S. CROCKER, *Network Working Group Request for Comment: 1*, RFC 1, Internet Engineering Task Force, April 1969.

[31] W. DE DONATO, P. MARCHETTA, AND A. PESCAPÉ, *A Hands-On Look at Active Probing Using the IP Prespecified Timestamp Option*, in Proc. Passive and Active Measurement Conference (PAM), April 2012.

[32] M. DE KUNDER, *The size of the World Wide Web (The Internet).* `https://worldwidewebsize.com/`. Accessed: July 6, 2021.

[33] S. DEERING, *Host Extensions for IP Multicasting*, RFC 1112, Internet Engineering Task Force, August 1989.

[34] N. DEVELOPERS, *NetworkX – Network Analysis in Python.* `https://networkx.org/`. Accessed: May 21, 2021.

[35] W. DIDIMO AND G. LIOTTA, *Mining Graph Data*, John Wiley & Sons, Ltd, 2006.

[36] X. DIMITROPOULOS, D. KRIOUKOV, M. FOMENKOV, B. HUFFAKER, Y. HYUN, K. CLAFFY, AND G. RILEY, *AS Relationships: Inference and Validation*, ACM SIGCOMM Computer Communication Review, vol. 37, pp. 29–40, May 2006.

[37] B. DONNET AND T. FRIEDMAN, *Internet topology discovery: a survey*, IEEE Communications Surveys and Tutorials, vol. 9, no. 4, pp. 2–15, December 2007.

[38] B. DONNET, P. RAOULT, AND T. FRIEDMAN, *Efficient Route Tracing from a Single Source*, tech. rep., May 2006.
Can be consulted at `http://arxiv.org/abs/cs/0605133`.

[39] B. DONNET, P. RAOULT, T. FRIEDMAN, AND M. CROVELLA, *Efficient Algorithms for Large-Scale Topology Discovery*, in Proc. ACM SIGMETRICS, June 2005.

[40] M. FALOUTSOS, P. FALOUTSOS, AND C. FALOUTSOS, *On Power-Law Relationships of the Internet Topology*, in SIGCOMM, August 1999, pp. 251–262.

[41] D. FELDMAN, Y. SHAVITT, AND N. ZILBERMAN, *A Structural Approach for PoP Geo-Location*, Computer Networks (COMNET), vol. 56, no. 3, pp. 1029–1040, February 2012.

[42] K. D. FRAZER, *NSFNET: A Partnership for High-Speed Networking*.
`https://www.merit.edu/wp-content/uploads/2019/06/NSFNET_final-1.pdf`.
Accessed: July 14, 2021.

[43] V. FULLER AND T. LI, *Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan*, RFC 4632, Internet Engineering Task Force, August 2006.

[44] F. GONT, R. ATKINSON, AND C. PIGNATARO, *Recommendations on filtering of IPv4 packets containing IPv4 options*, RFC 7126, Internet Engineering Task Force, February 2014.

[45] R. GOVINDAN AND H. TANGMUNARUNKIT, *Heuristics for Internet map discovery*, in Proc. IEEE INFOCOM 2000, vol. 3, August 2000, pp. 1371–1380.

[46] J.-F. GRAILET, *RTrack, a traceroute extension to discover routing anomalies*.
`https://github.com/JefGrailet/rtrack`.
Public GitHub repository.

[47] ——, *SAGE (Subnet AGgrEgation) - Beta version*.
`https://github.com/JefGrailet/SAGE_beta`.
Public GitHub repository.

[48] ——, *SAGE (Subnet AGrEgation)*.
`https://github.com/JefGrailet/SAGE`.
Public GitHub repository.

[49] ——, *WISE (Wide and lInear Subnet inferencE)*.
`https://github.com/JefGrailet/WISE`.
Public GitHub repository.

[50] J.-F. GRAILET AND B. DONNET, *Discovering Routers in Load-balanced Paths*, in Proc. ACM CoNEXT Student Workshop (CoNEXT), December 2017.

[51] ——, *Towards a Renewed Alias Resolution with Space Search Reduction and IP Fingerprinting*, in Proc. Network Traffic Measurement and Analysis Conference (TMA), June 2017.

[52] ——, *Revisiting Subnet Inference WISE-ly*, in Proc. Network Traffic Measurement and Analysis Conference (TMA), June 2019.

[53] ——, *Virtual Insanity: Linear Subnet Discovery*, IEEE Transactions on Network and Service Management (TNSM), vol. 17, no. 2, pp. 1268–1281, June 2020.

[54] ——, *Travelling Without Moving: Discovering Neighborhood Adjacencies*, in Proc. Network Traffic Measurement and Analysis Conference (TMA), September 2021.
To appear at the conference (accepted).

[55] J.-F. GRAILET, F. TARISSAN, AND B. DONNET, *TreeNET: Discovering and Connecting Subnets*, in Proc. Traffic and Monitoring Analysis Workshop (TMA), April 2016.

[56] J.-L. GUILLAUME AND M. LATAPY, *Bipartite graphs as models of complex networks*, Physica A: Statistical Mechanics and its Applications, vol. 371, no. 2, pp. 795–813, November 2006.

[57] M. H. GUNES AND K. SARAC, *Importance of IP Alias Resolution in Sampling Internet Topologies*, in Proc. IEEE Global Internet Symposium, May 2007.

[58] ——, *Inferring Subnets in Router-Level Topology Collection Studies*, in Proc. ACM/USENIX Internet Measurement Conference (IMC), November 2007.

[59] ——, *Resolving Anonymous Routers in the Internet Topology Measurement Studies*, in Proc. IEEE INFOCOM, April 2008.

[60] ——, *Resolving IP Aliases in Building Traceroute-Based Internet Maps*, IEEE/ACM Transactions on Networking (ToN), vol. 17, no. 6, pp. 1738–1751, December 2009.

[61] H. HADDADI, G. IANNACCONE, A. MOORE, R. MORTIER, AND M. RIO, *Network topologies: Inference, modeling and generation*, IEEE Communications Surveys and Tutorials, vol. 10, no. 2, pp. 48–69, April 2008.

[62] H. Haddadi, S. Uhlig, A. Moore, R. Mortier, and M. Rio, *Modeling Internet Topology Dynamics*, Computer Communication Review, vol. 38, pp. 65–68, March 2008.

[63] M. Hauben and R. Hauben, *Behind the Net: The Untold Story of the ARPANET and Computer Science (chapter 7).*, First Monday, vol. 3, February 1998.

[64] J. Hawkinson and T. Bates, *Guidelines for creation, selection, and registration of an Autonomous System (AS)*, RFC 1930, Internet Engineering Task Force, March 1996.

[65] Y. Huang, M. Rabinovich, and R. Al-Dalky, *FlashRoute: Efficient Traceroute on a Massive Scale*, in Proc. ACM Internet Measurement Conference (IMC), October 2020.

[66] A. Iamnitchi, R. Matei, and I. Foster, *Small World File-Sharing Communities*, in Proc. IEEE INFOCOM, April 2004.

[67] V. Jacobson, *mrinfo*.
`http://cvsweb.netbsd.org/bsdweb.cgi/src/usr.sbin/mrinfo/?onlywithtag=MAIN`.
Accessed: May 25, 2021.

[68] X. Jin, W.-P. K. Yiu, S.-H. G. Chan, and Y. Wang, *Network Topology Inference Based on End-to-End Measurements*, IEEE Journal on Selected Areas in Communication, Sampling the Internet: Techniques and Applications, vol. 24, no. 12, pp. 2182–2195, December 2006.

[69] R. Kahn, *Communications principles for operating systems*, Internal BBN memorandum, vol. 67, 1972.

[70] H. Kardes, M. H. Gunes, and T. Oz, *Cheleby: a Subnet-Level Internet Topology Mapping System*, in Proc. International Communications Systems and Networks and Workshops (COMSNETS), January 2012.

[71] K. Keys, *iffinder*.
`http://www.caida.org/tools/measurement/iffinder/`.
Accessed: July 14, 2021.

[72] ——, *Internet-scale IP Alias Resolution Techniques*, ACM SIGCOMM Computer Communication Review, vol. 40, no. 1, pp. 50–55, January 2010.

[73] K. Keys, Y. Hyun, M. Luckie, and k. claffy, *Internet-scale IPv4 Alias Resolution with MIDAR*, IEEE/ACM Transactions on Networking, vol. 21, no. 2, pp. 383–399, April 2013.

[74] S. Kim and K. Harfoush, *Efficient Estimation of More Detailed Internet IP maps*, in Proc. IEEE International Conference on Communications (ICC), June 2007.

[75] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, Pearson, 6 ed., 2012.

[76] A. LAKHINA, J. BYERS, M. CROVELLA, AND P. XIE, *Sampling Biases in IP Topology Measurements*, in Proc. IEEE INFOCOM, April 2003.

[77] M. LATAPY, C. MAGNIEN, AND N. DEL VECCHIO, *Basic notions for the analysis of large two-mode networks*, Social Networks, vol. 30, no. 1, pp. 31–48, January 2008.

[78] L. LI, D. ALDERSON, W. WILLINGER, AND J. DOYLE, *A First-Principles Approach to Understanding the Internet's Router-level Topology*, in Proc. ACM SIGCOMM, August 2004.

[79] K. LOUGHEED AND Y. REKHTER, *A Border Gateway Protocol 3 (BGP-3)*, RFC 1267, Internet Engineering Task Force, October 1991.

[80] M. LUCKIE, B. HUFFAKER, AND K. CLAFFY, *Learning Regexes to Extract Router Names from Hostnames*, in Proc. ACM Internet Measurement Conference (IMC), October 2019.

[81] M. LUCKIE, H. HYUN, AND B. HUFFAKER, *Traceroute probe method and forward IP path inference*, in Proc. ACM Internet Measurement Conference (IMC), October 2008.

[82] C. MAGNIEN, F. OUÉDRAOGO, G. VALADON, AND M. LATAPY, *Fast Dynamics in Internet Topology: Observations and First Explanations*, in 2009 Fourth International Conference on Internet Monitoring and Protection, May 2009, pp. 137–142.

[83] P. MARCHETTA, P. MÉRINDOL, B. DONNET, A. PESCAPÉ, AND J.-J. PANSIOT, *Topology Discovery at the Router Level: a New Hybrid Tool Targeting ISP Networks*, IEEE Journal on Selected Areas in Communication, Special Issue on Measurement of Internet Topologies, vol. 29, no. 6, pp. 1776–1787, October 2011.

[84] ——, *Quantifying and Mitigating IGMP Filtering in Topology Discovery*, in Proc. IEEE Global Communications Conference (GLOBECOM), December 2012.

[85] P. MARCHETTA, A. MONTIERI, V. PERSICO, A. PESCAPÉ, I. CUNHA, AND E. KATZ-BASSETT, *How and how much traceroute confuses our understanding of network paths*, in IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN) Conference, June 2016.

[86] P. MARCHETTA, V. PERSICO, AND A. PESCAPÉ, *Pythia: Yet Another Active Probing Technique for Alias Resolution*, in ACM CoNEXT, December 2013.

[87] A. MARDER, *APPLE: Alias Pruning by Path Length Estimation*, in Proc. Passive and Active Measurement Conference (PAM), March 2020.

[88] E. MARÉCHAL AND B. DONNET, *Network Fingerprinting: Routers under Attack*, in IEEE International Workshop on Traffic Measurements for Cybersecurity (WTMC), September 2020.

[89] A. Medina, I. Matta, and J. Byers, *On the Origin of Power Laws in Internet Topologies*, ACM SIGCOMM Computer Communications Review, vol. 30, no. 2, pp. 18–28, April 2000.

[90] P. Mérindol, B. Donnet, O. Bonaventure, and J.-J. Pansiot, *On the Impact of Layer-2 on Node Degree Distribution*, in Proc. ACM Internet Measurement Conference (IMC), November 2010.

[91] P. Mérindol, V. Van den Schriek, B. Donnet, O. Bonaventure, and J.-J. Pansiot, *Quantifying ASes multiconnectivity using multicast information*, in Proc. ACM Internet Measurement Conference (IMC), November 2009.

[92] J. Mogul and J. Postel, *Internet standard subnetting procedure*, RFC 950, Internet Engineering Task Force, August 1985.

[93] M. Newman, *Scientific Collaboration Networks. Network Construction and Fundamental Results*, Physical Review E, vol. 64, no. 1, June 2001.

[94] J. Noel Chiappa, *ARPANET Technical Information: Geographic Maps.*
`http://mercury.lcs.mit.edu/~jnc/tech/arpageo.html`.
Accessed: July 14, 2021.

[95] G. Nomikos and X. Dimitropoulos, *traIXroute: Detecting IXPs in traceroute paths*, in Proc. Passive and Active Measurement Conference (PAM), April 2016.

[96] J.-J. Pansiot, *mrinfo dataset.*
`http://svnet.u-strasbg.fr/mrinfo/`.
Accessed: May 25, 2021.

[97] J.-J. Pansiot, P. Mérindol, B. Donnet, and O. Bonaventure, *Extracting Intra-Domain Topology from mrinfo Probing*, in Proc. Passive and Active Measurement Conference (PAM), April 2010.

[98] R. Pastor-Satorras and A. Vespignani, *Evolution and structure of the Internet: A statistical physics approach*, Cambridge University Press, 2007.

[99] L. Peterson, *It's Been a Fun Ride.*
`https://www.systemsapproach.org/blog/its-been-a-fun-ride`.
Accessed: April 16, 2021.

[100] J. Postel, *User Datagram Protocol*, RFC 768, Internet Engineering Task Force, August 1980.

[101] ——, *Internet Control Message Protocol*, RFC 792, Internet Engineering Task Force, September 1981.

[102] ——, *Internet Protocol*, RFC 791, Internet Engineering Task Force, September 1981.

[103] ——, *Transmission Control Protocol*, RFC 793, Internet Engineering Task Force, September 1981.

[104] ——, *Assigned numbers*, RFC 1700, Internet Engineering Task Force, October 1994.

[105] T. PUSATERI, *Distance Vector Multicast Routing Protocol version 3 (DVMRP)*, tech. rep., Internet Engineering Task Force, October 2003.

[106] E. C. RYE AND R. BEVERLY, *Discovery the IPv6 Network Periphery*, in Proc. Passive and Active Measurement Conference (PAM), March 2020.

[107] N. SCIENCE FOUNDATION, *A Brief History of NSF and the Internet.* `https://www.nsf.gov/news/news_summ.jsp?cntn_id=103050`. Accessed: July 6, 2021.

[108] B. C. SENEL, M. MOUCHET, J. CAPPOS, O. FOURMAUX, T. FRIEDMAN, AND R. MCGEER, *EdgeNet: A Multi-Tenant and Multi-Provider Edge Cloud*, in Proc. International Workshop on Edge Systems, Analytics and Networking (EdgeSys), April 2021.

[109] Y. SHAVITT AND N. ZILBERMAN, *Geographical Internet PoP Level Maps*, in Proc. Traffic Monitoring and Analysis Workshop (TMA), March 2012.

[110] J. SHERRY, E. KATZ-BASSETT, M. PIMENOVA, H. V. MADHYASTHA, T. ANDERSON, AND A. KRISHNAMURTHY, *Resolving IP Aliases with Prespecified Timestamps*, in Proc. ACM Internet Measurement Conference (IMC), November 2010.

[111] R. SHERWOOD, A. BENDER, AND N. SPRING, *Discarte: a Disjunctive Internet Cartographer*, in Proc. ACM SIGCOMM, August 2008.

[112] R. SHERWOOD AND N. SPRING, *Touring the Internet in a TCP Sidecar*, in Proc. ACM Internet Measurement Conference (IMC), October 2006.

[113] N. SPRING, R. MAHAJAN, AND D. WETHERALL, *Measuring ISP Topologies with Rocketfuel*, in Proc. ACM SIGCOMM, August 2002.

[114] F. TARISSAN, M. LATAPY, P. MÉRINDOL, J.-J. PANSIOT, B. QUOITIN, AND B. DONNET, *Towards Internet Topology Modeling Through Bipartite Graphs*, Computer Networks (COMNET), vol. 57, no. 11, pp. 2331–2347, August 2013.

[115] A. TOONK, *Massive route leak causes Internet slowdown.* `https://bgpmon.net/massive-route-leak-cause-internet-slowdown/`. Accessed: July 7, 2021.

[116] M. E. TOZAL AND K. SARAC, *TraceNET: an Internet Topology Data Collector*, in Proc. ACM Internet Measurement Conference (IMC), November 2010.

[117] ——, *Palm tree: an IP Alias Resolution Algorithm with Linear Probing Complexity*, Computer Communications (COMCOM), vol. 34, no. 5, pp. 658–669, April 2011.

[118] ——, *Subnet Level Network Topology Mapping*, in Proc. IEEE International Performance Computing and Communications Conference (IPCCC), November 2011.

[119] V. JACOBSON ET AL., *traceroute*, man page, UNIX, 1989.
See source code: `ftp://ftp.ee.lbl.gov/traceroute.tar.gz`.

[120] Y. VANAUBEL, P. MÉRINDOL, J.-J. PANSIOT, AND B. DONNET, *Through the Wormhole: Tracking Invisible MPLS Tunnels*, in Proc. ACM Internet Measurement Conference (IMC), November 2017.

[121] Y. VANAUBEL, J.-J. PANSIOT, P. MÉRINDOL, AND B. DONNET, *Network Fingerprinting: TTL-Based Router Signatures*, in Proc. ACM Internet Measurement Conference (IMC), October 2013.

[122] K. VERMEULEN, J. P. ROHRER, R. BEVERLY, O. FOURMAUX, AND T. FRIEDMAN, *Diamond-Miner: Comprehensive Discovery of the Internet's Topology Diamonds*, in Proc. USENIX Symposium on Networked Systems Design and Implementations (NSDI), February 2020.

[123] X. WANG AND D. LOGUINOV, *Understanding and Modeling the Internet Topology: Economics and Evolution Perspective*, IEEE/ACM Transactions on Networking, vol. 18, no. 1, pp. 257–270, February 2010.

[124] D. J. WATTS AND S. H. S.H. STROGATZ, *Collective Dynamics of Small-World Networks*, Nature, vol. 393, pp. 440–442, June 1998.

[125] B. YAO, V. R., F. CHANG, AND D. WADDINGTON, *Topology Inference in the Presence of Anonymous Routers*, in Proc. IEEE INFOCOM, April 2003.

[126] M. ZHANG, Y. RUAN, V. PAI, AND J. REXFORD, *How DNS Misnaming Distorts Internet Topology Mapping*, in Proc. USENIX Annual Technical Conference, May/June 2006.