Contents

Chapter 1

Introduction

This thesis aims at providing a first mathematical formalization of segment routing and showcase several of its use cases. We believe that such a formalization is going to help advance the research of the algorithmic aspects of segment routing by providing a solid mathematical foundation and notations which can serve as a starting point for others to research this topic. The presented use cases validate the practical interest of the deployment of segment routing on existing networks.

Traditional networks use shortest path routing to forward the packets between the routers in the network. Without going into details of shortest path routing, each link in the network has a weight configured on it and shortest paths are computed with respect to these weights. We refer to these weights as interior gateway protocol weights (IGP weights). These paths are used to create a routing table on each router that maps the destination of the packet to an outgoing link interface. In this way, when a packet to `dst` reaches a router, that router will inspect its routing table to find out the link interface towards which it needs to forward that packet. By doing so, the packet will reach `dst` by traversing the shortest path relative to these IGP weights. Figure 1.1 provides a high level illustration of shortest path routing of the Abilene network [1]. When forwarding packets from router `a` to `i`, the shortest path, shown in green, will be traversed. The table on top of router `d` shows an abstraction of its routing table. When `d` receives a packet with destination `i`, it inspects its routing table and sees that the next hop in the IGP shortest path to `i` is `e` so it forwards it there. All of this is an abstraction of what really goes on in a real network because it hides a lot of technical aspects like, for instance, the fact that the information stored is the IP address. However, for this thesis, this view is accurate enough as we focus on mathematical optimization problems related to computer networks rather than compute networks per se.

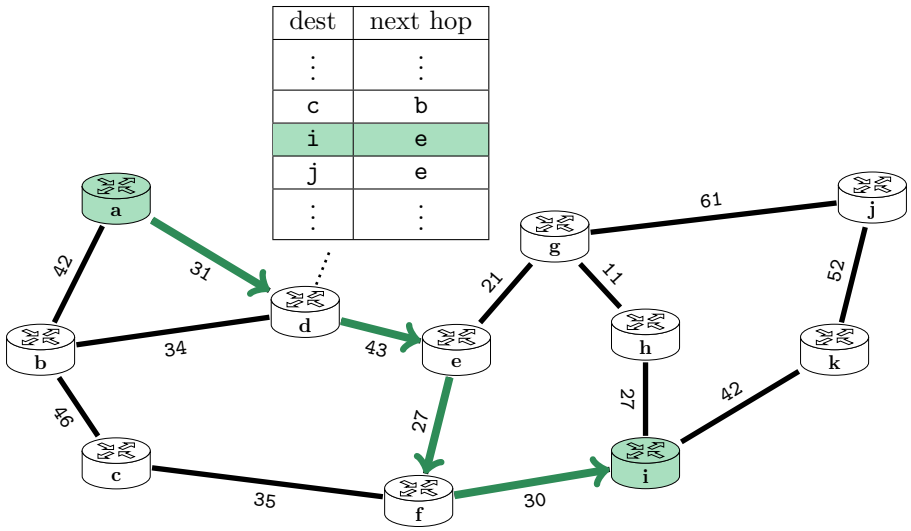


Figure 1.1: Illustration of shortest path routing on the Abilene network.

During recent years, networking vendors and network operators have designed [22,24], implemented [21] and deployed [14,46] a new routing architecture called Segment Routing (SR). Segment Routing is a modern variant of source routing which can be used in either MPLS or IPv6 networks. In a nutshell, Segment Routing allows the source of a packet or the ingress node in a network to easily specify the path that packets need to follow inside the network. This path is specified as a series of labels, called *segments*, (MPLS labels or IPv6 addresses in a special IPv6 header extension) that are added to each packet. These segment are stored as a stack and are popped out as the packets reaches the routers or links they represent. Some implementations do not explicitly remove the segments but rather keep a point to the next segment.

More concretely, a segment routing path is composed of one or more segments. There are two types of segments: (i) *node* and (ii) *adjacency* segments. Node segments represent routers. When a node segment is at the top of the stack, the packet is routed towards the corresponding router using standard shortest path routing. Figure 1.2 shows an example over the Abilene network of what happens when packets are sent from `a` to `i` using segment routing with one intermediate node segment representing node `g`. First, `a` will inspect the segment stack and see a node segment representing router `g`. It will pop this segment and forward the packet using shortest path routing to `g`. Then, node `g` will inspect the packet header and see that the next segment is router `i`. It will pop this segment and forward the packet using shortest path routing to `i`. At this point the segment stack is empty so the packet as arrived to its destination.

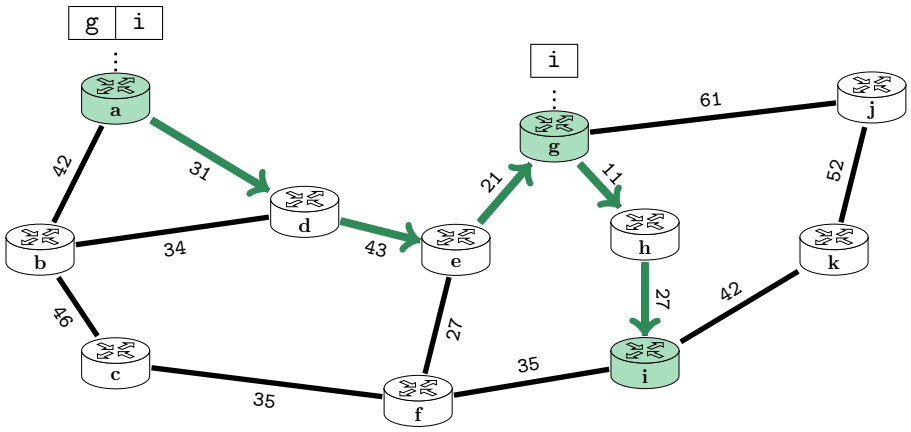


Figure 1.2: Illustration of SR from a to i with a node segment on g.

Adjacency segments represent links. When an adjacency segment is at the top of the stack, the packet is routed towards the origin of the edge corresponding to the link using standard shortest path routing. Then, it is forwarded over that specific link to the router corresponding to the destination of that edge.

Figure 1.3 illustrates an example of what happens when packets are sent from a to i using segment routing with an adjacency segment representing edge (g, j). First, a will inspect the segment stack and see an adjacency segment representing edge (g, j). It will forward the packet using shortest path routing to g. Then, node g will inspect the packet header and see that there is an adjacency segment of which he is the origin. It pops the adjacency segment from the segment stack and sends it over edge (g, j). From there, j will inspect the segment stack and see that in needs to forward that packet towards i.

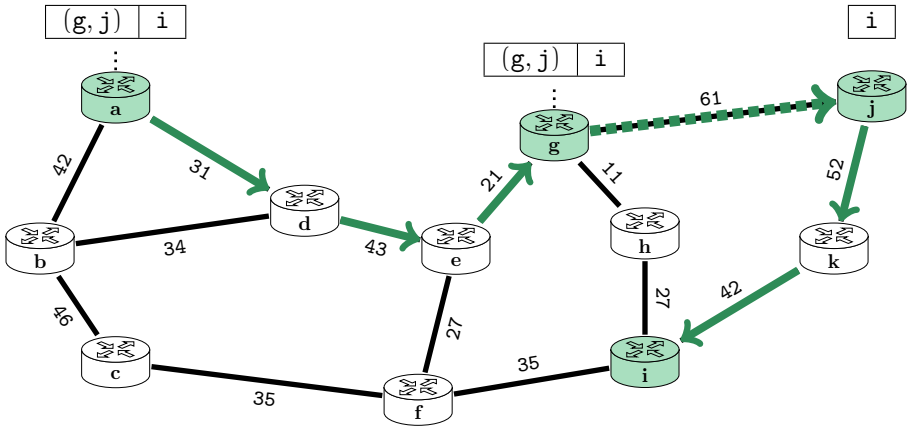


Figure 1.3: Illustration of SR from a to i with an adjacency segment on (g, j).

1.1 Why segment routing

Segments routing offers a lot of flexibility for routing packets by allowing to exploit non shortest path as illustrated in Figures 1.2 and 1.3. It does so by leveraging traditional IP routing so it is easily implementable on current networks without needing to change the way networks operate or their infrastructure. Moreover, there is no need for routers to maintain more state than their, already existing, routing tables. The only overhead is on the packet header which now needs to contain the segment stack.

SR is not the first technique to have been developed that makes it possible to route over non shortest paths. Another technique is to use Multiprotocol Label Switching (MPLS) [28]. In a high level, MPLS works by configuring label forwarding tables on the routers. Conceptually, these tables are similar to the routing tables except that they are not built with respect to the IGP weights. Instead, they map pairs of (label, incoming interface) into other pairs (label, outgoing interface). Figure 1.4 shows an example of how a MPLS configuration can allow us to route packets from `a` to `c` over the non-shortest path $(\mathtt{a}, \mathtt{d}, \mathtt{b}, \mathtt{c})$. In this example, this is achieved by configuring the labeling table of `d` to map packets coming from the interface corresponding to link $(\mathtt{a}, \mathtt{d})$ with label 5 to label 8 and link $(\mathtt{d}, \mathtt{b})$. Second, we configure the labeling table of `b` to map packets coming from link $(\mathtt{d}, \mathtt{b})$ with label 8 to be routed over $(\mathtt{b}, \mathtt{c})$ with the same label.

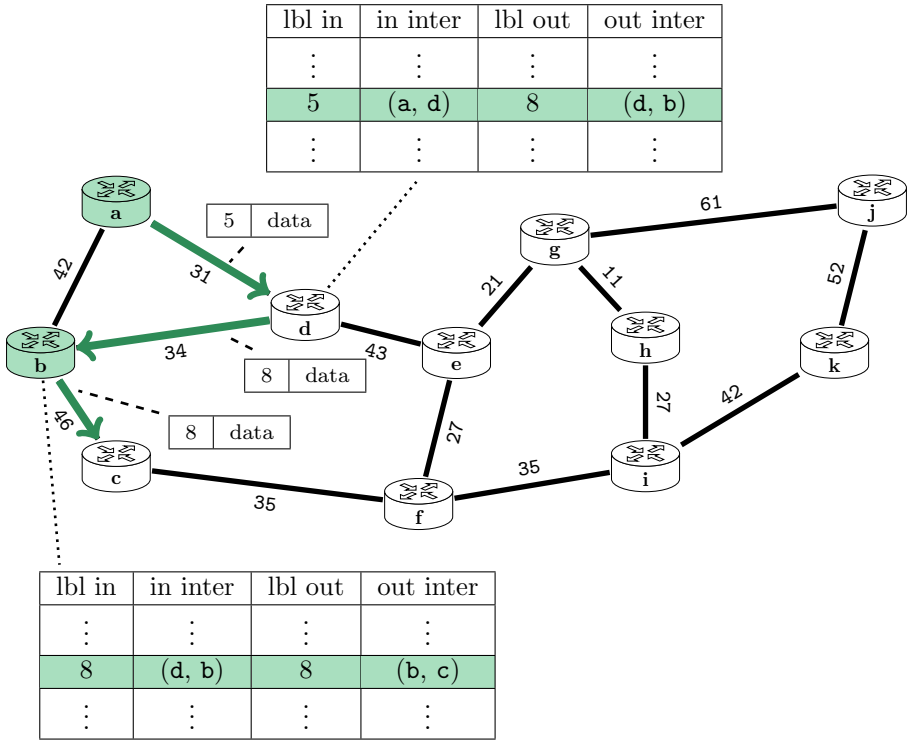


Figure 1.4: Illustration of MPLS.

One drawback of MPLS when compared to segment routing is that the routers need to maintain an extra state, namely, the labeling tables. This makes it harder for solutions based on MPLS to scale has these labeling tables can become huge if a lot of different paths are configured [20,33]. MPLS also has some operational limitations [48] and makes a sub-optimal usage of resources [43].

In contrast, as we mentioned above, with segment routing only the ingress node needs to maintain a state and all the path information is inserted in the packed header as segment. However, one drawback of SR is that, due to hardware limitations, some commercial routers only support a very limited amount of segments [50]. An average low end router can support up to 5 segments whereas high end router can go up to about 10 segments.

Other more recent techniques exist that overcome these limitations as, for instance, Fibbing [52]. Fibbing is an new architecture proposed in 2014 that enables central control over distributed routing. By doing so, similarly to segment routing, it combines the advantages of software defined networks (flexibility, expressiveness, and manageability) and traditional approaches (robustness, and scalability).

The way Fibbing works is that it introduces fake nodes and links into an underlying link-state routing protocol. The fake routers and links are then taken into account by routers when computing their forwarding tables. Figure 1.5 illustrates how to implement the non shortest path $(\texttt{a}, \texttt{d}, \texttt{b}, \texttt{c})$ with fibbing. To do so, we can introduce a fake node `x` connected to `a` and `d`. This fake node advertises that it can reach `c` directly, with a weight of 1. Therefore, from the point of view of `a`, the next hop on the shortest path to `c` is not `b` but rather `x`. We ensure that the table are configured in a way such that when `a` sends packets towards fake node `x`, link $(\texttt{a}, \texttt{d})$ is used. In this way, `a` will send the packets to `d` and then from there routing will be done normally using shortest path $(\texttt{d}, \texttt{b}, \texttt{c})$.

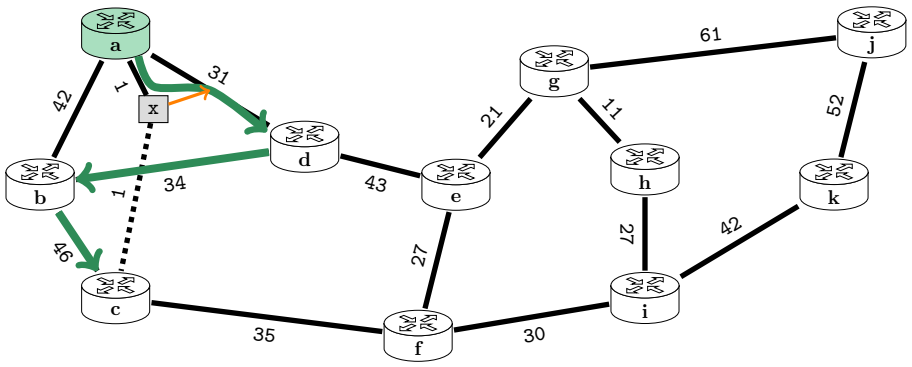


Figure 1.5: Illustration of fibbing on the Abilene network.

There is no particular reason for having chosen segment routing over fibbing other than the fact that we already had started this thesis when we first learned about fibbing. It would be interesting to make a comparative study between the two technologies and better understand their limitations and when is one more suitable than the other.

1.2 SR applications and contributions

In what follows, we give an overview of the uses cases of segment routing that we study in this thesis.

Traffic engineering

One of most successful and widely studied application of SR is traffic engineering (TE). In a nutshell, TE is the problem of routing a set of demands in the most efficient way (avoiding congestion) on a given network. A demand corresponds to a volume of traffic to be routed between two endpoints. Researchers have been exploiting the flexibility of SR to route over a wide range of paths to better balance the link utilisation when routing a large set of demands [11, 26, 34, 35]. We proposed a solution using column generation to find near optimal solutions for the TE problem [40]. A detailed explanation of our solution is given in Chapter 6.

Network monitoring

In Chapter 7 we explain how we leverage segment routing to develop a network monitoring technique to detect single link failures in a network from a single vantage point [7]. Our technique consists of computing a set of cycles that cover every single link of the network and to continuously send probes over those cycles in order to detect when a link goes down. SR is crucial to our technique as it offers the necessary routing flexibility to ensure that we can route the probes over those cycles.

Traffic duplication over disjoint paths

Another application of segment routing that we explored is traffic duplication over disjoint paths. To achieve more efficient communications for low volume traffic, we propose a solution that consists of duplicating this traffic over several disjoint paths [6]. Again, the ability of segment routing to forward traffic over non shortest paths is crucial to forward traffic over disjoint paths.

We also show that we can exploit some properties of segment routing paths to provide disjoint paths that a robust to link failures [8]. Chapter 8 provides an in depth explanation of how this was achieved.

1.3 Publications

A lot of the work presented on this thesis goes well beyond the publications listed below. While writing this thesis we explored each of the problems that we worked on in more depth finding new interesting and yet unpublished results.

- François Aubry, David Lebrun, Yves Deville, Olivier Bonaventure. *Traffic duplication through segmentable disjoint paths* published in IFIP Networking Conference 2015 [6].
- François Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, Olivier Bonaventure. *SCMon: Leveraging segment routing to improve network monitoring* published in IEEE INFOCOM 2016 [7].

- François Aubry, Stefano Vissicchio, Olivier Bonaventure, Yves Deville. *Robustly disjoint paths with segment routing* published in the International Conference on emerging Networking EXperiments and Technologies (CoNEXT) 2018 [8].
- Mathieu Jadin, Fraçois Aubry, Pierre Schaus, Olivier Bonaventure. *CG4SR: Near Optimal Traffic Engineering for Segment Routing with Column Generation* published in IEEE INFOCOM 2019 [40].

1.4 Structure and style

We did our best effort for this work be as self contained as possible while at the time trying to remain focused on what is novel. From a theoretical point of view this means that we define every concept that is used basing ourselves only on very general mathematical concepts such as sets, ordered pairs, sequences and so on. From a practical point of view, we provide a full Java implementation of every algorithm developed in this thesis which is available in the git repository:

`https://github.com/yunoac/thesis`

We also provide a pseudocode formalization of the main algorithms that we propose.

Most of the data that we use is also publicly available as well as the scripts to perform the experiments described in this thesis. The only artifacts that are not available are private ISP topologies that we are not allowed to disclose.

We hope that these choices will make it easier for others to build upon our work and develop new theoretical results and practical applications of segment routing.

The remainder of this thesis is organized as follows. In Chapter 2 we provide the necessary background in graph theory which forms the basic building blocks of everything else in this thesis. We start by giving a graph representation of computer networks which slight differs from the traditional graph definitions. Our model will more easily enable us to model common features of networks such as, for instance, parallel links. Then, we provided a short explanation of the shortest path theory. Since segment routing paths are essentially concatenations of shortest paths, understanding the properties of shortest paths is crucial for understanding segment routing.

In Chapter 3 we provide a description of the dataset that was used on the experimental evaluation of our algorithm. We also analyze some of the properties related to the structure of these networks like, for instance, how dense they are or the path diversity between pairs of routers.

Then, in Chapter 4 we provide, to be best of our knowledge, the first theoretical study of segment routing that fully covers both node and adjacency segments. Here we prove some fundamental theorems which are key to each of the applications of segment routing that we developed in this thesis. We also provide an analysis of segment routing on the topologies of our dataset.

Chapter 5 explores the problem of computing optimal segment routing paths with respect to several different metrics. We will show how to compute segment routing paths of minimum latency which is useful if one wants to use SR to route traffic between two points over a path of minimum latency. As we will see

later, this problem also arises in one form or another as a subproblem of each and every one of the applications that we studied. We also provide an algorithm for computing segment routing paths with maximum bandwidth.

Next, in Chapter 6 we propose a solution for the traffic engineering problem with segment routing based on column generation. We compare our approach with existing ones and show that our solutions are near optimal and provide a better lower bound that traditional techniques based on the multi-commodity flow problem.

In Chapter 7 we present a solution for detecting link failures in a network. Our solution is granular to the point that it can detect link failures within link bundles. We will also show that using the tools developed in Chapter 4, we can provide exact bounds on the minimum number of segments required in any cycle cover of a network.

Finally, we study the problem of computing sets of disjoint paths with SR in Chapter 8. We show that traffic duplication can be used to reduce the latency of network communications. We also exploit properties of segment routing paths to prove pairs of disjoint paths that are robust to any link failure.

1.5 Experimental setting

Apart from the result in chapter 6, all experimental were ran on a DELL Latitude E5450 computer with a Intel Core i5-4310U at 2.00GHz and 8 GB of RAM.

The results from chapter 6 were obtained by running on a computer with 32 CPUs at 2.60GHz, 128GB of RAM. Our solution does not actually need 128GB of RAM but it is able to take advantage of the 32CPUs thanks to the multithreading.

All code is written in Java 1.8 JVM. The mixed integer programming solved that we use is Gurobi v8.0.

Chapter 2

Network and routing model

Introduction

A natural way to provide a mathematical model of a computer network is to represent it with a graph. Graphs consist of a collection of objects called *nodes* and a relation between these objects called *edges*. In the case of computer networks, the nodes correspond to the routers and the edges to the links. Graphs provide an abstraction that captures the connectivity relationships and topological structure of its objects. It abstracts from properties like, say, physical position of the routers.

Such a structure can be augmented with functions on the edges in order to be able to provide a less abstract model and represent some properties of these connections such as the distance between two routers connected by a network link or the bandwidth of the links connecting them.

In this chapter we describe our computer network model and prove the core properties about shortest paths. These properties act as the building blocks for understanding segment routing.

2.1 Graph theory

Definition 2.1. *A* directed graph G *is a pair* (V, E) *where* V *is a finite set and* $\mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$*. The set* V *is called* nodes *of* G *and* E *is called the set of* edges *of* G*. Whenever* G *might not be clear from the context, we will write* $\mathsf{V}(\mathsf{G})$ *and* $\mathsf{E}(\mathsf{G})$ *to represented its set of nodes and edges, respectively.*

Definition 2.2. *A* weighted directed graph *if a triple* $(\mathsf{V}, \mathsf{E}, w)$ *where* (V, E) *is a directed graph and* $w : \mathsf{E} \to \mathbb{R}$ *is a function representing edge weights. In some situations we need to model several different weights on the edges. In this case the weighted graph is represented by a tuple* $(\mathsf{V}, \mathsf{E}, w_1, \ldots, w_k)$ *with each* $w_i : \mathsf{E} \to \mathbb{R}$*.*

As mentioned above, graphs offer a good abstraction for representing a computer network because the main features that we need to care about when designing most optimization algorithms for networking problems consist of which routers are interconnected and the characteristics these connections. Typically

each link is characterized by three numbers: the IGP weight, the capacity (bandwidth) and the latency.

In this work we propose a slightly different model of networks. We add an index on each edge so that we can represent parallel links without having to use multi-sets. These indexes also make it easier, both in theory and in practice, to work with subgraphs and logical operations on these.

Definition 2.3. *A* network *is a tuple* $G = (V, E, \mathsf{igp}, \mathsf{cap}, \mathsf{lat})$ *where* $V = \{0, \dots, n-1\}$ *for some* $n \in \mathbb{N}$ *and* $E \subseteq V \times V \times \mathbb{N}$ *such that*

$$\{i \mid (u, v, i) \in E\} = \{0, \dots, |E| - 1\}.$$

The weight functions are such that $\mathsf{igp} : E \to \mathbb{N}^+$ *represents the IGP weights configured on the links* $\mathsf{cap} : E \to \mathbb{N}$ *represents the links capacities and* $\mathsf{lat} : E \to \mathbb{R}^+$ *represents the link latencies.*

We refer to the set of nodes of G *as* $V(G)$ *and the set of edges as* $E(G)$*. We denote the number of nodes as* $|V(G)|$*, the number of edges as* $|E(G)|$ *and define* $|G| = |V(G)| + |E(G)|$*.*

When the index is not relevant in a given context, we will simplify the notation an omit it. For example, if we want to refer to an arbitrary edge in a network between nodes u and v we will simply write $(u, v) \in E(G)$ instead of $(u, v, i) \in E(G)$. Sometimes it will be useful to be able to refer to all edges between two given nodes. For this, we defined the following.

Definition 2.4. *Let* G *be a network and* $u, v \in V(G)$*. We denote the set of edges between* u *and* v *by*

$$E(G, u, v) = \{(u, v, i) \mid (u, v, i) \in E(G)\}.$$

Whenever G *is clear from the contest we will omit it from the notation.*

What follows is another useful notation that allows to easily refer to the end-point of a given edge in the network.

Definition 2.5. *Let* G *be a network and* $e = (u, v, i) \in E(G)$*. We say that* e *is an edge from* u *to* v *and denote* $e^1 = u$*,* $e^2 = v$ *and* $\mathsf{idx}(e) = i$*.*

Next we define the concept of a *path* in the network. Traditionally, a path is represented by a sequence of nodes that are traversed. So $(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})$ would represent a path from $\mathtt{a}$ to $\mathtt{d}$ that visits $\mathtt{b}$ and $\mathtt{c}$ in that order. However, since we can have parallel links in a network, specifying the nodes that are visited does not fully describe the path. Instead we represent a path by the sequence of edges that are visited.

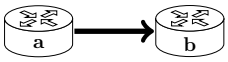
Definition 2.6. *A* path p *in a network* G *is a sequence* $(e_1, \dots, e_l)$ *such that for each* $i = 1, \dots, l$*,* $e_i \in E(G)$ *and if* $i < l$ *then* $e_i^2 = e_{i+1}^1$*.*

We define

$$\mathsf{igp}(p) = \sum_{i=1}^{l} \mathsf{igp}(e_i)$$

and similarly for $\mathsf{cap}(p)$ *and* $\mathsf{lat}(p)$*.*

We say that p *is a path from* u *to* v *if* $e_1^1 = u$ *and* $e_l^2 = v$*.*



A network with a single link between a and b

A network with a bidirectional link between a and b. This means that it has two edges: $(\mathtt{a}, \mathtt{b}, 1)$ and $(\mathtt{b}, \mathtt{a}, 2)$

A network with two bidirectional links between a and b. This means that it has four edges: $(\mathtt{a}, \mathtt{b}, 1)$, $(\mathtt{b}, \mathtt{a}, 2)$, $(\mathtt{a}, \mathtt{b}, 3)$ and $(\mathtt{b}, \mathtt{a}, 4)$

Figure 2.1: In and out neighbors of a node.

If p *is a path ending at* v*, that is, if* $e_l^2 = v$*, then we define* $p \oplus (v, u) = (e_1, \ldots, e_l, (v, u))$ *If* v *is a node and* p *is a path starting at* v *we define* $(v) \oplus p = p$*. If* p *ends at* v *we define* $p \oplus (v) = p$*. This notation will be convenient later on.*

The set of edges of p *is* $E(p) = \{e_1, \ldots, e_l\}$ *and the set of nodes of* p *is* $V(p) = \{e_1^1, e_1^2, \ldots, e_l^1, e_l^2\}$*.*

We say that path p *has* node sequence $(e_1^1, e_1^2, \ldots, e_l^1, e_l^2)$*. A path is said to be* simple *if its node sequence does not contain repeated elements.*

A cycle *is a path such that* $e_1^1 = e_l^2$*.*

For most networks, whenever there is a link connecting a router u to another router v, there usually also is a link connecting router v to router u. To capture these networks we define the concept of symmetric network.

Definition 2.7. *A network* G *is said to be* symmetric *if for every link* (u, v, i) *there exists another link* (v, u, i') *with a bijective correspondence* rev *between links* (u, v, i) *with* $u < v$ *and links* (u, v, i) *with* $u > v$*.*

In general we do not assume that the IGP weights are the same in both directions, that is, we can have some $e \in E(G)$ in a symmetric network such that $\mathsf{igp}(e) = \mathsf{igp}(\mathsf{rev}(e))$. However, in some contexts we will need this assumption. Whenever this is the case, that is, whenever for each $e \in E(G)$, $\mathsf{igp}(e) \neq \mathsf{igp}(\mathsf{rev}(e))$ we say that igp is *symmetric*.

Before we proceed, we want to make a comment about the conventions used to represent networks in the figures used in this thesis. In most examples (if not all), we assume that the network is symmetric with unit IGP weights. When this is the case, we draw a single line between u and v representing both these edges. If several parallel links exists, then several such lines will be drawn. Figure 2.1 illustrates this.

To represent a specific link or a path in the network, we draw colored directed edges on top the lines representing the link.

By default, numbers above links represent IGP weights. As we said, in most examples we use unit IGP weights on every link, meaning that shortest paths are measured in terms of the number of links on the path. In this case we omit the weight to lighten the figures. Whenever this is not the case, the numbers on

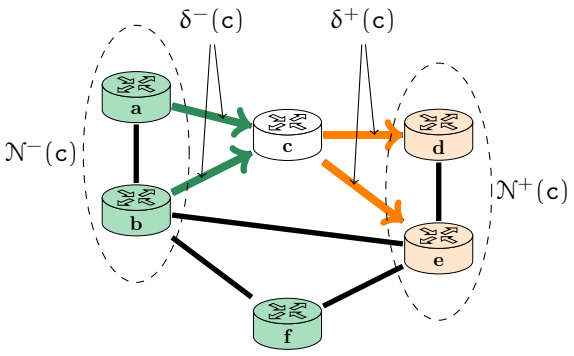


Figure 2.2: In and out neighbors of a node.

top of the links represent the IGP weight. If we want to represent some other link metric, it will be explicitly stated in the example what the numbers represent. This closes the parenthesis about graphical representations of networks in this work.

The following definitions make it easy to refer to the neighbors of a given node. We define the set of edges going into and out of a node as well as the sets of neighbors consisting of their endpoints.

Definition 2.8. *Let* G *be a network and* $v \in V(G)$*. We define*

i) $\delta^+(G, v) = \{(v, u, i) \mid (v, u, i) \in E(G)\}$

ii) $\delta^-(G, v) = \{(u, v, i) \mid (u, v, i) \in E(G)\}$

iii) $\mathcal{N}^+(G, v) = \{u \mid (v, u, i) \in E(G)\}$

iv) $\mathcal{N}^-(G, v) = \{u \mid (u, v, i) \in E(G)\}$

When the underlying graph G is clear from context, we omit it to lighten the notation. Figure 2.2 illustrates this definition.

We conclude this section with a definition of subnetwork.

Definition 2.9. *Let* G, H *be two networks. If* $V(G) = V(H)$ *and* $E(H) \subseteq E(G)$ *we say that* H *is a* subnetwork *of* G *and write* $H \subseteq G$*. The weights* $\mathsf{igp}, \mathsf{lat}$ *and* cap *are defined on* H *by restricting them to* $E(H)$*.*

Traditionally subgraphs are defined in a way that also restricts the set of nodes in the result. More precisely, the set of nodes is often restricted to the set of nodes that are the endpoints of some edge in the subgraph. We find it more convenient to keep the same set of nodes as this is more in line with how the algorithms proposed in this thesis are implemented.

2.2 Shortest paths

As mentioned in the introduction, segment routing leverages shortest path routing in order to forward packets over the network. For this reason, understanding

shortest paths is central to understanding segment routing. In this section we provide an overview of the most important properties shortest paths and of the existing algorithms used to compute such paths.

Unless stated otherwise, whenever we refer to a shortest path in a given network G, we mean a shortest path with respect to the igp weight function.

Definition 2.10. *Let* G *be a network and* $\mathtt{s}, \mathtt{t} \in \mathsf{V}(\mathsf{G})$*. A shortest path between* $\mathtt{s}$ *and* $\mathtt{t}$ *is a path* $\mathtt{p}$ *from* $\mathtt{s}$ *to* $\mathtt{t}$ *such that* $\mathsf{igp}(\mathtt{p})$ *is minimum.*

Shortest paths are not necessarily unique. When several shortest paths exist between two nodes we say that there is Equal-cost multi-path (ECMP) between those nodes. In order to characterize the set of network links where packets might pass when forwarded from a given network router, we need to be able to refer to the subgraph containing all shortest paths starting from a given node in the network. This concept is formalized with the following definition.

Definition 2.11. *Let* G *be a network and* $\mathtt{r} \in \mathsf{V}(\mathsf{G})$*. We define the* shortest path subnetwork rooted at $\mathtt{r}$ *as* $\mathsf{SP}(\mathsf{G}, \mathtt{r}) = (\mathsf{V}, \mathsf{E}')$ *where* $\mathtt{e} \in \mathsf{E}'$ *if there exists* $\mathtt{v} \in \mathsf{V}(\mathsf{G})$ *and a shortest path* $\mathtt{p}$ *from* $\mathtt{r}$ *to* $\mathtt{v}$ *such that* $\mathtt{e} \in \mathsf{E}(\mathtt{p})$*. Whenever* G *is clear from the context, we omit it from the notation and simply write* $\mathsf{SP}(\mathtt{r})$*.*

Definition 2.12. *Let* G *be a network. We say that* G *is* acyclic *if it contains no cycles.*

It is not hard to see that the set of all shortest path between starting from a given node is acyclic.

Lemma 2.1. *Let* G *be a network and* $\mathtt{r} \in \mathsf{V}(\mathsf{G})$*. Then* $\mathsf{SP}(\mathtt{r})$ *is an acyclic subnetwork of* G*.*

Proof. Suppose that there exists a cycle $\mathtt{c} = (\mathtt{e}_1, \ldots, \mathtt{e}_l)$ on $\mathsf{SP}(\mathtt{r})$. Then, by definition there exist shortest paths $\mathtt{p}_1, \ldots, \mathtt{p}_l$ such that $\mathtt{p}_i$ is a shortest path starting at $\mathtt{r}$ containing $\mathtt{e}_i$. For each $\mathtt{i}$, write $\mathtt{e}_i = (\mathtt{u}_i, \mathtt{v}_i)$, $w_i = \mathsf{igp}(\mathtt{p}_1) - \mathsf{igp}(\mathtt{e}_i)$ and $\mathtt{j}_i$ such that

$$\mathtt{j}_i = \begin{cases} i+1 & \text{if } i+1 < l \\ 1 & \text{if } i = l \end{cases}$$

Before we proceed with the general case, let's see the proof in an example with three nodes. So suppose that $\mathtt{c} = (\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3) = ((\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{d}))$ as shown in Figure 2.3. By definition, $w_1 + \mathsf{igp}((\mathtt{a}, \mathtt{b}))$ is the cost of a shortest path from $\mathtt{r}$ to $\mathtt{b}$. Since w_2 is also the cost of a path from $\mathtt{r}$ to $\mathtt{b}$ we have that $w_1 + \mathsf{igp}((\mathtt{a}, \mathtt{b})) \leqslant w_2$. Since $\mathsf{igp}((\mathtt{a}, \mathtt{b})) > 0$ we conclude that $w_1 < w_2$. In the same way we can see that $w_2 < w_3$ and $w_3 < w_1$ leading to the contradiction that $w_1 < w_1$.

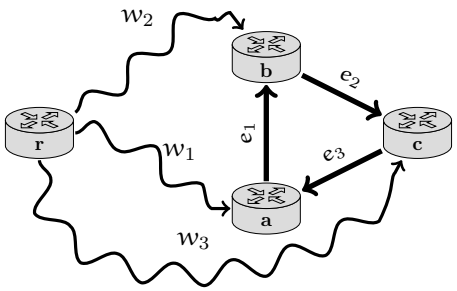


Figure 2.3: Proof intuition on a cycle with 3 edges. Wiggly arrows represent shortest path.

In the general case, we have that $w_i + \mathsf{igp}(e_i) \leqslant w_j$ because $w_i + \mathsf{igp}(e_i)$ is by definition the cost of the shortest path from r to v_i and w_j is the cost of a path from r to v_i. Since for each i, $\mathsf{igp}(w_i) > 0$ we conclude that $w_i < w_j$. As this is true for all i we have $w_1 < w_2 < \ldots < w_l < w_1$ which is impossible. Therefore such a cycle cannot exist. □

Definition 2.13. *Given a network* G *we denote the shortest path distance (minimum IGP cost) between two nodes by* $d(G, u, v)$ *or simply* $d(x, y)$ *if* G *is clear from the context.*

Note that with the above definition, p is a shortest path from u to v if and only if $\mathsf{igp}(p) = d(u, v)$.

One important property of shortest paths that we will use often when proving properties of segment routing is that a sub-path of a shortest path is also a shortest path.

Proposition 2.2. *Let* G *be a network and* $p = (e_1, \ldots, e_l)$ *a shortest path on* G*. Then for each* $i, j \in \{1, \ldots, l\}$ *with* $i \leqslant j$*,* $(e_i, \ldots, e_j)$ *a shortest path.*

Proof. Let G be a network and $p = (e_1, \ldots, e_l)$ a shortest path on G and $i, j \in \{1, \ldots, l\}$ with $i \leqslant j$. Let $v = e_i^1$ and $u = e_j^2$. Suppose that $(e_i, \ldots, e_j)$ is not a shortest path. There there exists another shorter path $q = (f_1, \ldots, f_r)$ from u to v. Consider $p' = (e_1, \ldots, e_{i-1}, f_1, \ldots, f_r, e_{j+1}, \ldots, e_l)$. Since q is a path from u to v and since $u = f_i^1 = e_i^1 = e_{i-1}^2$ and $v = f_r^2 = e_j^2 = e_{j+1}^1$, we have that p' is also a path from u to v. This path is shorter than p since

$$
\begin{aligned}
\mathsf{igp}(p') &= \sum_{k=1}^{i-1} \mathsf{igp}(e_k) + \sum_{k=1}^{r} \mathsf{igp}(f_k) + \sum_{k=j+1}^{l} \mathsf{igp}(e_i) && \text{[definition of igp}(p')\text{]} \\
&= \sum_{k=1}^{i-1} \mathsf{igp}(e_k) + \mathsf{igp}(q) + \sum_{k=j+1}^{l} \mathsf{igp}(e_i) && \text{[definition of igp}(q)\text{]} \\
&< \sum_{k=1}^{i-1} \mathsf{igp}(e_k) + \sum_{k=i}^{j} \mathsf{igp}(e_k) + \sum_{k=j+1}^{l} \mathsf{igp}(e_i) && \text{[q is shorter than } (e_i, \ldots, e_j)\text{]} \\
&= \sum_{k=1}^{l} \mathsf{igp}(e_k) = \mathsf{igp}(p) && \text{[definition of igp}(p)\text{]}
\end{aligned}
$$

This contradicts the fact that p is a shortest path. □

The shortest path subgraphs are easily characterized by the values of $d(u, v)$ as shown in the following proposition.

Proposition 2.3. *Let* G *be a network* $v, u \in V(G)$ *and* $e \in E(G)$. *Then*

i) There exists a shortest path containing e *if and only if* $d(e^1, e^2) = igp(e)$.

ii) Edge e *belongs to a shortest path from* v *to* e^2 *if and only if* $d(v, e^2) = d(v, e^1) + igp(e)$.

iii) There exists a shortest path from v *to* u *containing edge* e *if and only if* $d(v, e^1) + igp(e) + d(e^2, u) = d(v, u)$.

Proof. Let G be a network $v, u \in V(G)$ and $e \in E(G)$.

i) ($\Rightarrow$) Suppose that there exists a shortest path $p = (e_1, \dots, e_l)$ containing edge e. Then $e = e_i$ for some i and by Proposition 2.2 the sub-path (e_i) of p is a shortest path. Thus $d(e^1, e^2) = igp(e)$.

($\Leftarrow$) If $d(e^1, e^2) = igp(e)$ then by definition $p = (e)$ is a shortest path.

ii) ($\Rightarrow$) Suppose that e belongs to a shortest path $p = (e_1, \dots, e_l)$ from v to y. Since p is a shortest path we must have $e_l = e$ or otherwise, since IGP weights are positive, p would contain a positive cycle. Therefore

$$d(v, e^2) = \sum_{i=1}^{l} igp(e_i) = \sum_{i=1}^{l-1} igp(e_i) + igp(e)$$

By Proposition 2.2, $(e_1, \dots, e_{l-1})$ is a shortest path between v and e^1 and thus

$$\sum_{i=1}^{l-1} igp(e_i) = d(v, e^1)$$

showing that $d(v, e^2) = d(v, e^1) + igp(e)$.

($\Leftarrow$) Suppose that $d(v, e^2) = d(v, e^1) + igp(e)$ and let $p = (e_1, \dots, e_l)$ be a shortest path from v to e^1. Then $p \oplus e = (e_1, \dots, e_l, e)$ has cost $igp(p) = d(v, e^1) + igp(e) = d(v, e^2)$ showing that $p \oplus e$ is a shortest path between e^1 and e^2.

iii) ($\Rightarrow$) Suppose that there exists a shortest path $p = (e_1, \dots, e_l)$ from v to u containing edge e. Assume that $e = e_i$ for some $i \in \{1, \dots, l\}$. Then

$$\begin{aligned}
d(v, u) &= igp(p) && \text{[p is shortest path from } v \text{ to } u\text{]} \\
&= igp((e_1, \dots, e_{i-1})) + igp(e_i) + igp((e_{i+1}, \dots, e_l)) && \text{[def of igp(p)]} \\
&= igp((e_1, \dots, e_{i-1})) + igp(e) + igp((e_{i+1}, \dots, e_l)) && [e_i = e] \\
&= d(v, e^1) + igp(e) + d(e^2, u) && \text{[sub-paths are shortest paths]}
\end{aligned}$$

($\Leftarrow$) Suppose that $d(v, e^1) + igp(e) + d(e^2, u) = d(v, u)$. Let p be a shortest path from v to e^2 and q be a shortest path from e^2 to u. Then $p \oplus e \oplus q$ is a path from v to u containing edge e of cost $d(v, e^1) + igp(e) + d(e^2, u)$. Since $d(v, e^1) + igp(e) + d(e^2, u) = d(v, u)$ it follows that $p \oplus e \oplus q$ is a shortest path from v to u.

□

Next, we are going to prove some properties about shortest paths on symmetric networks.

Definition 2.14. *Let* G *be a symmetric network and* $\mathsf{X} \subseteq (\mathsf{E}(\mathsf{G}))$*. We define*

$$\mathsf{rev}(\mathsf{X}) = \{\mathsf{rev}(\mathsf{e}) \mid \mathsf{e} \in \mathsf{X}\}.$$

Definition 2.15. *Let* G *be a symmetric network and* $\mathsf{p} = (\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{l})$ *a path on* G*. We define*

$$\mathsf{rev}(\mathsf{p}) = (\mathsf{rev}(\mathsf{e}_\mathsf{l}), \ldots, \mathsf{rev}(\mathsf{e}_1)).$$

Lemma 2.4. *Let* G *be a symmetric network* $\mathsf{u}, \mathsf{v} \in \mathsf{V}(\mathsf{G})$ *and* $\mathsf{p} = (\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{l})$ *be a path on* G *from* u *to* v*. If* igp *is symmetric then* $\mathsf{rev}(\mathsf{p})$ *is a path from* v *to* u *with* $\mathsf{igp}(\mathsf{p}) = \mathsf{igp}(\mathsf{rev}(\mathsf{p}))$*.*

Proof. Let $\mathsf{i} \in \{\mathsf{l}, \ldots, 2\}$. Since p is a path, we have $\mathsf{e}^2_{\mathsf{i}-1} = \mathsf{e}^1_\mathsf{i}$. Thus $\mathsf{rev}(\mathsf{e}_\mathsf{i})^2 = \mathsf{e}^1_\mathsf{i} = \mathsf{e}^2_{\mathsf{i}-1} = \mathsf{rev}(\mathsf{e}_{\mathsf{i}-1})^1$ so that $\mathsf{rev}(\mathsf{p})$ is a path. It starts at $\mathsf{rev}(\mathsf{e}_\mathsf{l})^1 = \mathsf{e}^2_\mathsf{l} = \mathsf{v}$ and ends at $\mathsf{rev}(\mathsf{e}_1)^2 = \mathsf{e}^1_1 = \mathsf{u}$. Finally, by definition, we have that $\mathsf{igp}(\mathsf{e}_\mathsf{i}) = \mathsf{igp}(\mathsf{rev}(\mathsf{e}_\mathsf{i}))$ for each i so $\mathsf{igp}(\mathsf{p}) = \mathsf{igp}(\mathsf{rev}(\mathsf{p}))$. □

Definition 2.16. *Let* G *be a symmetric network and* $\mathsf{H} \subseteq \mathsf{G}$*. We define* $\mathsf{rev}(\mathsf{H}) = (\mathsf{V}, \mathsf{rev}(\mathsf{E}(\mathsf{H})))$*.*

Lemma 2.5. *Let* G *be a symmetric network,* $\mathsf{u}, \mathsf{v} \in \mathsf{V}(\mathsf{G})$ *and* p *a shortest path from* u *to* v*. If* igp *is symmetric then* $\mathsf{rev}(\mathsf{p})$ *is a shortest path from* v *to* u*.*

Proof. We already from Lemma 2.4 that $\mathsf{rev}(\mathsf{p})$ is a path from v to u with $\mathsf{igp}(\mathsf{p}) = \mathsf{igp}(\mathsf{rev}(\mathsf{p}))$. Suppose that it is not a shortest path. Then there exists a path q from v to u such that $\mathsf{igp}(\mathsf{q}) < \mathsf{igp}(\mathsf{rev}(\mathsf{p}))$. Therefore,

$$\mathsf{igp}(\mathsf{rev}(\mathsf{q})) = \mathsf{igp}(\mathsf{q}) < \mathsf{igp}(\mathsf{rev}(\mathsf{p})) = \mathsf{igp}(\mathsf{p})$$

contradicting the fact that p is a shortest path from u to v since $\mathsf{rev}(\mathsf{q})$ is also a path from u to v. □

Corollary 2.6. *Let* G *be a symmetric network and* $\mathsf{u}, \mathsf{v} \in \mathsf{V}(\mathsf{G})$*. If* igp *is symmetric then* $\mathsf{SP}(\mathsf{u}, \mathsf{v}) = \mathsf{rev}(\mathsf{SP}(\mathsf{v}, \mathsf{u}))$*.*

Proof. By definition, $\mathsf{V}(\mathsf{SP}(\mathsf{u}, \mathsf{v})) = \mathsf{V}(\mathsf{rev}(\mathsf{SP}(\mathsf{v}, \mathsf{u})))$. Hence we need to show that $\mathsf{E}(\mathsf{SP}(\mathsf{u}, \mathsf{v})) = \mathsf{E}(\mathsf{rev}(\mathsf{SP}(\mathsf{v}, \mathsf{u})))$. Let $\mathsf{e} \in \mathsf{E}(\mathsf{SP}(\mathsf{u}, \mathsf{v}))$. By definition of $\mathsf{SP}(\mathsf{u}, \mathsf{v})$, there exists a shortest path p from u to v. By Lemma 2.5 $\mathsf{rev}(\mathsf{p})$ is a shortest path from v to u that contains e. Hence $\mathsf{E}(\mathsf{rev}(\mathsf{p})) \subseteq \mathsf{SP}(\mathsf{v}, \mathsf{u})$. By definition of $\mathsf{rev}(\mathsf{p})$, there must be $\mathsf{e}' \in \mathsf{rev}(\mathsf{p})$ such that $\mathsf{e} = \mathsf{rev}(\mathsf{e}')$. Therefore, since $\mathsf{e}' \in \mathsf{SP}(\mathsf{v}, \mathsf{u})$, $\mathsf{e} = \mathsf{rev}(\mathsf{e}') \in \mathsf{rev}(\mathsf{SP}(\mathsf{v}, \mathsf{u}))$. We conclude that $\mathsf{E}(\mathsf{SP}(\mathsf{u}, \mathsf{v})) \subseteq \mathsf{E}(\mathsf{rev}(\mathsf{SP}(\mathsf{v}, \mathsf{u})))$. The proof that $\mathsf{E}(\mathsf{SP}(\mathsf{u}, \mathsf{v})) \supseteq \mathsf{E}(\mathsf{rev}(\mathsf{SP}(\mathsf{v}, \mathsf{u})))$ is analogous. □

Conclusion

In this chapter we have defined the graph definition that we use throughout this thesis and proved the basic properties of shortest path adapted to our definitions. These properties will be fundamental in the remainder of this thesis as we build upon them to develop our segment routing theory and prove our results.

Chapter 3

Topology dataset analysis

Introduction

In this chapter we provide a brief description of the dataset used in the experimental evaluation of this thesis.

We use four groups of topologies: the RocketFuel topologies [47] which contains 6 topologies, the topologies from the topology Zoo [37] which consists of 520 topologies of and 3 private ISP topologies and a topology of the OVH-Europe network [2]. The later topology is important because if contains a lot of link bundles (parallel links) which is a property that is often ignored in the algorithmic community. A lot of graph algorithms assume that the graphs are simple and therefore there are not a lot of empirical results over these kinds of topologies.

These topologies were cleaned so that each of them contains only one strongly connected component, meaning that there is at least one path between each pair of nodes in each topology. This is a simple cleaning step that is necessary because it does not make sense to have a network topology where some nodes cannot communicate with each other.

3.1 Topology sizes

Tables 3.1, 3.2 and 3.3 show the sizes of the topologies in each of the groups considered in this thesis. For the `zoo` group we only show the largest topologies since on one hand those are the most relevant for our evaluation and the group contains too many topologies allows for a complete listing.

In table 3.4 we show the minimum and maximum sizes of the topologies in our dataset and table 3.5 shows the percentage of topologies whose number nodes lies in different sizes. This shows that we tackle a wide range of topologies of realistic sizes.

3.2 ECMP and non shortest path links

We mentioned in the introduction that segment routing supports two kinds of segments: node segments and adjacency segments. We will see that adjacency

Topology name	$\|V(G)\|$	$\|E(G)\|$
AS 1221	104	302
AS 1239	315	1944
AS 1755	87	322
AS 3257	161	656
AS 3967	79	294
AS 6461	138	744

Figure 3.1: Topology sizes in the `rf` group.

Topology name	$\|V(G)\|$	$\|E(G)\|$
ISP 1	≈ 150	≈ 700
ISP 2	≈ 220	≈ 800
ISP 3	≈ 170	≈ 440
OVH	57	402

Figure 3.2: Topology sizes in the `real` group and `ovh`.

Topology name	$\|V(G)\|$	$\|E(G)\|$
ITZ Cogentco	197	490
ITZ Colt	153	382
ITZ Deltacom	113	366
ITZ Dia	138	302
ITZ GtsCe	149	386
ITZ Interoute	110	312
ITZ Ion	125	300
ITZ Tata	145	388
ITZ UsCarrier	158	378

Figure 3.3: Largest topologies in the `zoo` group.

Min $\|V(G)\|$	Max $\|V(G)\|$	Min $\|E(G)\|$	Max $\|E(G)\|$
4	315	8	1955

Figure 3.4: Minimum and maximum topology sizes.

Small	Medium	Large	Huge
$[0, 20]$	$]20, 50]$	$]50, 100]$	> 100
30%	43%	21%	6%

Figure 3.5: Number of topologies by group size.

Links not in shortest paths	Pairs with ECMP
1%	26%

Figure 3.6: Percentage of links not in shortest paths and pairs with ECMP.

segments are more costly and thus we would like to avoid using them whenever possible. However, will see later that adjacency segments in segment routing can we necessary to implement paths that belong to ECMP components or that traverse links that do not belong to any IGP shortest path. For that reason we analyzed the amount of ECMP component that exist in out dataset as well as the amount of edges that do not belong to any shortest path. These values will give an estimation of how important supporting adjacency segments is. Figure 3.6 shows the percentage of links that do not belong to any shortest path as well as the percentage of pairs of nodes with ECMP between them. We can see that, as expected, IGP weights are configured so that most links are used for shortest path routing, we have only 1% of the links falling outside of that. We expect that mainly backup links will be configured so that they are not used unless a failure occurs. On the other hand, we see that there is a relatively high percentage of pairs of nodes with ECMP. This indicates that adjacency segments will be useful whenever we want to represent specific paths that traverse ECMP components.

We also analyzed how many equal cost paths exist. Figure 3.6 shows the CDF of the total number of shortest paths between each pair of nodes over all topologies. This shows that some nodes are connected with a very high number of shortest paths but that most nodes have at most 10 shortest paths between them.

3.3 Degrees and density

We also analyzed the degrees of the nodes in the topologies in our dataset as well as the edge densities. The degree of a router represents the number of routers to which it tis connected to. It represents an upper bound on the number of edge-disjoint paths that can be used to connect a given router to other routers. Figure 3.8 shows a box plot of these. We can observe that some nodes have a very high degree but that the tendency is that most nodes have a degree lower than 10. Nodes of degree 1 are problematic in a network because it means that there is a single point of failure to reach them. We observe that the largest topologies are the ones that suffer the least from this problem.

One degree nodes probably exist on these topologies due to the fact that most of them are were collected using inexact methods thus leading to incomplete data. A computer network should at least be biconnected (have at least two disjoint paths between every pair of nodes) to prevent a single link failure from partitioning it.

The edge density of a network evaluates how close a network is from a complete graph. It is defined for graphs with no parallel links as

$$\frac{|\mathsf{E}(\mathsf{G})|}{|\mathsf{V}(\mathsf{G})| \cdot (|\mathsf{V}(\mathsf{G})| - 1)}.$$

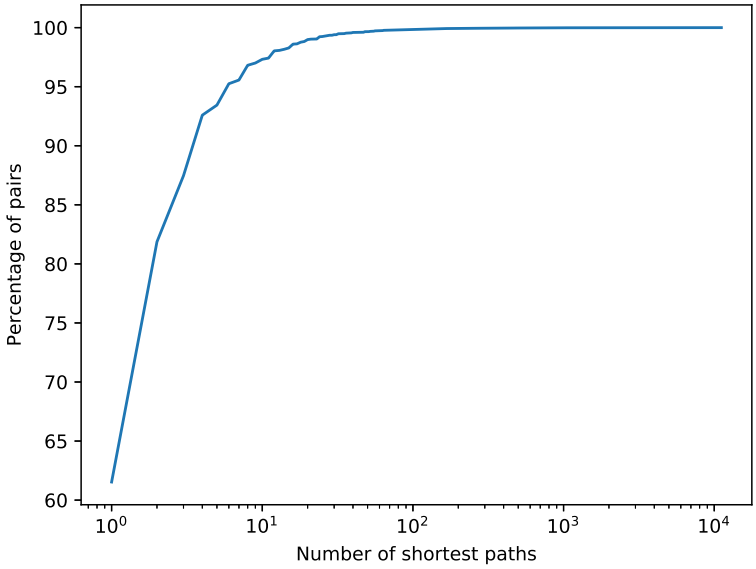


Figure 3.7: CDF over all topologies and all pairs of nodes of the number of shortest paths between those nodes.

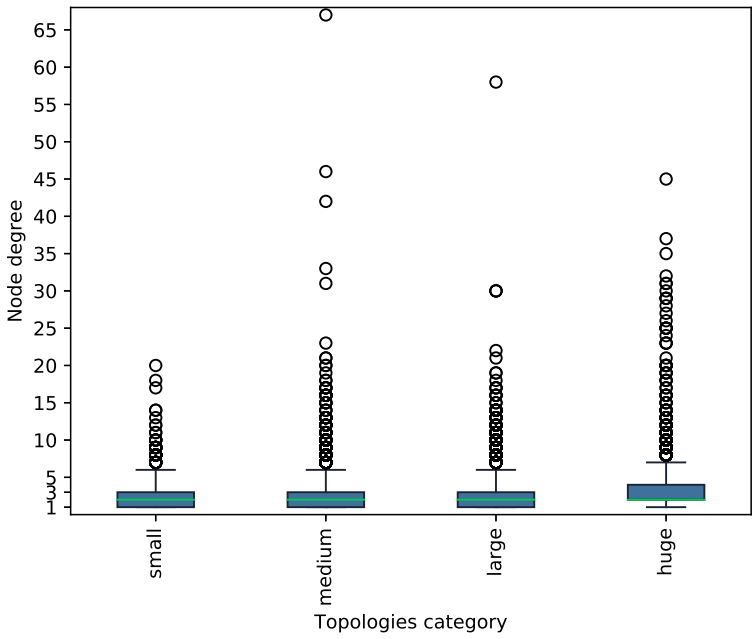


Figure 3.8: Box plot of the degrees over the different size categories of our topologies.

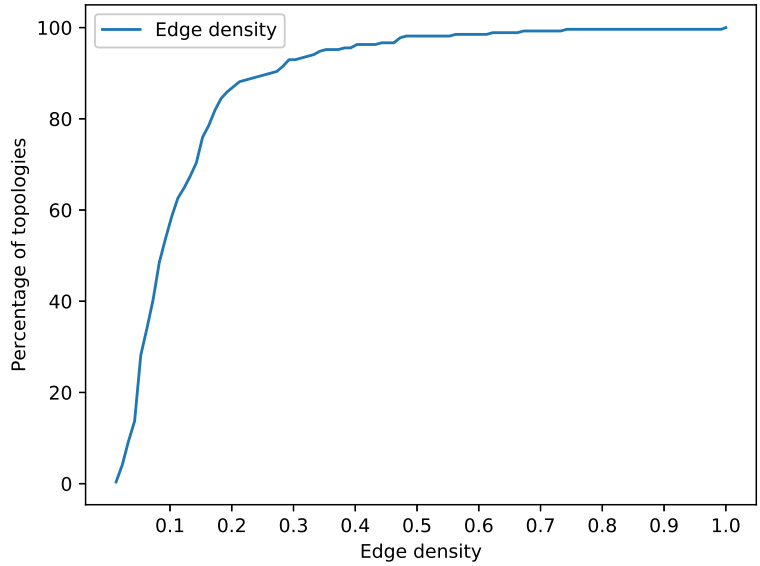


Figure 3.9: Box plot of the degrees over the different size categories of our topologies.

A tree is the lowest density connected network that one can have. Any link failure in a tree will cause the network to become disconnected. High density network are costly but are more robust to failures. They also provide a higher path diversity so optimal solution of network optimization problems tend to be better on dense networks. Figure 3.9 shows a CDF of the densities over all topologies in our dataset. We used the above formula even though some of our networks contain parallel links. We can see that some network have a very low density. About 60% percent of the network topologies have an edge density of at least 10% and 20% of the networks have an edge density above 20%.

3.4 Connectivity

We saw that some pairs of nodes have a lot of shortest paths between them. However these paths are of course not disjoint. In this section we analyze how well the nodes are connected in the network. To measure this, we compute for each pair of nodes of each topology the minimum number of links that need to be removed from the network in order to disconnect those nodes. This is known in graph theory as the minimum cut between the nodes [4]. Figure 3.10 shows a CDF of these minimum cuts. It is not hard to see that the minimum cut between two nodes is the same as the maximum number of edge-disjoint paths between them [4].

We can see that about 40% of the pairs of nodes are connected by a single path as we had already observed above. This is quite low for a network as it does not offer a lot of redundancy and link failures can easily disconnect the nodes. The remaining nodes require the removal of at least 2 links to disconnect

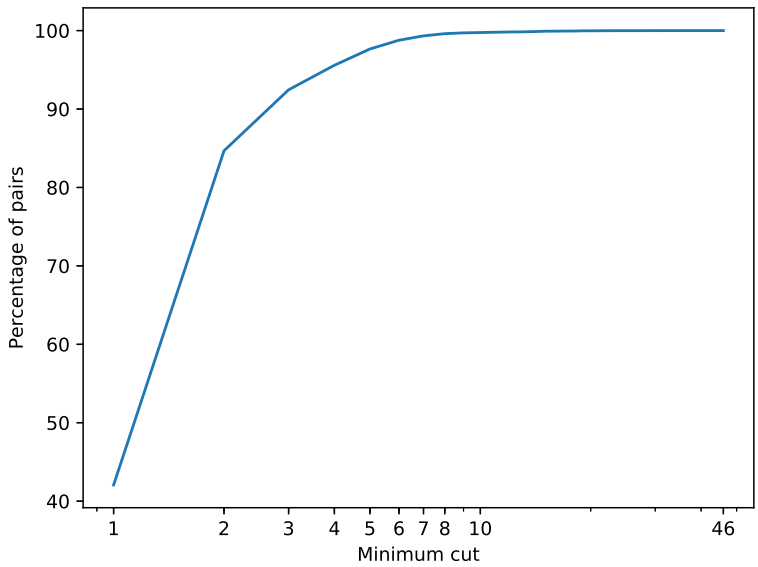


Figure 3.10: CDF over all topologies and all pairs of nodes of the number of shortest paths between those nodes.

them (in other words, they are in the same biconnected component [12]). Also, nodes are very well connected having up to 46 edge-disjoint paths connecting them.

Conclusion

From this evaluation we observe that there is probably missing data on some of the topologies used in this thesis as some of them are weakly connected. We have also seen that due to a high amout of ECMP (26%) between pairs of nodes, adjacency segments are likely to be necessary to implement paths in the graph topology with segment routing.

Chapter 4

Segment routing

Introduction

As mentioned in the introduction, segment routing [23] is a new forwarding architecture that is being developed within the Internet Engineering Task Force and network operators. Segment Routing changes the way packets are forwarded inside a network to enable network operators to have better control on the path followed by the packets. Traffic can be forced to follow a series of detours which can either correspond to passing by a specific router or network link.

This chapter is dedicated to formalizing segment routing. We provide, to the best of our knowledge, the first formalization that comprises both node and adjacency segments. We define minimal segmentations and provide an efficient algorithm for computing them. We also provide reachability concepts which allow to analyze the capability of a given network topology to support segment routing as well as giving lower bound on the minimum number of segments needed reach every single link in the network. These concepts will be fundamental later on when we propose an algorithm for computing cycle covers of a network.

4.1 Segment routing formalization

The starting point of our segment routing formalization is the concept of segment routing path, or sr-path for short.

Definition 4.1. *Let* G *be a network. A* sr-path $\vec{\mathsf{p}}$ *on* G *is a sequence* $\langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle$ *such that each* $\mathsf{x}_\mathsf{i} \in \mathsf{V}(\mathsf{G}) \cup \mathsf{E}(\mathsf{G})$.

We represent sr-paths with the vector notation to be able to more easily distinguish between a path p and a sr-path $\vec{\mathsf{p}}$ in the network G. In practice, the elements of $\vec{\mathsf{p}}$ that are nodes model node segments and the elements of $\vec{\mathsf{p}}$ that are edges model adjacency segments. It is important to understand that, because of ECMP, a sr-path actually can correspond to a set of paths in the original network.

Consider the sr-path $\vec{\mathsf{p}} = \langle \mathtt{a}, \mathtt{c}, \mathtt{e}, (\mathtt{f}, \mathtt{j}), \mathtt{i} \rangle$ shown in Figure 4.1. The solid green edges show the set of edges that belong to shortest paths between consecutive segments and the dashed green edge represents the adjacency segment

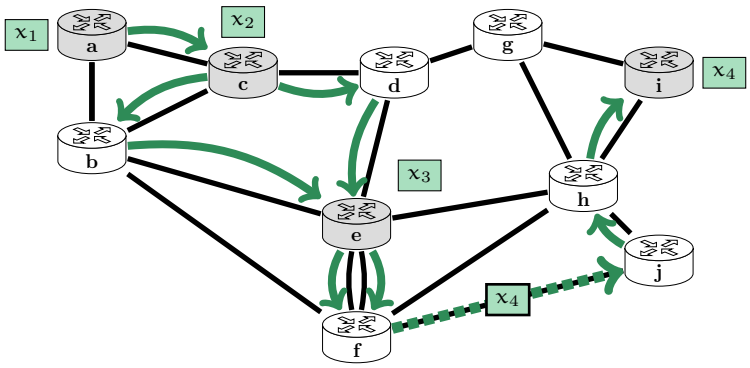


Figure 4.1: Illustration of sr-path $\vec{p} = \langle \mathtt{a}, \mathtt{c}, \mathtt{e}, (\mathtt{f}, \mathtt{j}), \mathtt{i} \rangle$.

$(\mathtt{f}, \mathtt{j})$. The square boxes represent segments. The box is represented next to a node for node segments and on top of the link of an adjacency segment. So, in this case, $x_1 = \mathtt{a}, x_2 = \mathtt{c}, x_3 = \mathtt{e}, x_4 = (\mathtt{f}, \mathtt{j})$ and $x_5 = \mathtt{i}$.

In this example, between nodes $\mathtt{c}$ and $\mathtt{e}$ there are two shortest paths, namely $((\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{e}))$ and $((\mathtt{c}, \mathtt{b}), (\mathtt{b}, \mathtt{e}))$. In the same way, two shortest paths exist between nodes $\mathtt{e}$ and $\mathtt{f}$ because we have two parallel links with the same IGP weight between them. In each case, any of those two paths could be used to forward packets over this sr-path. Thus, we see that the sr-path $\vec{p}$ can correspond to four paths over the original network, namely,

$$
\begin{aligned}
&((\mathtt{a}, \mathtt{c}), (\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{f}, i_1), (\mathtt{f}, \mathtt{j}), (\mathtt{j}, \mathtt{h}), (\mathtt{h}, \mathtt{i})),\\
&((\mathtt{a}, \mathtt{c}), (\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{f}, i_2), (\mathtt{f}, \mathtt{j}), (\mathtt{j}, \mathtt{h}), (\mathtt{h}, \mathtt{i})),\\
&((\mathtt{a}, \mathtt{c}), (\mathtt{c}, \mathtt{b}), (\mathtt{b}, \mathtt{e}), (\mathtt{e}, \mathtt{f}, i_1), (\mathtt{f}, \mathtt{j}), (\mathtt{j}, \mathtt{h}), (\mathtt{h}, \mathtt{i})),\\
&((\mathtt{a}, \mathtt{c}), (\mathtt{c}, \mathtt{b}), (\mathtt{b}, \mathtt{e}), (\mathtt{e}, \mathtt{f}, i_2), (\mathtt{f}, \mathtt{j}), (\mathtt{j}, \mathtt{h}), (\mathtt{h}, \mathtt{i}))
\end{aligned}
$$

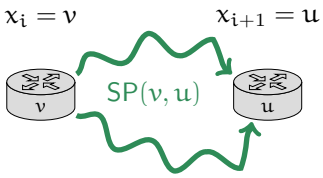
where $(\mathtt{e}, \mathtt{f}, i_1)$ and $(\mathtt{e}, \mathtt{f}, i_2)$ represent the two links between $\mathtt{e}$ and $\mathtt{f}$.

In general, when we use a sr-path $\vec{p} = \langle x_1, \ldots, x_l \rangle$ to forward traffic, between segments x_i and x_{i+1} the set of paths over which this traffic might be sent corresponds to the set of all shortest paths between those segments. In this thesis we consider two models for forwarding traffic over ECMP:

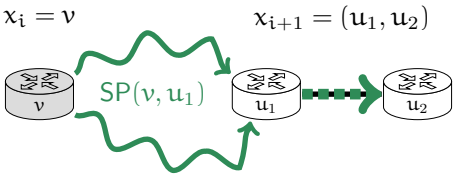
1. *Hash model*: Whenever several next-hops exists with respect to the IGP shortest paths, a hash function is used to select which of them is used. This hash function is unknown and depends on the traffic that is sent. From a practical point of view this, more or less, corresponds to assuming that one of the multiple shortest paths is selected at random.

2. *Split model*: The traffic is split evenly between all shortest paths. This means that if, for instance, a router contains two next-hops, it will forward 50% of the traffic towards one of them and the other 50% towards the other.

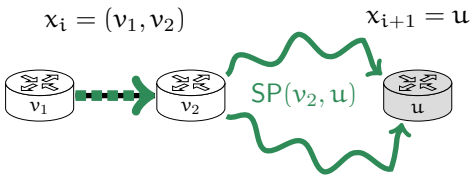
But these segments, x_i and x_{i+1} might not both correspond to nodes in which case we need to give a precise definition of what the set of shortest paths between two segments means. In general, we have four cases which are summarized in Figure 4.2. If we have two node segments $x_i = v, x_{i+1} = u$ then it is the subgraph between v and u, $\mathsf{SP}(v, u)$. If $x_i = v$ and $x_{i+1} = (u_1, u_2)$ then it the subgraph between v and u_1, $\mathsf{SP}(v, u_1)$. If $x_i = (v_1, v_2)$ and $x_{i+1} = u$ then it is the shortest path subgraph between v_2 and u. Finally, if both are adjacency segments with $x_i = (v_1, v_2)$ and $x_{i+1} = (u_1, u_2)$ then it is $\mathsf{SP}(v_2, u_1)$.



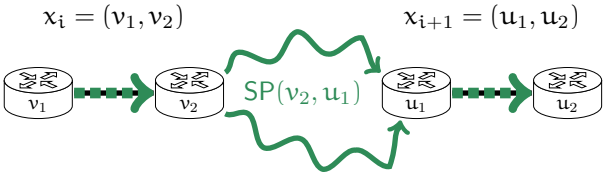
Case 1: x_i and x_{i+1} are both node segments



Case 2: x_i is a node segment and x_{i+1} is an adjacency segment



Case 3: x_i is an adjacency segment and x_{i+1} is a node segment



Case 4: x_i and x_{i+1} are both adjacency segments

Figure 4.2: Shortest paths between consecutive segments.

Having to deal with these four cases would be very cumbersome. When considering a sr-path $\vec{p}$ we want to have a way to ignore as much as possible the nature of each of the segments. This will make proving results involving sr-paths easier and also will ease the readability of proofs. This lead us to define the following notation so that we can very simply capture all four cases.

Definition 4.2. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle$. *If* $\mathsf{x}_\mathsf{i} \in \mathsf{V}(\mathsf{G})$ *we define* $\mathsf{x}_\mathsf{i}^1 = \mathsf{x}_\mathsf{i}^2 = \mathsf{x}_\mathsf{i}$ *and if* $\mathsf{x}_\mathsf{i} = (\mathsf{u}_1, \mathsf{u}_2) \in \mathsf{E}(\mathsf{G})$ *we define* $\mathsf{x}_\mathsf{i}^1 = \mathsf{u}_1$ *and* $\mathsf{x}_\mathsf{i}^2 = \mathsf{u}_2$.

With this notation, we can now simply say that the set of shortest paths between two consecutive segments x_i and $\mathsf{x}_{\mathsf{i}+1}$ of a sr-path $\vec{\mathsf{p}}$ is the subgraph $\mathsf{SP}(\mathsf{x}_\mathsf{i}^2, \mathsf{x}_{\mathsf{i}+1}^1)$. This works regardless of what the type of segments x_i and $\mathsf{x}_{\mathsf{i}+1}$ are. This notation also makes it easy to refer to the starting and ending routers of a sr-path.

Definition 4.3. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle$. *We say that* $\vec{\mathsf{p}}$ *is a sr-path from node* x_1^1 *to* x_l^2. *We say that* x_1^1 *is the first node of* $\vec{\mathsf{p}}$ *and* x_l^2 *is the last node of* $\vec{\mathsf{p}}$. *A sr-path that starts and ends at the same node is called a sr-cycle.*

As we mentioned in the introduction of this chapter, when we use segment routing as a forwarding mechanism, we need to append to the packet header the segments that are to be used. Commercial routers have strong limitations regarding the size of this header. Some routers can support sr-paths with up to 10 segments but on average this number is closer to 5 [50]. This means that it is important to be able to capture the cost, in terms of segments, of a sr-path. A node segment needs to contain the IP address of the corresponding router whereas an adjacency segment needs the IP address of the source node of the corresponding link as well as the interface identifier of that link. For this reason, we model the cost of individual segments and sr-paths as follows.

Definition 4.4. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots \mathsf{x}_\mathsf{l} \rangle$ *a sr-path on* G. *We define the cost of* x_i *as*

$$\mathit{sr\text{-}cost}(\mathsf{x}_\mathsf{i}) = \begin{cases} 1 & \text{if } \mathsf{x}_\mathsf{i} \in \mathsf{V}(\mathsf{G}) \\ 2 & \text{otherwise} \end{cases}$$

We define the segment cost *of* $\vec{\mathsf{p}}$ *as*

$$\mathit{sr\text{-}cost}(\vec{\mathsf{p}}) = \sum_{\mathsf{i}=1}^{\mathsf{l}} \mathit{sr\text{-}cost}(\mathsf{x}_\mathsf{i})$$

Note that this model does not exactly match the reality. In practice, if the first segment is a node segment, its IP address will never be put into the segment stack. Also, if the first segment is an adjacency segment only the link interface would be necessary for the same reason. We chose to ignore this and count each segment equally because it greatly simplifies the mathematical developments. Furthermore, we believe that the model is close enough that this will have a minor impact in practice. In the worst case, we overestimate the real segment cost of a sr-path by 1.

Definition 4.5. *Let* G *be a network. We denote the set of all sr-paths on* G *by* $\vec{\mathcal{P}}$. *Given* $\mathsf{k} \in \mathbb{N}$ *we define*

$$\vec{\mathcal{P}}_\mathsf{k}(\mathsf{G}) = \{\vec{\mathsf{p}} \mid \vec{\mathsf{p}} \text{ is a sr-path on } \mathsf{G} \text{ and } \mathit{sr\text{-}cost}(\vec{\mathsf{p}}) \leqslant \mathsf{k}\}.$$

Finally, given $\mathsf{s}, \mathsf{t} \in \mathsf{V}(\mathsf{G})$ *we denote the set of all sr-paths from* s *to* t *of segment cost at most* k *by* $\vec{\mathcal{P}}_\mathsf{k}(\mathsf{s}, \mathsf{t})$.

We now define some operations on sr-paths and prove some of theirs properties.

Definition 4.6. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle, \vec{\mathsf{q}} = \langle \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{r} \rangle$ *be two sr-paths on* G*. We define the sum of* $\vec{\mathsf{p}}$ *and* $\vec{\mathsf{q}}$ *as*

$$\vec{\mathsf{p}} + \vec{\mathsf{q}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l}, \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{r} \rangle$$

Lemma 4.1. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle, \vec{\mathsf{q}} = \langle \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{r} \rangle$ *be two sr-paths on* G*. Then*

$$\mathit{sr\text{-}cost}(\vec{\mathsf{p}} + \vec{\mathsf{q}}) = \mathit{sr\text{-}cost}(\vec{\mathsf{p}}) + \mathit{sr\text{-}cost}(\vec{\mathsf{q}})$$

Proof.

$$\begin{aligned} \mathsf{sr\text{-}cost}(\vec{\mathsf{p}} + \vec{\mathsf{q}}) &= \mathsf{sr\text{-}cost}(\langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l}, \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{r}) \\ &= \sum_{\mathsf{i}=1}^{\mathsf{l}} \mathsf{sr\text{-}cost}(\mathsf{x}_\mathsf{i}) + \sum_{\mathsf{i}=1}^{\mathsf{r}} \mathsf{sr\text{-}cost}(\mathsf{y}_\mathsf{i}) \\ &= \mathsf{sr\text{-}cost}(\vec{\mathsf{p}}) + \mathsf{sr\text{-}cost}(\vec{\mathsf{q}}) \end{aligned}$$

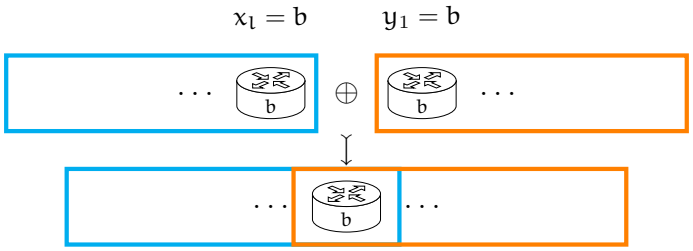
□

Addition of sr-paths will turn out to be useful to prove bounds on the segment cost of sr-paths produced by some of our algorithms. However, addition of sr-paths does not care about the segments contained in each path. It simply takes all segments from both paths and put them in order into a single sr-path. For example, it could be the case that $\mathsf{x}_\mathsf{l} = \mathsf{y}_1$ so the resulting sr-path would have a segment repeated twice next to each other. This does not cause any practical problem but in terms of routing it is redundant. Moreover, it wastes space in the segment stack. This can be the case even if $\mathsf{x}_\mathsf{l} \neq \mathsf{y}_1$ but the last node of x_l and the first node of y_1 are the same, that is, if $\mathsf{x}_\mathsf{l}^2 = \mathsf{y}_1^1$. For instance let $\vec{\mathsf{p}} = \langle \mathtt{a}, \mathtt{f} \rangle$ and $\vec{\mathsf{q}} = \langle (\mathtt{f}, \mathtt{j}), \mathtt{i} \rangle$. Then $\vec{\mathsf{p}} + \vec{\mathsf{q}} = \langle \mathtt{a}, \mathtt{f}, (\mathtt{f}, \mathtt{j}), \mathtt{i} \rangle$. In terms of routing, this sr-path traverses the same links as the sr-path $\langle \mathtt{a}, (\mathtt{f}, \mathtt{j}), \mathtt{i} \rangle$ but it costs one more because it has a useless node segment on $\mathtt{f}$. With this in mind, we define a concatenation operation $\oplus$ on sr-paths which takes this into account and discards useless segments at the concatenation point in the resulting sr-path.

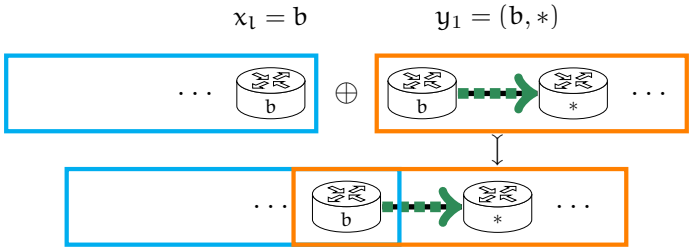
Definition 4.7. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle$ *be a sr-path from* $\mathtt{a}$ *to* $\mathtt{b}$ *and* $\vec{\mathsf{q}} = \langle \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{r} \rangle$ *be a sr-path from* $\mathtt{b}$ *to* $\mathtt{c}$*, where* $\mathtt{a}, \mathtt{b}, \mathtt{c} \in \mathsf{V}(\mathsf{G})$*. We define the concatenation of* $\vec{\mathsf{p}}$ *and* $\vec{\mathsf{q}}$ *as*

$$\vec{\mathsf{p}} \oplus \vec{\mathsf{q}} = \begin{cases} \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l}, \mathsf{y}_2, \ldots, \mathsf{y}_\mathsf{r} \rangle & \text{if } \mathsf{x}_\mathsf{l} = \mathsf{y}_1 = \mathtt{b} \in \mathsf{V}(\mathsf{G}) \\ \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l}, \mathsf{y}_2, \ldots, \mathsf{y}_\mathsf{r} \rangle & \text{if } \mathsf{x}_\mathsf{l} \in \mathsf{E}(\mathsf{G}) \text{ and } \mathsf{y}_1 \in \mathsf{V}(\mathsf{G}) \\ \langle \mathsf{x}_1, \ldots, \mathsf{x}_{\mathsf{l}-1}, \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{r} \rangle & \text{if } \mathsf{x}_\mathsf{l} \in \mathsf{V}(\mathsf{G}) \text{ and } \mathsf{y}_1 \in \mathsf{E}(\mathsf{G}) \\ \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l}, \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{r} \rangle & \text{if } \mathsf{x}_\mathsf{l} \in \mathsf{E}(\mathsf{G}) \text{ and } \mathsf{y}_1 \in \mathsf{E}(\mathsf{G}) \end{cases}$$

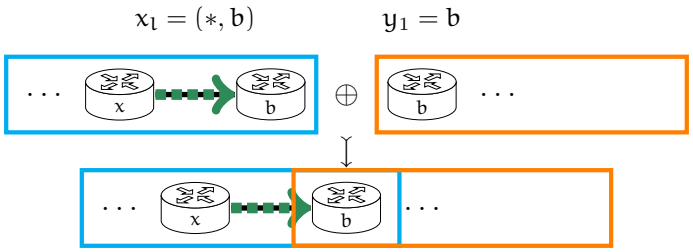
With sr-path concatenation we avoid consecutive redundant segments by removing them if necessary beforehand. For the above example with $\vec{\mathsf{p}} = \langle \mathtt{a}, \mathtt{f} \rangle$ and $\vec{\mathsf{q}} = \langle (\mathtt{f}, \mathtt{j}), \mathtt{i} \rangle$ we have $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}} = \langle \mathtt{a}, (\mathtt{f}, \mathtt{j}), \mathtt{i} \rangle$. Figure 4.3 illustrates the four cases of sr-path concatenation from Definition 4.7.



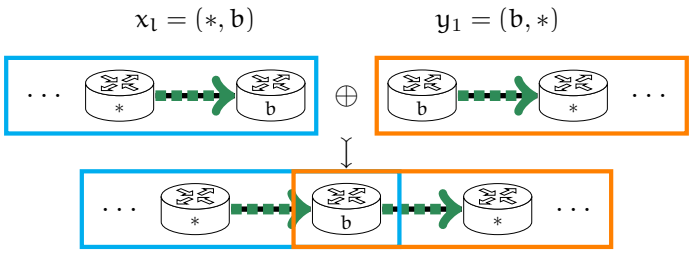
Case 1: $x_l = y_1 = b \in V$



Case 2: $x_l \in E(G)$, $y_1 \in V(G)$ and $x_l^2 = y_1 = b$



Case 3: $x_l \in V(G)$, $y_1 \in E(G)$ and $x_l = y_1^1 = b$



Case 4: $x_l \in E(G)$, $y_1 \in E(G)$ and $x_l^2 = y_1^1 = b$

Figure 4.3: The four cases of sr-path concatenation.

In contrast with sr-path addition, the cost of the resulting sr-path after a concatenation will depend on the types of the segments at the end of the first path and at the start of the second.

Lemma 4.2. *Let* G *be a network and* $\vec{p} = \langle x_1, \dots, x_l \rangle$ *be a sr-path from* a *to* b *and* $\vec{q} = \langle y_1, \dots, y_r \rangle$ *be a sr-path from* b *to* c*, where* $a, b, c \in \mathsf{V}(\mathsf{G})$*. Then*

$$\textit{sr-cost}(\vec{p} \oplus \vec{q}) = \begin{cases} \textit{sr-cost}(\vec{p}) + \textit{sr-cost}(\vec{q}) & \textit{if } x_l \in \mathsf{E}(\mathsf{G}) \textit{ and } y_1 \in \mathsf{E}(\mathsf{G}) \\ \textit{sr-cost}(\vec{p}) + \textit{sr-cost}(\vec{q}) - 1 & \textit{otherwise} \end{cases}$$

Proof. If $x_l \in \mathsf{E}(\mathsf{G})$ and $y_1 \in \mathsf{E}(\mathsf{G})$ then $\vec{p} \oplus \vec{q} = \vec{p} + \vec{q}$ so the result comes from Lemma 4.1. Otherwise, in each case $\vec{p} \oplus \vec{q}$ has the same elements as $\vec{p} + \vec{q}$ except that we removed a node segment with cost 1. Thus, by Lemma 4.1, we have that

$$\mathsf{sr\text{-}cost}(\vec{p} \oplus \vec{q}) = \mathsf{sr\text{-}cost}(\vec{p} + \vec{q}) - 1 = \mathsf{sr\text{-}cost}(\vec{p}) + \mathsf{sr\text{-}cost}(\vec{q}) - 1$$

□

These operations will be important later on because a lot of algorithms for solving problems related to segment routing can be expressed as dynamic programs where the solution sr-path is built upon smaller sr-paths. These properties tell use how the segment cost of the paths is affected by these operations.

It is often useful to refer to the set of nodes and edges that belong to some path represented by a sr-path leading to the following definition.

Definition 4.8. *Let* G *be a network and* $\vec{p} = \langle x_1, \dots, x_l \rangle$ *a sr-path on* G*. We define the set of nodes in* $\vec{p}$ *as*

$$\mathsf{V}(\vec{p}) = \bigcup_{i=1}^{l} \{x_i^1, x_i^2\} \quad \cup \quad \bigcup_{i=2}^{l} \mathsf{V}(\mathsf{SP}(x_{i-1}^2, x_i^1))$$

We define the set of edges of $\vec{p}$ *as*

$$\mathsf{E}(\vec{p}) = \{x_i \mid x_i \textit{ is an adjacency segment}\} \quad \cup \quad \bigcup_{i=2}^{l} \mathsf{E}(\mathsf{SP}(x_{i-1}^2, x_i^1))$$

Definition 4.9. *Let* G *be a network and* $\vec{p}$ *be a sr-path on* G*. We call the subgraph* $(\mathsf{V}(\mathsf{G}), \mathsf{E}(\vec{p}))$ *the* forwarding subnetwork *of* $\vec{p}$ *and denote it by* $\mathsf{forw}(\vec{p})$*. We write the set of edge in the forwarding graph as* $\mathsf{E}(\mathsf{forw}(\vec{p})) = \mathsf{E}(\vec{p})$*. We can easily see that if* $\vec{p} = \langle x_1, \dots, x_l \rangle$ *then*

$$\mathsf{E}(\vec{p}) = \bigcup_{i=2} \mathsf{E}(\mathsf{SP}(x_{i-1}^2, x_i^1)) \cup \bigcup i : x_i \in \mathsf{E}(\mathsf{G}) x_i.$$

To lighten the notation, we often omit the parenthesis and write $\mathsf{forw}\langle x_1, \dots, x_l \rangle$ rather than $\mathsf{forw}(\langle x_1, \dots, x_l \rangle)$.

The forwarding subnetwork of a sr-path $\vec{p}$ encodes every path on which packets might travel when forwarded over $\vec{p}$. However it does not encode how many times a given link is traversed by the sr-path because each edge that is traversed by $\vec{p}$ appears exactly once.

Definition 4.10. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \dots, \mathsf{x}_\mathsf{l} \rangle$ *be a sr-path on* G. *We say that* $\vec{\mathsf{p}}$ *is* deterministic *if and only if for each* $\mathsf{i} \in \{2, \dots, \mathsf{l}\}$, *there exists a unique shortest path between* $\mathsf{x}^2_{\mathsf{i}-1}$ *and* x^1_i.

Deterministic sr-paths are important when we want to have guarantees about the set of links that are traversed when one uses SR to forward traffic over a given sr-path. Because there exists a single shortest path between any two consecutive endpoints, they never use ECMP. For this reason they map back to a unique path on the network G. This kind of paths will be important in Chapter 7 where we design an algorithm that leverages segment routing to provide network monitoring for detecting single link failures. In this context, we will need to know exactly which links are covered by each sr-path used for the monitoring.

Another application of deterministic sr-paths, which we will tackle in the next section, is the reverse problem, where we start from a path on G and we want to find a sr-path whose forwarding graph matches that path.

Lemma 4.3. *Let* G *be a network and* $\vec{\mathsf{p}}$ *be a deterministic sr-path from* a *to* b *and* $\vec{\mathsf{q}}$ *be a deterministic sr-path from* b *to* c *then* $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}}$ *is a deterministic sr-path from* a *to* c.

Proof. The fact that $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}}$ is well defined is simply because we assumed that $\vec{\mathsf{p}}$ is a sr-path from a to b and $\vec{\mathsf{q}}$ a sr-path from b to c. It remains to show that it is deterministic.

Write $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \dots, \mathsf{x}_\mathsf{l} \rangle$ and $\vec{\mathsf{q}} = \langle \mathsf{y}_1, \dots, \mathsf{y}_\mathsf{r} \rangle$. Let $z_\mathsf{i}, z_{\mathsf{i}+1}$ be consecutive elements of $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}}$. We need to prove that there exists a unique shortest path between z^2_i and $z^1_{\mathsf{i}+1}$. If both $z_\mathsf{i}, z_{\mathsf{i}+1}$ belong to $\vec{\mathsf{p}}$ or $\vec{\mathsf{q}}$ then this is true since both $\vec{\mathsf{p}}$ and $\vec{\mathsf{q}}$ are deterministic. Otherwise, there are four cases that we need to consider.

Case 1: $\mathsf{x}_\mathsf{l} = \mathsf{y}_1$ are both node segments. In this case we know that $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}} = \langle \mathsf{x}_1, \dots, \mathsf{x}_\mathsf{l}, \mathsf{y}_2, \dots, \mathsf{y}_\mathsf{r} \rangle$. Thus, $z_\mathsf{i} = \mathsf{x}_\mathsf{l} = \mathsf{y}_1$ and $z_{\mathsf{i}+1} = \mathsf{y}_2$. Hence, since $\vec{\mathsf{q}}$ is deterministic, there exists a unique shortest path between $z^2_\mathsf{i} = \mathsf{y}^2_1$ and $z^1_{\mathsf{i}+1} = \mathsf{y}^1_2$.

Case 2: $\mathsf{x}_\mathsf{l}, \mathsf{y}_1$ are both adjacency segments and $\mathsf{x}^2_\mathsf{l} = \mathsf{y}^1_1$. In this case we know that $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}} = \langle \mathsf{x}_1, \dots, \mathsf{x}_\mathsf{l}, \mathsf{y}_1, \mathsf{y}_2, \dots, \mathsf{y}_\mathsf{r} \rangle$. Hence, $z_\mathsf{i} = \mathsf{x}_\mathsf{l}$ and $z_{\mathsf{i}+1} = \mathsf{y}_1$ so $z^2_\mathsf{i} = \mathsf{x}^2_\mathsf{l} = \mathsf{y}^1_1 = z^1_{\mathsf{i}+1}$ so the unique shortest path between z^2_i and $z^1_{\mathsf{i}+1}$ is the empty path.

Case 3: x_l is an adjacency segment and y_1 is a node segment such that $\mathsf{x}^2_\mathsf{l} = \mathsf{y}_1$. By definition, $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}} = \langle \mathsf{x}_1, \dots, \mathsf{x}_\mathsf{l}, \mathsf{y}_2, \dots, \mathsf{y}_\mathsf{r} \rangle$. Thus, $z_\mathsf{i} = \mathsf{x}_\mathsf{l}$ and $z_{\mathsf{i}+1} = \mathsf{y}_2$. Thus, since $\vec{\mathsf{q}}$ is deterministic, there exists a unique shortest path between $z^2_\mathsf{i} = \mathsf{x}^2_\mathsf{l} = \mathsf{y}^2_1$ and $z^1_{\mathsf{i}+1} = \mathsf{y}^1_2$.

Case 4: x_l is a node segment and y_1 is an adjacent segment such that $\mathsf{y}^1_1 = \mathsf{x}_\mathsf{l}$. Therefore $\vec{\mathsf{p}} \oplus \vec{\mathsf{q}} = \langle \mathsf{x}_1, \dots, \mathsf{x}_{\mathsf{l}-1}, \mathsf{y}_1, \mathsf{y}_2, \dots, \mathsf{y}_\mathsf{r} \rangle$. Thus, $z_\mathsf{i} = \mathsf{x}_{\mathsf{l}-1}$ and $z_{\mathsf{i}+1} = \mathsf{y}_1$. Thus, since $\vec{\mathsf{p}}$ is deterministic, there exists a unique shortest path between $z^2_\mathsf{i} = \mathsf{x}^2_{\mathsf{l}-1}$ and $z^1_{\mathsf{i}+1} = \mathsf{y}^1_2 = \mathsf{x}^1_\mathsf{l}$. □

4.2 Acyclic sr-paths

Sr-paths can contain cycles as shown in Figure 4.4. In some situations we can take advantage of these cycles to obtain better solution. We discuss one such

example when we talk about traffic engineering in Chapter 6. In other situations we can show that cycles can never lead to an optimal solution. We are now going to prove that we can always transform a sr-path that contains cycles into a sr-path that is acyclic, visits a subset of the edges previously visited and does not have a higher segment cost. In the case of Figure 4.4 we could achieve this by replacing node segment $\mathtt{d}$ by $\mathtt{e}$. The set of edges in $\mathsf{forw}\langle \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{f} \rangle$ is a subset of the set of edges in $\mathsf{forw}\langle \mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{f} \rangle$ but $\langle \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{f} \rangle$ is acyclic.

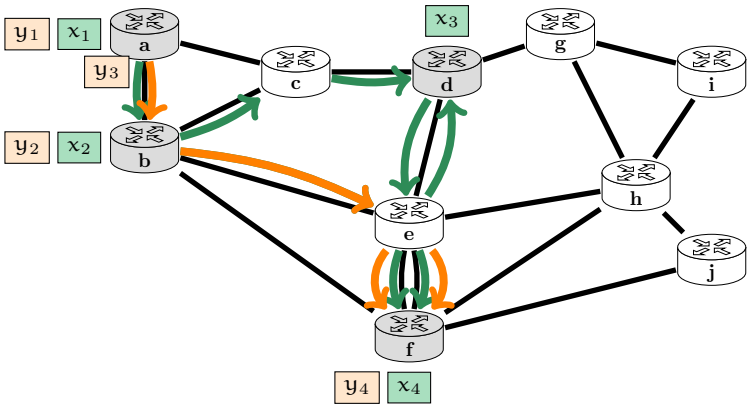


Figure 4.4: Illustration of a cyclic sr-path $\vec{\mathsf{p}} = \langle \mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{f} \rangle$. It contains cycle $((\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{d}))$. The sr-path $\vec{\mathsf{q}} = \langle \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{f} \rangle$ also goes from $\mathtt{a}$ to $\mathtt{f}$ but is acyclic. It visits a subset of the edges of $\mathsf{forw}(\vec{\mathsf{p}})$ so it is a sr-subpath of $\vec{\mathsf{q}}$.

Next we give a formal definition of cyclic sr-paths.

Definition 4.11. *Let* G *be a network and* $\vec{\mathsf{p}}$ *a sr-path on* G*. We say that* $\vec{\mathsf{p}}$ *is* cyclic *if* $\mathsf{forw}(\vec{\mathsf{p}})$ *is cyclic.*

Do not confuse this definition with sr-cycles. A sr-cycle is simply a sr-path whose first and last nodes are the same. Of course any sr-cycle is cyclic. But not all cyclic sr-paths are sr-cycles. We start by proving the theorem in the case where the sr-path contains only node segments.

The following simple lemma shows that if two nodes are connected then we can always find an acyclic sr-path between them.

Lemma 4.4. *Let* G *be a network and* $\mathsf{u}, \mathsf{v} \in \mathsf{G}$*. If there exists a path from* u *to* v *on* G *then there exists an acyclic sr-path from* u *to* v*.*

Proof. We can simply take a minimal segmentation $\vec{\mathsf{p}}$ of p. In this case $\mathsf{forw}(\vec{\mathsf{p}}) = \mathsf{p}$ so it is acyclic. □

Definition 4.12. *Let* G *be a network and* $\vec{\mathsf{p}}$ *a sr-path from* u *to* v *on* G*. We say that* $\vec{\mathsf{q}}$ *is a* sr-subpath *of* $\vec{\mathsf{p}}$ *if* $\vec{\mathsf{q}}$ *is a sr-path from* u *to* v *and* $\mathsf{forw}(\vec{\mathsf{q}})$ *is a subnetwork of* $\mathsf{forw}(\vec{\mathsf{p}})$*.*

The next theorem shows that for any sr-path, we can always find a sr-subpath of it that is acyclic and whose segment cost is at most the segment cost of the original path. This theorem has important practical implications as we will see in Chapter 8.

Theorem 4.5. *Let* G *be a network with* $\mathsf{igp}(e) > 0$ *for all* $e \in \mathsf{E}(\mathsf{G})$, $u, v \in \mathsf{V}(\mathsf{G})$ *with* $u \neq v$ *and* $\vec{p}$ *be a sr-path on* G *from* u *to* v *using only node segments. Then there exists an acyclic sr-path* $\vec{q}$ *from* u *to* v *such that* $\mathsf{forw}(\vec{q}) \subseteq \mathsf{forw}(\vec{p})$ *and* $\mathit{sr\text{-}cost}(\vec{q}) \leqslant \mathit{sr\text{-}cost}(\vec{p})$.

Proof. Let $\vec{p} = \langle x_1, \ldots, x_n \rangle$. The proof is by induction on n. If $n = 1$ then $\mathsf{forw}(\vec{p})$ has 0 edges and is therefore acyclic. If $n = 2$ then $\mathsf{forw}(\vec{p})$ is equal to the union of shortest paths from x_1 to x_2 which form an acyclic graph. Assume that $\langle x_1, \ldots, x_{n-1} \rangle$ is acyclic but $\vec{p}$ is not. We will show how to construct $\vec{q}$ satisfying the theorem conditions.

Since $\vec{p}$ is cyclic, we can let i be the smallest index such that there exists a node v in $\mathsf{forw}\langle x_i, x_{i+1} \rangle$ that belongs to a cycle in $\mathsf{forw}(\vec{p})$. If several such nodes exist, assume that v is the one closest to x_i (in terms of IGP, if several exist, choose any).

Let $\vec{q} = \langle x_1, \ldots, x_i, v, x_n \rangle$. Since v belongs to both $\mathsf{forw}\langle x_i, x_{i+1} \rangle$ and $\mathsf{forw}\langle x_{n-1}, x_n \rangle$ we have

$$\mathsf{forw}\langle x_1, \ldots, x_i, v \rangle \subseteq \mathsf{forw}\langle x_1, \ldots, x_i, x_{i+1} \rangle$$

and

$$\mathsf{forw}\langle v, x_n \rangle \subseteq \mathsf{forw}\langle x_{n-1}, x_n \rangle$$

so that

$$\begin{aligned}
\mathsf{forw}(\vec{q}) &= \mathsf{forw}\langle x_1, \ldots, x_i, v \rangle \cup \mathsf{forw}\langle v, x_n \rangle \\
&\subseteq \mathsf{forw}\langle x_1, \ldots, x_i, x_{i+1} \rangle \cup \mathsf{forw}\langle x_{n-1}, x_n \rangle \\
&\subseteq \mathsf{forw}\langle x_1, \ldots, x_i, x_{i+1} \rangle \cup \mathsf{forw}\langle x_{i+2}, \ldots x_{n-2} \rangle \cup \mathsf{forw}\langle x_{n-1}, x_n \rangle \\
&= \mathsf{forw}(\vec{p}).
\end{aligned}$$

Similarly, we have

$$\begin{aligned}
\mathsf{sr\text{-}cost}(\vec{q}) &= \mathsf{sr\text{-}cost}\langle x_1, \ldots, x_i, v \rangle + \mathsf{sr\text{-}cost}\langle v, x_n \rangle - 1 \\
&= \mathsf{sr\text{-}cost}\langle x_1, \ldots, x_i, x_{i+1} \rangle + \mathsf{sr\text{-}cost}\langle x_{n-1}, x_n \rangle - 1 \\
&< \mathsf{sr\text{-}cost}\langle x_1, \ldots, x_i, x_{i+1} \rangle + \mathsf{sr\text{-}cost}\langle x_{i+2}, \ldots x_{n-2} \rangle + \mathsf{sr\text{-}cost}\langle x_{n-1}, x_n \rangle \\
&= \mathsf{sr\text{-}cost}(\vec{p}).
\end{aligned}$$

The sr-path $\vec{q}$ starts at $x_1 = u$ and ends at $x_n = v$. If $\vec{q}$ is acyclic then the proof is complete. Otherwise, let c by a cycle on $\mathsf{forw}(\vec{q})$. Since both $\mathsf{forw}\langle x_1, \ldots, x_i, v \rangle$ and $\mathsf{forw}\langle v, x_n \rangle$ are acyclic, c cannot be fully contained in either one of them. This means that there must be a node $u \in \mathsf{V}(c) \setminus v$ in $\mathsf{forw}\langle x_1, \ldots, x_i, v \rangle$. By choice of v, the distance from x_i to u and v must be the same and u must be in the shortest path from x_i to v. This is impossible since we assume that $\mathsf{igp}(e) > 0$ for all $e \in \mathsf{E}(\mathsf{G})$. □

The theorem is also true in the general case where we allow the sr-path $\vec{p}$ to contain adjacency segments. However the proof is quite tedious as it involves a lot of cases. Rather than provide a detailed proof, we instead provide the case analysis and a valid acyclic sr-subpath in each case.

The construction works very similarly as in the proof of Theorem 4.5. Let's focus on the general case only. Assume that $\vec{p} = \langle x_1, \ldots, x_n \rangle$ is cyclic but $\langle x_1, \ldots, x_{n-1} \rangle$ is acyclic. As in the above proof, let i be the smallest index such

	$v = x_i^1$	$v = x_i^2$	$v \in \langle x_i^2, x_{i+1}^1 \rangle$
$v = x_{n-1}^1$	$\langle x_1, \ldots, x_{i-1}, x_{n-1}, x_n \rangle$	$\langle x_1, \ldots, x_i, x_{n-1}, x_n \rangle$	$\langle x_1, \ldots, x_i, x_{n-1}, x_n \rangle$
$v = x_{n-1}^2$	$\langle x_1, \ldots, x_{i-1}, v, x_n \rangle$	$\langle x_1, \ldots, x_i, x_n \rangle$	$\langle x_1, \ldots, x_i, v, x_n \rangle$
$v = x_n^1$	$\langle x_1, \ldots, x_{i-1}, x_n \rangle$	$\langle x_1, \ldots, x_i, x_n \rangle$	$\langle x_1, \ldots, x_i, v, x_n \rangle$
$v = x_n^2$	$\langle x_1, \ldots, x_{i-1}, v \rangle$	$\langle x_1, \ldots, x_i \rangle$	$\langle x_1, \ldots, x_i, v \rangle$
$v \in \langle x_{n-1}, x_n \rangle$	$\langle x_1, \ldots, x_{i-1}, v, x_n \rangle$	$\langle x_1, \ldots, x_i, x_n \rangle$	$\langle x_1, \ldots, x_i, v, x_n \rangle$

Table 4.1: Construction of an acyclic sr-subpath.

that there exists a node v in $\mathsf{forw}\langle x_i, x_{i+1} \rangle$ that belongs to a cycle in $\mathsf{forw}(\vec{p})$. If several such nodes exist, assume that v is the one closest to x_i (in terms of IGP, if several exist, choose any).

With this choice of x_i and v we can build $\vec{q}$ by following Table 4.1. The construction of $\vec{q}$ depends on two things:

1. Where v is crossed when we move on $\vec{p}$ from x_i to x_{i+1}.
2. Where v is crossed when we move on $\vec{p}$ from x_{n-1} to x_n.

For the first, we consider three cases: (1) v is the node x_i^1, (2) v is the node x_i^2 or (3) v is neither of those nodes and belongs to a shortest path from x_i^2 to x_{i+1}^1 (we represent this by $v \in \langle x_i^2, x_{i+1}^1 \rangle$ on the table). This is represented the columns of the table.

For the second, we consider five cases: (1) v is the node x_{n-1}^1, (2) v is the node x_{n-1}^2, (3) v is the node x_n^1, (4) v is the node x_n^2 and finally (5) v is neither of those nodes and belongs the shortest path between x_{n-1}^2 to x_n^1 (we represent this by $v \in \langle x_{n-1}^2, x_n^1 \rangle$ on the table). These cases are represented by the rows of the table.

For example, if $v = x_i^2$ and $v \in \langle x_{n-1}, x, \rangle$ then $\vec{q} = \langle x_1, \ldots, x_i, x_n \rangle$ is a solution.

Note that for construction to be correct we do not need to assume anything about the kind of segments x_i, x_{i+1}, x_{n-1} and x_n are. It may seem that we are assuming that they are adjacency segments but if they are not the construction still works. The only thing that might happen is that we may have some repeated redundant segments in $\vec{q}$. However even with these, we can show that $\mathsf{sr\text{-}cost}(\vec{q}) \leqslant \mathsf{sr\text{-}cost}(\vec{p})$.

Putting these ideas together we can prove the following[2] general theorem.

Theorem 4.6. *Let* G *be a network with* $\mathsf{igp}(e) > 0$ *for all* $e \in E(G)$, $u, v \in V(G)$ *with* $u \neq v$ *and* $\vec{p}$ *be a sr-path on* G *from* u *to* v. *Then there exists an acyclic sr-path* $\vec{q}$ *from* u *to* v *such that* $\mathsf{forw}(\vec{q}) \subseteq \mathsf{forw}(\vec{p})$ *and* $\mathit{sr\text{-}cost}(\vec{q}) \leqslant \mathit{sr\text{-}cost}(\vec{p})$.

4.3 Minimal segmentations

In this section we study the problem of efficiently representing paths in the network with SR. Suppose that you have a path in your network that you want to use to send traffic between two nodes. Traditional shortest path routing does

not allow you to forward traffic on it unless that path already is a shortest paht with respect to the igp weights configured on the links. SR makes it possible, at least in theory, to forward traffic on *any* path in the network since we can always just add all edges of the path as adjacency segments in the segment routing stack. In practice however, this solution cannot be implemented because, as we mentioned above, due to hardware limitations of the routers currently deployed in the networks, the stack size is often limited to a few segments [50].

This motivates the problem of, given a path p in the network, finding a sr-path $\vec{\mathsf{p}}$ that represents p. That is, a sr-path $\vec{\mathsf{p}}$ such that if traffic is routed over it, packets will traverse exactly the edges of p in order. To do that, we need a formal definition of what does it mean for a sr-path to represent a path in the network.

Definition 4.13. *Let $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle$ be a deterministic sr-path. We define*

$$\mathsf{path}(\vec{\mathsf{p}}) = \mathsf{x}_1 \oplus \mathsf{SP}(\mathsf{x}_1^2, \mathsf{x}_2^1) \oplus \mathsf{x}_2 \oplus \ldots \oplus \mathsf{SP}(\mathsf{x}_{\mathsf{l}-1}^2, \mathsf{x}_\mathsf{l}^1) \oplus \mathsf{x}_\mathsf{l}$$

Note that since $\vec{\mathsf{p}}$ is deterministic, $\mathsf{path}(\vec{\mathsf{p}})$ corresponds to a path in G since each $\mathsf{SP}(\mathsf{x}_{\mathsf{i}-1}^2, \mathsf{x}_\mathsf{i}^1)$ is a single path.

Definition 4.14. *Let G be a network and p a path on G. A deterministic sr-path $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{l} \rangle$ is said to be a segmentation of p if and only if $\mathsf{p} = \mathsf{path}(\vec{\mathsf{p}})$.*

Note that we restrict ourselves to deterministic sr-paths since the equality in the above definition would never hold if for some i, $\mathsf{SP}(\mathsf{x}_\mathsf{i}^2, \mathsf{x}_{\mathsf{i}+1}^1)$ was not a path. Moreover, $\oplus$ is only defined if both operands are paths. We start by showing that any path possess a segmentation.

Lemma 4.7. *Let G be a network and p a path on G. Then, there exists a sr-path $\vec{\mathsf{p}}$ such that $\vec{\mathsf{p}}$ is a segmentation of p. Moreover, any path admits a segmentation of cost at most $2|\mathsf{E}(\mathsf{G})|$.*

Proof. Let $\mathsf{p} = (\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{l})$ be a path on G and let $\vec{\mathsf{p}} = \langle \mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{l} \rangle$. For each $\mathsf{i} = 1, \ldots, \mathsf{l} - 1$ we have that $\mathsf{SP}(\mathsf{e}_\mathsf{i}^2, \mathsf{e}_{\mathsf{i}+1}^1) = \emptyset$ since $\mathsf{e}_\mathsf{i}^2 = \mathsf{e}_{\mathsf{i}+1}^1$. Thus

$$\mathsf{e}_1 \oplus \mathsf{SP}(\mathsf{e}_1^2, \mathsf{e}_2^1) \oplus \mathsf{e}_2 \oplus \ldots \oplus \mathsf{SP}(\mathsf{e}_{\mathsf{l}-1}^2, \mathsf{e}_\mathsf{l}^1) \oplus \mathsf{e}_\mathsf{l} = \mathsf{e}_1 \oplus \mathsf{e}_2 \oplus \ldots \oplus \mathsf{e}_\mathsf{l} = \mathsf{p}.$$

□

Note that previous models of segment routing considered only node segments and therefore this result was not true. In such models it is impossible to represent some network paths in terms of SR. As we saw in Chapter 3, there are about 1% of the network links that do not belong to a shortest path and 24% of pairs of nodes with ECMP. We mentioned that those were the conditions that might lead to the need of using an adjacency segment. We characterize in Proposition 4.9 exactly which situations lead to the need of using an adjacency segment.

We call the segmentation used in the proof of Lemma 4.7 the *edge segmentation* of p. Consider the path $((\mathtt{a},\mathtt{b}),(\mathtt{b},\mathtt{c}),(\mathtt{c},\mathtt{d}),(\mathtt{d},\mathtt{g}),(\mathtt{g},\mathtt{i}))$ shown in green in Figure 4.5. We already saw in Lemma 4.7 above that $\vec{\mathsf{p}} = \langle (\mathtt{a},\mathtt{b}),(\mathtt{b},\mathtt{c}),(\mathtt{c},\mathtt{d}),(\mathtt{d},\mathtt{g}),(\mathtt{g},\mathtt{i}) \rangle$

is a segmentation of $\mathtt{p}$. But this is not the unique segmentation of $\mathtt{p}$. For example $\vec{\mathtt{q}} = \langle \mathtt{a}, (\mathtt{b}, \mathtt{c}), \mathtt{g}, \mathtt{i} \rangle$ is also a segmentation of $\mathtt{p}$. We have

$$\begin{aligned} e_1 \oplus \mathsf{SP}(e_1^2, e_2^1) \oplus e_2 \oplus \ldots \oplus \mathsf{SP}(e_{l-1}^2, e_l^1) \oplus e_l = \\ (\mathtt{a}) \oplus \mathsf{SP}(\mathtt{a}, \mathtt{b}) \oplus (\mathtt{b}, \mathtt{c}) \oplus \mathsf{SP}(\mathtt{c}, \mathtt{g}) \oplus (\mathtt{g}) \oplus \mathsf{SP}(\mathtt{g}, \mathtt{i}) \oplus (\mathtt{i}) = \\ (\mathtt{a}) \oplus ((\mathtt{a}, \mathtt{b})) \oplus ((\mathtt{b}, \mathtt{c})) \oplus ((\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{g})) \oplus (\mathtt{g}) \oplus ((\mathtt{g}, \mathtt{i})) \oplus (\mathtt{i}) = \\ ((\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{g}), (\mathtt{g}, \mathtt{i})). \end{aligned}$$

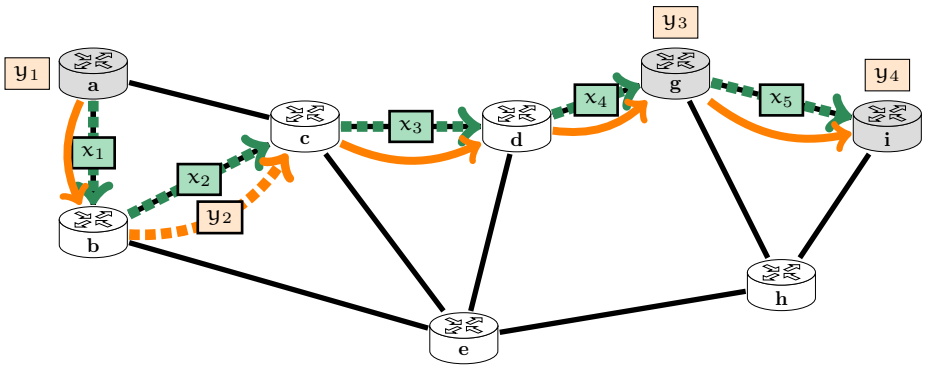


Figure 4.5: Illustration of two different segmentations of the same path $\mathtt{p} = ((\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{g}), (\mathtt{g}, \mathtt{i}))$. In green we represent the edge segmentation $\langle (\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{g}), (\mathtt{g}, \mathtt{i}) \rangle$ and in orange the segmentation $\langle \mathtt{a}, (\mathtt{b}, \mathtt{c}), \mathtt{g}, \mathtt{i} \rangle$.

Lemma 4.8. *Let* G *be a network. Any path in* G *has a segmentation with segment cost at most* $2 \cdot |\mathsf{E}(\mathsf{G})|$.

Proof. Given a path $\mathtt{p} = (e_1, \ldots, e_l)$ in G, we have that $\vec{\mathtt{p}} = \langle e_1, \ldots, e_2 \rangle$ is a segmentation of $\mathtt{p}$ with cost $2 \cdot |\mathsf{E}(\mathsf{G})|$. □

This shows that not all segmentations are equal. Even though they represent the same path in the network, they do not have the same segment cost. In practice we would like the one with the minimum cost. This motivates the following problem.

Problem 1 (Minimum path segmentation)

Given a network G *and a path* $\mathtt{p}$ *in* G*, find a segment* $\vec{\mathtt{p}}$ *of* $\mathtt{p}$ *such that* $\mathsf{sr\text{-}cost}(\vec{\mathtt{p}})$ *is minimal.*

We propose a greedy polynomial time algorithm solving Problem 1. The idea behind the algorithm is that we start walking over the input path and incrementally build the segmentation so that at each step the current sr-path is a segmentation of the prefix of the path that we traversed so far. At each node, we check whether replacing the last node of the current segmentation by that node yields a correct segmentation. If it does, then we replace it move on to the next node. If it does not, then we need to either append that node to

the segmentation or the adjacency segment corresponding to the last traversed edge.

To illustrate this, consider the graph on Figure 4.5 and suppose that the input path is $\mathtt{p} = ((\mathtt{a},\mathtt{b}),(\mathtt{b},\mathtt{c}),(\mathtt{c},\mathtt{d}),(\mathtt{d},\mathtt{g}),(\mathtt{g},\mathtt{i}))$, shown in blue. The idea is to start with a sr-path $\langle \mathtt{a} \rangle$ and follow path $\mathtt{p}$ adding segments whenever necessary. When we arrive at node $\mathtt{b}$ we have that $\mathsf{SP}(\mathtt{a},\mathtt{b}) = (\mathtt{a},\mathtt{b})$ so appending $\mathtt{b}$ to the segmentation make the paths match. Then we go to node $\mathtt{c}$ and try to see whether $\mathtt{c}$ can replace the last segment, $\mathtt{b}$. This would give the sr-path $\langle \mathtt{a},\mathtt{c} \rangle$ which is not ok since $\mathsf{SP}(\mathtt{a},\mathtt{c}) = (\mathtt{a},\mathtt{c})$ which is not a prefix of path $\mathtt{p}$. Thus, we cannot replace $\mathtt{b}$ so we try to add $\mathtt{c}$ getting the path $\langle \mathtt{a},\mathtt{b},\mathtt{c} \rangle$. Now we are good because the current sr-path represents $(\mathtt{a},\mathtt{b},\mathtt{c})$ which is a prefix of $\mathtt{p}$. Then, we do the same with $\mathtt{d}$. Replacing $\mathtt{c}$ by $\mathtt{d}$ gives the sr-path $\langle \mathtt{a},\mathtt{b},\mathtt{d} \rangle$. This path is not ok because there are two shortest paths between nodes $\mathtt{b}$ and $\mathtt{d}$, namely, $(\mathtt{b},\mathtt{c},\mathtt{d})$ and $(\mathtt{b},\mathtt{e},\mathtt{d})$ so $\vec{\mathtt{p}}$ would not be deterministic. This means that, as before, we need to add $\mathtt{d}$ to the end of the list getting the sr-path $\langle \mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d} \rangle$. Next we go to $\mathtt{g}$ and this time replacing node $\mathtt{d}$ works because the unique shortest path between $\mathtt{c}$ and $\mathtt{g}$ is $((\mathtt{c},\mathtt{d}),(\mathtt{d},\mathtt{g}))$ which matches $\mathtt{p}$. So far $\vec{\mathtt{p}} = \langle \mathtt{a},\mathtt{b},\mathtt{c},\mathtt{g} \rangle$ which represents the prefix $((\mathtt{a},\mathtt{b}),(\mathtt{b},\mathtt{c}),(\mathtt{c},\mathtt{d}),(\mathtt{d},\mathtt{g}))$ of $\mathtt{p}$. Finally, we process node $\mathtt{i}$. Replacing $\mathtt{g}$ by $\mathtt{i}$ gives the sr-path $\langle \mathtt{a},\mathtt{b},\mathtt{c},\mathtt{i} \rangle$ which does not match path $\mathtt{p}$ because there are two shortest paths between nodes $\mathtt{c}$ and $\mathtt{i}$. Thus we need to append node $\mathtt{i}$ instead and the final sr-path is $\vec{\mathtt{p}} = \langle \mathtt{a},\mathtt{b},\mathtt{c},\mathtt{g},\mathtt{i} \rangle$.

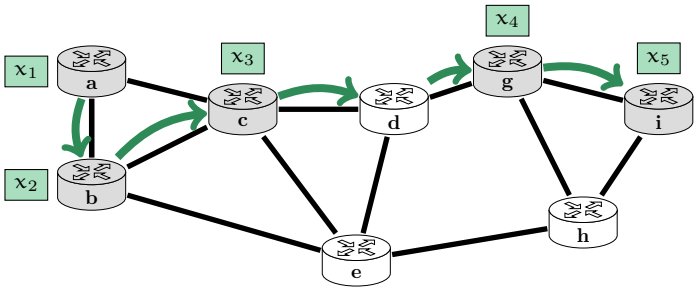


Figure 4.6: Minimal path segmentation example.

Observation. *In a previous example we saw that* $\langle \mathtt{a},(\mathtt{b},\mathtt{c}),\mathtt{g},\mathtt{i} \rangle$ *is a segmentation of* $\mathtt{p}$. *Our algorithm produced instead the segmentation* $\langle \mathtt{a},\mathtt{b},\mathtt{c},\mathtt{g},\mathtt{i} \rangle$. *Both these segmentations are minimal segmentations of* $(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{g},\mathtt{i})$ *on the network shown in Figure 4.6. This shows that* minimal segmentations are not unique.

An alternative way to think about our algorithm is to imagine that we start following path $\mathtt{p}$ on the shortest path subnetwork rooted at its origin. We do so until we either reach an edge $\mathtt{e}$ that forces us to move out of the current shortest path subnetwork or a node with in-degree in the subnetwork larger than 1 (ECMP). In both cases we need to exit the current shortest path subnetwork either by adding a node segment on e^1 and continuing the walk on $\mathsf{SP}(e^1)$ or by adding an adjacency segment over $\mathtt{e}$ and continuing the walk over $\mathsf{SP}(e^2)$.

In the remainder of this section, we are going to provide a formal description of the minimum path segmentation algorithm, give its time complexity and prove its correctness.

We start by analyzing all the situations where we need adjacency segments.

Algorithm 1 min-seg $(G, p = (e_1, \dots e_l))$

```
1: r ← p.firstNode()
2: p⃗ ← ⟨⟩
3: for e = (u, v) ∈ (e_1, ... e_l) do
4:    if e ∉ E(SP(G, r)) or δ⁻(SP(G, r), v) > 1 then
5:       if e ∉ E(SP(G, u)) or δ⁻(SP(G, u), v) > 1 then
6:          p⃗.append(e)
7:          r ← v
8:       else
9:          p⃗.append(u)
10:         r ← u
11: if |p⃗| = 0 or p⃗.origin() ≠ p.origin() then
12:    p⃗.addFirst(p.origin())
13: if p⃗.destination() ≠ p.destination() then
14:    p⃗.addFirst(p.destination())
15: return p⃗
```

Proposition 4.9. *Let* G *be a network and* p *a path on* G. *Suppose that* $e = (u, v) \in E(p)$ *is such that either* e *is not a shortest path between* u *and* v *or there exists a shortest path from* u *to* v *that does not contain* e. *Then any segmentation of* p *contains the adjacency segment* e. *We denote this set of adjacency segments as* $adj(G, p)$.

Proof. Let $\vec{p} = \langle x_1, \dots, x_l \rangle$ be a segmentation of p that does not contain $e = (u, v)$ in its segmentation. Then it means that there exists i such that e is on the shortest path between x_i^2 and x_{i+1}^1. By hypothesis, there exists a shortest path that does not contain e from u to v. Replacing e by that path yields another shortest path between x_i^2 and x_{i+1}^1 showing that $\vec{p}$ is not deterministic and thus not a segmentation. □

The next two lemmas give conditions upon which we can *remove* a node segment and an adjacency from a segmentation of a path p such that the result it still a segmentation of p.

Lemma 4.10. *Let* G *be a network and* $p = (e_1, \dots, e_l)$ *a path on* G. *Let* $\vec{p} = \langle x_1, \dots x_l \rangle$ *be a segmentation of* p *such that* $x_i \in V(G)$ *for some* i. *Then* $\langle x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_l \rangle$ *is a segmentation of* p *if and only if there is a unique shortest path between* x_{i-1}^2 *and* x_{i+1}^1 *that passes by node* x_i.

Proof. ($\Rightarrow$) Assume that $\vec{q} = \langle x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_l \rangle$ is also a segmentation of p. Then $\vec{q}$ is deterministic so there is a unique shortest path between x_{i-1}^2 and x_{i+1}^1. To see that it passes by node x_i we observe that by definition $path(\vec{p}) = path(\vec{q})$ so

$$SP(x_{i-1}^2, x_i^1) \oplus x_i \oplus SP(x_i^2, x_{i+1}^1) = SP(x_{i-1}^2, x_{i+1}^1).$$

($\Leftarrow$) Since there is a unique shortest path from x_{i-1}^2 and x_{i+1}^1 and this path passes by node $x_i^1 = x_i^2$ we have that

$$\begin{aligned}
SP(x_{i-1}^2, x_{i+1}^1) &= SP(x_{i-1}^2, x_i^1) \oplus SP(x_i^1, x_{i+1}^1) && \text{[unique sp passes by } x_i^1\text{]} \\
&= SP(x_{i-1}^2, x_i^1) \oplus SP(x_i^2, x_{i+1}^1) && [x_i^2 = x_i^1] \\
&= SP(x_{i-1}^2, x_i^1) \oplus x_i \oplus SP(x_i^2, x_{i+1}^1) && \text{[def of } \oplus \text{ and } x_i^1 = x_i = x_i^2\text{]}
\end{aligned}$$

Therefore,

$$\begin{aligned} p &= \mathsf{path}(\vec{p}) \\ &= x_1 \oplus \ldots \oplus x_{i-1} \oplus \mathsf{SP}(x_{i-1}^2, x_i^1) \oplus x_i \oplus \mathsf{SP}(x_i^2, x_{i+1}^1) \oplus x_{i+1} \oplus \ldots x_l \\ &= x_1 \oplus \ldots \oplus x_{i-1} \oplus \mathsf{SP}(x_{i-1}^2, x_{i+1}^1) \oplus x_{i+1} \oplus \ldots x_l \\ &= \mathsf{path}\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_l \rangle \end{aligned}$$

meaning that $\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_l \rangle$ is a segmentation of p. □

We have the following analogous result for adjacency segments.

Lemma 4.11. *Let* G *be a network and* $p = (e_1, \ldots, e_l)$ *a path on* G. *Let* $\vec{p} = \langle x_1, \ldots x_l \rangle$ *be a segmentation of* p *such that* $x_i \in E(G)$ *for some* i. *Then* $\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_l \rangle$ *is a segmentation of* p *if and only if there is a unique shortest path between* x_{i-1}^2 *and* x_{i+1}^1 *that contains edge* x_i.

Proof. The proof is analogous to the proof of Lemma 4.10. □

The next two results characterize the first and last segments of any segment of a path p. They are quite intuitive and say that a segmentation of a path $p = (e_1, \ldots, e_k)$ must either start with a node segment e_1^1 or an adjacency segment over e_1 and end with either a node segment e_k^2 or an adjacency segment over e_k.

Lemma 4.12. *Given a graph* G *and a path* $p = (e_1, \ldots, e_k)$ *on* G. *Let* $\vec{p} = \langle x_1, \ldots, x_l \rangle$ *be a segmentation of* p. *Then either* $x_1 = e_1^1$ *or* $x_1 = e_1$

Proof. Since $\vec{p}$ is a segmentation of p we have that $\mathsf{path}(\vec{p}) = p$. Therefore, by definition of $\mathsf{path}(\vec{p})$, either $x_1 = e_1$ or x_1 is a node segment and e_1 is the first edge of $\mathsf{SP}(x_1^2, x_2^1)$. In the latter case we must have $x_1^1 = e_1^1$. □

Lemma 4.13. *Given a graph* G *and a path* $p = (e_1, \ldots, e_k)$ *on* G. *Let* $\vec{p} = \langle x_1, \ldots, x_l \rangle$ *be a segmentation of* p. *Then either* $x_l = e_k^2$ *or* $x_l = e_k$

Proof. Since $\vec{p}$ is a segmentation of p we have that $\mathsf{path}(\vec{p}) = p$. Therefore, by definition of $\mathsf{path}(\vec{p})$, either $x_l = e_k$ or x_l is a node segment and e_k is the last edge of $\mathsf{SP}(x_{l-1}^2, x_l^2)$. In the latter case we must have $x_l^2 = e_k^2$. □

To prove that our minimum segmentation algorithm is correct, we will show that any minimal segmentation can be transformed in a series of steps to match the output of our algorithm. The next lemma will be important in one of those transformation steps. Contrary to Lemma 4.10, it tells us a condition under wich we can *add* a segment to an existing segmentation of a path p while preserving the fact that it is a segmentation of p.

Lemma 4.14. *Let* G *be a network and* p *a path on* G. *Let* $\vec{p} = \langle x_1, \ldots, x_l \rangle$ *be a segmentation of* p. *For any* $i = 1, \ldots, l-1$, *if* $v \in V(\mathsf{SP}(x_i^2, x_{i+1}^1))$ *then* $\langle x_1, \ldots, x_i, v, x_{i+1}, \ldots, x_l \rangle$ *is a segmentation of* p.

Proof. Since $\vec{p}$ is deterministic and $v \in V(\mathsf{SP}(x_i^2, x_{i+1}^1))$ it holds that

$$\mathsf{SP}(x_i^2, x_{i+1}^1) = \mathsf{SP}(x_i^2, v) \oplus v \oplus \mathsf{SP}(v, x_{i+1}^1).$$

Therefore,

$$\begin{aligned}
\mathsf{p} &= \mathsf{path}(\vec{\mathsf{p}}) \\
&= \mathsf{x}_1 \oplus \ldots \oplus \mathsf{x}_\mathsf{i} \oplus \mathsf{SP}(\mathsf{x}_\mathsf{i}^2, \mathsf{x}_{\mathsf{i}+1}^1) \oplus \mathsf{x}_{\mathsf{i}+1} \oplus \ldots \mathsf{x}_\mathsf{l} \\
&= \mathsf{x}_1 \oplus \ldots \oplus \mathsf{x}_\mathsf{i} \oplus \mathsf{SP}(\mathsf{x}_\mathsf{i}^2, \mathsf{v}) \oplus \mathsf{v} \oplus \mathsf{SP}(\mathsf{v}, \mathsf{x}_{\mathsf{i}+1}^1) \oplus \mathsf{x}_{\mathsf{i}+1} \oplus \ldots \mathsf{x}_\mathsf{l} \\
&= \mathsf{path}(\langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{i}, \mathsf{v}, \mathsf{x}_{\mathsf{i}+1}, \ldots, \mathsf{x}_\mathsf{l} \rangle).
\end{aligned}$$

□

Proposition 4.15. *Given a graph* G *and a path* p *on* G*, Algorithm 1 outputs a segmentation* $\vec{\mathsf{p}}$ *of* p*.*

Proof. Let $\mathsf{p} = (\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{l})$, $(\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_{\mathsf{l}+1})$ be the sequences of nodes visited by p and $\vec{\mathsf{p}} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{r} \rangle$. We start by proving by induction on $\mathsf{j} \geqslant 2$ that $\langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{j} \rangle$ is deterministic and $\mathsf{path}\langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{j} \rangle$ is a prefix of p.

Base case: $\mathsf{j} = 2$.

By construction we have that either $\mathsf{x}_1 = \mathsf{v}_1$ or $\mathsf{x}_1 = \mathsf{e}_1$. This is so because the only three lines where this element could have been inserted into $\vec{\mathsf{p}}$ are lines 9, 6 when $\mathsf{e} = \mathsf{e}_1$ or line 12 at the end. If it was on one of the lines 9 or 12 then $\mathsf{x}_1 = \mathsf{v}_1$. Otherwise $\mathsf{x}_1 = \mathsf{e}_1$.

Case 1: $\mathsf{x}_1 = \mathsf{v}_1$. Then the algorithm proceeds by iterating over the elements $\mathsf{e} \in \mathsf{E}(\mathsf{p})$ and only add the next element when we reach some edge $\mathsf{e}_\mathsf{i} = (\mathsf{u}, \mathsf{v}) = (\mathsf{v}_\mathsf{i}, \mathsf{v}_{\mathsf{i}+1})$ such that there exist multiple shortest paths between $\mathsf{r} = \mathsf{v}_1$ and $\mathsf{v}_{\mathsf{i}+1}$ or e_i does not belong to any shortest path starting at r. At this point we add x_2 which is either equal to v_i or e_i. In either case, since condition on line 4 ensures the uniqueness of the shortest path between $\mathsf{x}_1^2 = \mathsf{v}_1$ and $\mathsf{x}_2^1 = \mathsf{v}_\mathsf{i}$ it holds that $\langle \mathsf{x}_1, \mathsf{x}_2 \rangle$ is deterministic. To see that it corresponds to a prefix of p, we consider the two cases for x_2. If $\mathsf{x}_2 = \mathsf{v}_\mathsf{i}$ then

$$\mathsf{x}_1 \oplus \mathsf{SP}(\mathsf{x}_1^2, \mathsf{x}_2^1) \oplus \mathsf{x}_2 = \mathsf{v}_1 \oplus \mathsf{SP}(\mathsf{v}, \mathsf{v}_\mathsf{i}) \oplus \mathsf{v}_\mathsf{i} = \mathsf{SP}(\mathsf{v}_1, \mathsf{v}_\mathsf{i}) = \mathsf{e}_1 \oplus \ldots \oplus \mathsf{e}_{\mathsf{i}-1}$$

and if $\mathsf{x}_2 = \mathsf{e}_\mathsf{i}$ then

$$\mathsf{x}_1 \oplus \mathsf{SP}(\mathsf{x}_1^2, \mathsf{x}_2^1) \oplus \mathsf{x}_2 = \mathsf{SP}(\mathsf{v}_1, \mathsf{v}_\mathsf{i}) \oplus \mathsf{e}_\mathsf{i} = \mathsf{e}_1 \oplus \ldots \oplus \mathsf{e}_{\mathsf{i}-1} \oplus \mathsf{e}_\mathsf{i}.$$

Case 2: $\mathsf{x}_1 = \mathsf{e}_1$. In this case everything remains the same except that $\mathsf{r} = \mathsf{v}_2 = \mathsf{x}_1^2$. Therefore, if $\mathsf{x}_2 = \mathsf{v}_\mathsf{i}$ then

$$\mathsf{x}_1 \oplus \mathsf{SP}(\mathsf{x}_1^2, \mathsf{x}_2^1) \oplus \mathsf{x}_2 = \mathsf{e}_1 \oplus \mathsf{SP}(\mathsf{v}_2, \mathsf{v}_\mathsf{i}) = \mathsf{e}_1 \oplus \ldots \oplus \mathsf{e}_{\mathsf{i}-1}$$

and if $\mathsf{x}_2 = \mathsf{e}_\mathsf{i}$ then

$$\mathsf{x}_1 \oplus \mathsf{SP}(\mathsf{x}_1^2, \mathsf{x}_2^1) \oplus \mathsf{x}_2 = \mathsf{e}_1 \oplus \mathsf{SP}(\mathsf{v}_2, \mathsf{v}_\mathsf{i}) \oplus \mathsf{x}_2 = \mathsf{e}_1 \oplus \mathsf{e}_2 \oplus \ldots \oplus \mathsf{e}_{\mathsf{i}-1} \oplus \mathsf{e}_\mathsf{i}.$$

In any case, $\langle \mathsf{x}_1, \mathsf{x}_2 \rangle$ is deterministic and $\mathsf{path}\langle \mathsf{x}_1, \mathsf{x}_2 \rangle$ is a prefix of p. This completes the proof of the base case.

Induction step: Suppose that for some $\mathsf{j} \geqslant 2$ the sr-path $\langle \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{j} \rangle$ is deterministic and a prefix of p. Using analogous arguments as above, we can show that if e_i is the edge on the for loop in line 3 when x_j is added and e_k $(\mathsf{j} < \mathsf{k})$ the one when $\mathsf{x}_{\mathsf{j}+1}$ is added then $\langle \mathsf{x}_\mathsf{j}, \mathsf{x}_{\mathsf{j}+1} \rangle$ is deterministic with

$$\mathsf{path}\langle \mathsf{x}_\mathsf{j}, \mathsf{x}_{\mathsf{j}+1} \rangle = \begin{cases} \mathsf{e}_\mathsf{i} \oplus \ldots \oplus \mathsf{e}_{\mathsf{k}-1} & \text{if } \mathsf{x}_{\mathsf{j}+1} = \mathsf{v}_\mathsf{k} \\ \mathsf{e}_\mathsf{i} \oplus \ldots \oplus \mathsf{e}_\mathsf{k} & \text{if } \mathsf{x}_{\mathsf{j}+1} = \mathsf{e}_\mathsf{k} \end{cases}$$

Therefore $\langle x_1, \ldots, x_j, x_{j+1} \rangle$ is deterministic and

$$\mathsf{path}\langle x_1, \ldots, x_j, x_{j+1} \rangle = \begin{cases} e_1 \oplus \ldots \oplus e_{k-1} & \text{if } x_{j+1} = v_k \\ e_1 \oplus \ldots \oplus e_k & \text{if } x_{j+1} = e_k \end{cases}$$

is a prefix of p. This concludes the proof that for $j \geqslant 2$, $\langle x_1, \ldots, x_j \rangle$ is deterministic and $\mathsf{path}\langle x_1, \ldots, x_j \rangle$ is a prefix of p.

It remains to show that at the end of the algorithm $\vec{p}$ covers the whole path. Let x be the last element of $\vec{p}$ after the for loop ends. If $x^2 = v_{l+1}$ then there is nothing to prove. Otherwise at line 14 we add one node segment $x_l = v_{l+1}$. With the same argument as above, we know that $\mathsf{SP}(x^2, x_l)$ is unique and covers the remaining edges of p since the condition on line 4 was always false for the remaining edges. □

Definition 4.15. *Let G be a network and p a path on G visiting nodes $(v_1, \ldots, v_l)$. Given two nodes $v, u \in V(p)$ we write $v \leqslant_p u$ if and only if $v = v_i$ and $u = v_j$ with $i \leqslant j$.*

Lemma 4.16. *Let G be a network and p a path on G. Let $\vec{p} = \langle x_1, \ldots, x_l \rangle$ be the segmentation of p produced by Algorithm 1. Let $\vec{q} = \langle y_1, \ldots, y_r \rangle$ be any other segmentation of p. Then for any $i \in \{1, \ldots, l-1\}$, there exists $j \in \{1, \ldots, r\}$ such that $x_i^2 \leqslant_p y_j^2 \leqslant_p x_{i+1}^1$.*

Proof. Let $i \in \{1, \ldots, l-1\}$. Suppose that no such j exists. Then, since $\vec{q}$ is a segmentation of p, by Lemmas 4.12 and 4.13 there exists j such that $y_j^2 <_p x_i^2$ and $x_{i+1}^1 <_p y_{j+1}^1$. But then, since $\vec{q}$ is deterministic, there exists a unique shortest path between y_j^2 and y_{j+1}^1. This path will contain the shortest path between x_i^2 and x_{i+1}^1 plus, at least, the next edge of p with source at node x_{i+1}^1. This contradicts the fact that x_{i+1} is a segment of $\vec{p}$ since condition 4 would prevent it to be added to $\vec{p}$. Figure 4.7 illustrates this proof. □

Lemma 4.17. *Let G be a network and p a path on G. There exists a minimal segmentation of p such that all its adjacency segments belong to $\mathsf{adj}(G, p)$.*

Proof. Let G be a network and p a path on G. Let $\vec{p} = \langle x_1, \ldots, x_l \rangle$ be a minimal segmentation of p. Assume that x_i is an adjacency not in $\mathsf{adj}(G, p)$. Then $x_i = e \in E(p)$ such that e is the unique shortest path between (e^1, e^2). Therefore if we let $\vec{q} = \langle x_1, \ldots, x_{i-1}, e^1, e^2, x_{i+1}, \ldots, x_l \rangle$ we have

$$\begin{aligned} \mathsf{path}(\vec{q}) &= x_1 \oplus \ldots x_{i-1} \oplus e^1 \oplus \mathsf{SP}(e^1, e^2) \oplus e^2 \oplus x_{i+1} \oplus \ldots \oplus x_l \\ &= x_1 \oplus \ldots x_{i-1} \oplus \mathsf{SP}(e^1, e^2) \oplus x_{i+1} \oplus \ldots \oplus x_l \\ &= x_1 \oplus \ldots x_{i-1} \oplus e \oplus x_{i+1} \oplus \ldots \oplus x_l \\ &= \mathsf{path}(\vec{p}) = p. \end{aligned}$$

We conclude that $\vec{q}$ is a segmentation of p. Moreover, since $\mathsf{sr\text{-}cost}(\vec{q}) = (\mathsf{sr\text{-}cost}(\vec{p}) - 2) + 2 = \mathsf{sr\text{-}cost}(\vec{p})$ $\vec{q}$ is a minimal segmentation of p. By repeating this process we can obtain a minimal segmentation of p such that all of its adjacency segments are in $\mathsf{adj}(G, p)$. □

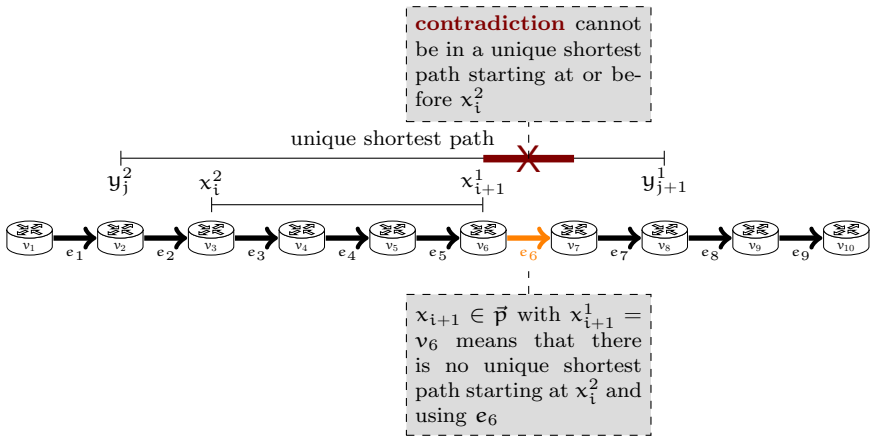


Figure 4.7: Illustration of what happens if there is no element y_j such that $x_i^2 \leqslant_p y_j^2 \leqslant_p x_{i+1}^1$ on the proof of Lemma 4.16.

Lemma 4.18. *Let* G *be a network and* p *a path on* G*. The set of adjacency segments in a segmentation computed by Algorithm 1 on input* p *is* $\mathsf{adj}(G, p)$*.*

Proof. Let $\vec{p} = \langle x_1, \ldots, x_l \rangle$ be segmentation produced by Algorithm 1 on input p. Suppose that $x_i = e = (u, v) \in E(p)$ is an adjacency segment. The only place where adjacency segments are added is on line 6. This happens if and only if $e \notin E(\mathsf{SP}(G, u))$ or $\delta^-(\mathsf{SP}(G, u), v) > 1$. If $e \notin E(\mathsf{SP}(G, u))$ then it means that e is not the shortest path between u and v. Otherwise, if $\delta^-(\mathsf{SP}(G, u), v) > 1$ it means that e is not the unique shortest path between its endpoints. Therefore $e \in \mathsf{adj}(G, p)$. □

Theorem 4.19. *Given a network* G *and a path* p *on* G*, Algorithm 1 outputs a minimal segmentation* $\vec{p}$ *of* p*.*

Proof. Let $\vec{p} = \langle x_1, \ldots, x_l \rangle$ be the segmentation produced by Algorithm 1 and $\vec{q} = \langle y_1, \ldots, y_r \rangle$ a minimal segmentation. By Lemmas 4.17 and 4.18 we can assume that $\vec{p}$ and $\vec{q}$ have the same adjacency segments. In particular, by Lemma 4.12 we have that $x_1 = y_1$. Since $\vec{p}$ and $\vec{q}$ have the same adjacency segments, if y_2 is an adjacency segment then so is x_2 and $x_2 = y_2$. Otherwise both are not segments. Lemma 4.16 says that, $x_1^2 \leqslant_p y_2^2 \leqslant_p x_2^1$. Let i be the largest index such that $x_1^2 \leqslant_p y_i^2 \leqslant_p x_2^1$. Then x_2 belongs to the unique shortest path between y_i^2 and y_{i+1}^1. Hence, by Lemma 4.14, $\langle y_1, \ldots, y_i, x_2, y_{i+1}, \ldots y_r \rangle$ is a segmentation of p. Finally, by applying Lemma 4.10 on $y_2, \ldots, y_i$ we conclude that $\langle y_1, x_2, y_{i+1}, \ldots y_r \rangle$ is also a segmentation of p.

With this argument we were able to replace the first segment of $\vec{q}$ not matching $\vec{p}$ with the corresponding segment of $\vec{p}$. By repeating this process for each node segment of $\vec{q}$ we are able to transform $\vec{q}$ into $\vec{p}$ while preserving the segment cost. Therefore $\vec{p}$ is also a minimal segmentation of p. □

In this section we provided an algorithm that is able to compute minimal segmentations efficiently. The time needed to compute the segmentation of a path depends on how we implement Algorithm 1. If we do not precompute

anything beforehand, we will need to perform at most $V(G)$ shortest path computations. If this is done is $O(|E(G)| \cdot \log(|V(G)|)$ with Dijkstra's algorithm we get a complexity of $O(|V(G)| \cdot |E(G)| \cdot \log(|V(G)|)$. However, if we precompute the all pairs shortest IGP shortest path matrix with Floyd Warshal's algorithm in $O(|V(G)|^3)$ we can then segment any path in linear time $O(|G|)$. Both these implementations are provided in our library.

4.4 SR reachability

In this section we focus on trying to understand how costly, in terms of segments, it is to connect two nodes with a deterministic sr-path. These results are very useful in order to provide lower bounds on the minimum cost of any segmentation of a path between two given nodes.

Since segment routing is a new technology, network topologies were not designed with SR in mind. Therefore it could very well be the case that current topologies require very long lists of segments to represent specific paths. For this reason we would like to have tools to analyse a given topology to see how suitable it is for SR. Consider the topology shown in Figure 4.8. If we follow shortest paths starting from router `e` (in green) and restrict ourselves to only visiting nodes whose shortest path from `e` is unique, then we will only be able to reach nodes `a`, `b`, `d`, `f`, `h` and `i` (in gray). Thus we can already say about this topology that it needs at least sr-paths of cost 3 to reach any given node with a deterministic sr-path from `e`. In other words, there exists a simple path starting at `e` that needs at least 3 segments to be represented with segment routing.

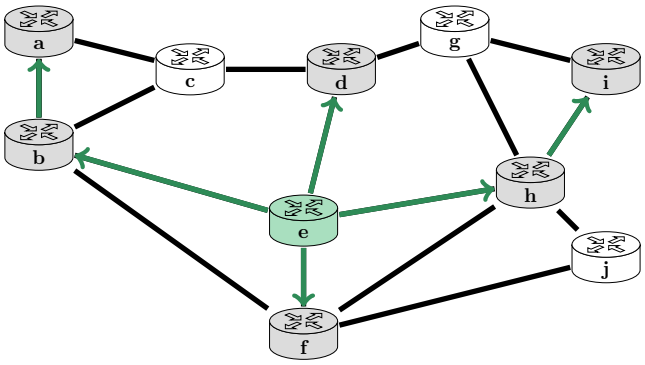


Figure 4.8: Shortest path reachability of node `e`

.

Of course this topology is very small and thus easy to analyse. To analyse larger topologies, we propose to define the k *deterministic reachability* of a node as the set nodes that can be reached from it with a deterministic sr-path of segment cost at most k. We will then propose efficient algorithms for computing these values for any given node v and segment cost k.

We start with the definition of shortest path deterministic reachability.

Definition 4.16. *Let* G *be a netwrok and* $v \in V(G)$*. We define the* shortest

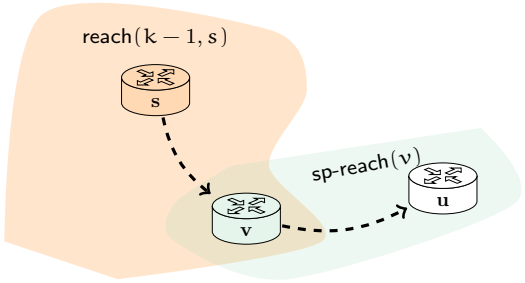


Figure 4.9: Illustration of the first part of the recurrence of reach(k, s)

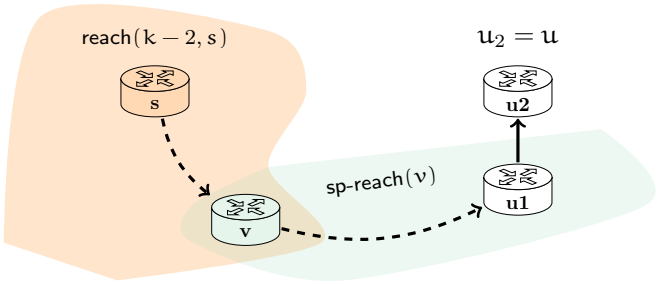


Figure 4.10: Illustration of the second part of the recurrence of reach(k, s)

path deterministic reachability *of* v *as*

$$\textit{sp-reach}(v) = \{u \in V(G) \mid \textit{there is a unique shortest path between } v \textit{ and } u\}$$

For example, on Figure 4.8 we see that $\mathsf{sp\text{-}reach}(e) = \{a, b, d, e, f, h, i\}$. As we prove in Theorem 4.20, this definition constitutes the basic building block that allows to compute the general deterministic reachability of a node.

Definition 4.17. *Let* G *be a network and* $s \in V(G)$*. For any integer* $k \geqslant 1$ *we define*

$$\textit{reach}(k, s) = \textit{set of nodes } v \in V(G) \textit{ such that there exists a deterministic sr-path } \vec{p} \textit{ from } s \textit{ to } v \textit{ of segment cost at most } k$$

The following theorem provides a recurrence relation for reach(k, s) in terms of smaller values of k and other nodes in the network.

The general intuition for computing the nodes in reach(k, s) is provided in Figures 4.9 and 4.10. Recall that a node u belongs to reach(k, s) if there is a deterministic sr-path from s to u of segment cost at most k. Such a path can have two forms, depending on the type of its last segment. If it is a node segment, then it is composed by a deterministic sr-path of cost at most $k-1$ to some node v and then appended a node segment on u (provided that there is a unique shortest path between v and u) as shown in Figure 4.9. Otherwise, it will be made of a deterministic sr-path of cost at most $k-2$ to some node v

and then will be appended an adjacency segment over some edge ending in u as shown in Figure 4.10. The following theorem formalizes this intuition.

Theorem 4.20. *Let* G *be a network,* $s \in V(G)$ *and an integer* $k \geqslant 3$*. It holds that*

$$\begin{aligned}
&\mathit{reach}(1, s) = \{s\} \\
&\mathit{reach}(2, s) = \{v \in V(G) \mid \mathsf{SP}(s, v) \textit{ is a path}\} \cup \{u \mid (s, u) \in E(G)\} \\
&\mathit{reach}(k, s) = \bigcup_{v \in \mathit{reach}(k-1, s)} \mathit{sp\text{-}reach}(v) \quad \cup \\
&\qquad\qquad \bigcup_{v \in \mathit{reach}(k-2, s)} \{u_2 \mid (u_1, u_2) \in E(G) \wedge u_1 \in \mathit{sp\text{-}reach}(v)\}
\end{aligned}$$

Proof. For $k = 1$ the only sr-path of cost 1 starting from s is $\langle s \rangle$. Thus $\mathsf{reach}(1, s) = \{s\}$. For $k = 2$, a sr-path of cost 2 starting at s has either the form $\langle s, v \rangle$ where $v \in V(G)$ or $\langle e \rangle$ where $e \in \delta^+(s)$. In the first case, since we need the sr-path to be deterministic, only nodes $v \in V(G)$ such that $\mathsf{SP}(s, v)$ is a path yield a deterministic sr-path. In the second case, a path of the form $\langle e \rangle$ is always deterministic so each such edge yields a valid path.

Let $k \geqslant 3$.

($\subseteq$) Suppose that $u \in \mathsf{reach}(k, s)$. Then there exists a deterministic sr-path $\vec{p} = \langle x_1, \ldots, x_l \rangle$ from s to u of segment cost at most k.

Case 1: $x_l = u$ is a node segment. Then $\langle x_1, \ldots, x_{l-1} \rangle$ is a deterministic sr-path from s to x^2_{l-1} with segment cost at most $k-1$. Hence $x^2_{l-1} \in \mathsf{reach}(k-1, s)$. By determinism of $\vec{p}$, there is a unique shortest path between x^2_{l-1} and $x^1_l = x_l$. Therefore, $u = x_l \in \mathsf{sp\text{-}reach}(x^2_{l-1})$. Thus letting $v = x^2_{l-1}$, we have that $v \in \mathsf{reach}(k-1, s)$ and $u \in \mathsf{sp\text{-}reach}(v)$ so

$$u \in \bigcup_{v \in \mathsf{reach}(k-1, s)} \mathsf{sp\text{-}reach}(v).$$

Case 2: x_l is an adjacency segment. Write $x_l = (u_1, u_2)$. Then $\langle x_1, \ldots, x_{l-1} \rangle$ is a deterministic sr-path from s to x^2_{l-1} with segment cost at most $k-2$ and by determinism, $u_1 = x^1_l \in \mathsf{sp\text{-}reach}(x^2_{l-1})$. Thus letting $v = x^2_{l-1}$, we have that $v \in \mathsf{reach}(k-2, s)$ and $u_1 \in \mathsf{sp\text{-}reach}(v)$ proving that

$$u = u_2 \in \bigcup_{v \in \mathsf{reach}(k-2, s)} \{u_2 \mid (u_1, u_2) \in E(G) \wedge u_1 \in \mathsf{sp\text{-}reach}(v)\}$$

($\supseteq$) Let $u \in \bigcup_{v \in \mathsf{reach}(k-1, s)} \mathsf{sp\text{-}reach}(v)$. Then there exists $v \in \mathsf{reach}(k-1, s)$ such that $u \in \mathsf{sp\text{-}reach}(v)$. Let $\vec{p} = \langle x_1, \ldots, x_l \rangle$ be a deterministic sr-path from s to v with $\mathsf{sr\text{-}cost}(\vec{p}) \leqslant k-1$. Since $u \in \mathsf{sp\text{-}reach}(v)$, there is a unique shortest path from v to u so the sr-path $\langle x_1, \ldots, x_l, u \rangle$ is a deterministic sr-path from s to u with segment cost at most $k-1+1 = k$. Thus $u \in \mathsf{reach}(k, s)$.

Let $u \in \bigcup_{v \in \mathsf{reach}(k-2, s)} \{u_2 \mid (u_1, u_2) \in E(G) \wedge u_1 \in \mathsf{sp\text{-}reach}(v)\}$. Then there exists $v \in \mathsf{reach}(k-2, s)$ and $(u_1, u_2) \in E(G)$ such that $u_1 \in \mathsf{sp\text{-}reach}(v)$ and $u_2 = u$. Let $\vec{p} = \langle x_1, \ldots, x_l \rangle$ be a sr-path from s to v with segment cost at most $k-2$. Then $\langle x_1, \ldots, x_l, (u_1, u_2) \rangle$ is a deterministic sr-path from s to u with segment cost at most $k-2+2 = k$ from s to u so $u \in \mathsf{reach}(k, v)$. □

Algorithm 2 compute-reach (g)

```
 1: reach ← Matrix(3, |V(G)|, Set())
 2: for v ∈ g.V() do
 3:    reach(1, v).add(v)
 4: spreach ← Array(|V(G)|)
 5: for v ∈ g.V() do
 6:    spreach(v) ← compute-sp-reach(g, v)
 7:    reach(2, v).or(spreach(v))
 8:    for (u1, u2) ∈ δ+(g, s) do
 9:       reach(2, v).add(u2)
10: k ← 3
11: while ∃s ∈ V(g) reach(k − 1, v) ≠ V(G) do
12:    for s ∈ V(G) do
13:       if reach(k − 1, s) = V(G) then
14:          reach(k, s) = V(G)
15:       else
16:          reach.addRow(Set())
17:          for v ∈ reach(k − 1, s) do
18:             reach(s, k).or(spreach(v))
19:          for v ∈ reach(k − 2, s) do
20:             for (u1, u2) ∈ E(g) such that u1 ∈ spreach(v) do
21:                reach(s, k).add(u2)
22: return nreach
```

Using Theorem 4.20 we can easily write down an algorithm for computing $\mathsf{reach}(\mathsf{k}, \mathsf{s})$ for all k, s. Algorithm 2 is designed so that at the end of its execution it computes a $\mathsf{reach}(\mathsf{k}, \mathsf{s})$ as a matrix for all relevant values of k and s. It will go on until we reach a value k' such that $\mathsf{reach}(\mathsf{k}', \mathsf{s}) = \mathsf{V}(\mathsf{G})$ for all $\mathsf{s} \in \mathsf{V}(\mathsf{G})$. For $\mathsf{k} > \mathsf{k}'$ we know that $\mathsf{reach}(\mathsf{k}, \mathsf{s}) = \mathsf{reach}(\mathsf{k}', \mathsf{s})$ so there is no point in computing it. Its correctness is provided by Theorem 4.20 since it quite literally implements the recurrences provided by the theorem. It uses Algorithm 3 to compute the deterministic shortest path reach defined in Definition 4.16. This algorithms uses Dijkstra's algorithm to compute the shortest path DAG rooted at the given node s and the performs a BFS to compute the set of nodes in this DAG that are reachable from s via a unique shortest path. On line 7 of Algorithm 3 we ensure that only nodes with unique shortest paths from s are visited by only adding new nodes to the queue when their in-degree in the shortest path DAG is equal to 1, as those nodes are the only ones for which unique shortest paths exist.

Lemma 4.21. *Let* G *be a network,* $\mathsf{u}, \mathsf{v} \in \mathsf{V}(\mathsf{G})$ *and* k *a non-negative integer. If* $\mathsf{u} \notin \mathsf{reach}(\mathsf{k}, \mathsf{v})$ *then the minimum segment cost sr-path between* v *and* u *has segment cost at least* $\mathsf{k} + 1$*.*

Proof. By definition, if there exists a sr-path $\vec{\mathsf{p}}$ between v and u such that $\mathsf{sr\text{-}cost}(\vec{\mathsf{p}}) \leqslant \mathsf{k}$ then $\mathsf{u} \in \mathsf{reach}(\mathsf{k}, \mathsf{v})$. Therefore, if $\mathsf{u} \notin \mathsf{reach}(\mathsf{k}, \mathsf{v})$, any sr-path between v and u must have a segment cost larger than k. □

Corollary 4.22. *Let* G *be a network,* $\mathsf{u}, \mathsf{v} \in \mathsf{V}(\mathsf{G})$ *and* k *a non-negative integer. If* $\mathsf{u} \notin \mathsf{reach}(\mathsf{k}, \mathsf{v})$ *then any path on* G *from* v *to* u *requires at least* k *segments in its minimal segmentation.*

Algorithm 3 compute-sp-reach (g, s)

```
1: dag ← dijkstra-dag(v)
2: visited ← Set()
3: Q ← Queue()
4: Q.add(s)
5: while |Q| > 0 do
6:    cur ← Q.poll()
7:    for (u1, u2) ∈ δ+(dag, cur) such that u2 ∉ visited and |δ−(dag, u2)| = 1 do
8:       Q.add(u2)
9:       visited.add(u2)
10: return visited
```

Proof. Immediate from Lemma 4.21. □

The two following measures are interesting to evaluate how suitable a topology is for segment routing.

Definition 4.18. *We denote the maximum* k *for which there exists* $v \in V(G)$ *such that* $\mathit{reach}(k, n) \neq V(G)$ *as* $k_{max}(G)$ *and the minimum* k *such that there exists* $v \in V(G)$ *such that* $\mathit{reach}(k, v) = V(G)$ *as* $k_{min}(G)$.

The value of $k_{max}(G)$ describes the worst case reachability of any node in the network. By Corollary 4.22, it means that there exists a pair of nodes u, v such that any path from u to v requires at last $k_{max}(G)$ segments in its minimal segmentation. Therefore, in a network where routers do not support $k_{max}(G)$ segments, any solution to a problem with a path from u to v cannot be implemented on that network. Also, $k_{max}(G)+1$ indicates the minimum number of segments that routers need to suppose in order to be able to implement a multi-cast tree rooted at any node (a spanning tree rooted at that node). On the other hand, $k_{min}(G)$ gives the maximum value for which some node v is able to reach every other node. For a network, this means that there exists a multi-cast tree rooted at v such that any root to leaf path is segmentable with at most k segments.

We computed these values for every topology in our dataset. Figure 4.11 shows the percentage of topologies for each value of $k_{max}(G)$ and $k_{min}(G)$.

In the figure we observe that for 20% of the instances, $k_{max}(G) = 1$. This means that with two segments, every node can reach every other node with a deterministic sr-path. In other words, 20% of the topologies do not contain ECMP which matches our analysis of the topologies in Chapter 3. We also observe that in the worst case we need up to 12 segments to connect some pairs of nodes with a determinism sr-path. This value exceeds the capacity of most commercial routers but very few topologies are in this case. For 90% of the topologies, with at most 5 segments any pair of routers can be connected with a deterministic sr-path.

We observe that for 87% of the topologies, there exists some router on the network which is able to reach each other router with a deterministic sr-path of segment cost at most 3. We also see that we never need a sr-path path with segment cost larger than 7 in order to find a root for a multi-cast tree implementable with segment routing.

These results are quite positive and indicate that path based solutions to networking optimization problems are likely to be implementable with segment

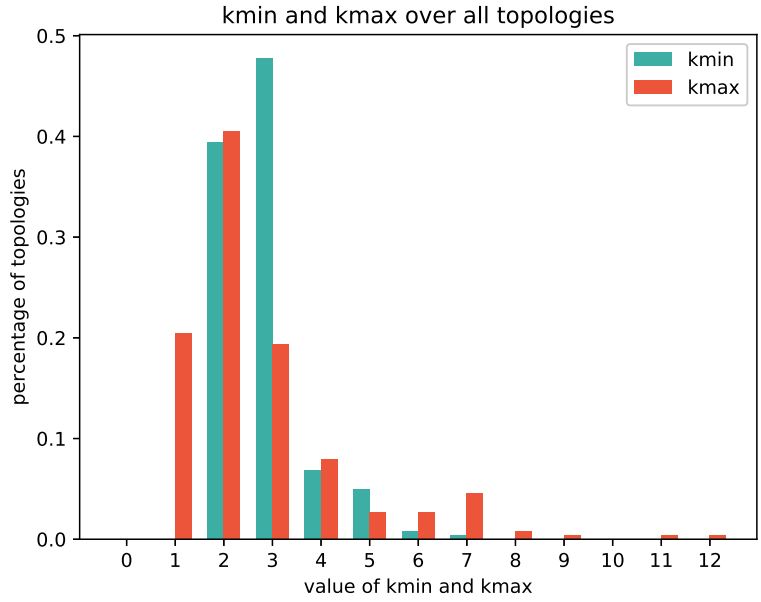


Figure 4.11: Percentage of topologies for each given value of $k_{min}(G)$ and $k_{max}(G)$.

routing. This result is not a definitive answer as the measure that we would really like to compute in order to be able to answer these questions is the maximum number of segments needed to represent any simple path in the original network. This would give an upper bound on the number of segments required for implementing any path based solution with segment routing on each topology. Unfortunately computing this value is NP-hard as we shown next.

Problem 2 (Maximum segmentation path)

Given a network G *compute the maximum value of* sr-cost(min-seg(p)) *such that* p *is a simple path in* G.

In order to prove that Problem 2 is NP-hard we need to defined some related problems first. The first problem that we consider is the same except that we only focus on paths between two given nodes.

Problem 3 (Maximum segmentation s-t path)

Given a network G *and* $s, t \in V(G)$ *find a simple* s-t *path* p *such that* sr-cost(min-seg(p)) *is maximum.*

Problem 4 (Unit weights longest path between nodes)

Given a graph G *and* $s, t \in V(G)$ *compute the longest path from* s *to* t *in terms*

of the number of links in the path.

Theorem 4.23. *Problem 4 is NP-hard.*

Proof. This is a known result. A proof can be found, for instance, in Corollary 8.11a from [45]. □

Theorem 4.24. *Problem 3 is NP-hard.*

Proof. We are going to prove that Problem 3 is NP-hardby showing that if we could solve it in polynomial time, then we could also solve Problem 4 in polynomial time.

Let G, s, t be an instance of Problem 4. Build a graph H which is a copy of G to which we add the following. For each edge $(u, v) \in E(G)$, add a node uv and two edges (u, uv) and (uv, v) both of weight 1. Figure 4.12 illustrates this transformation.

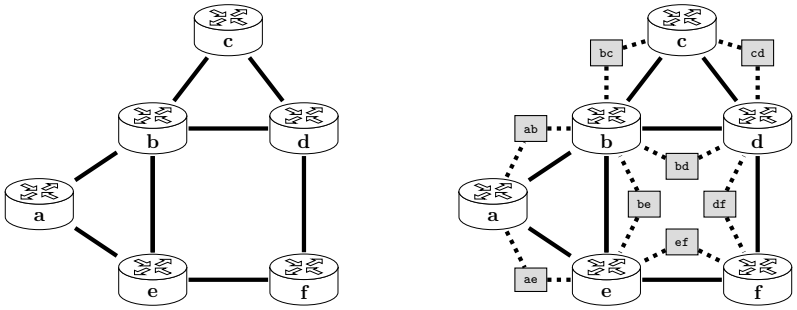
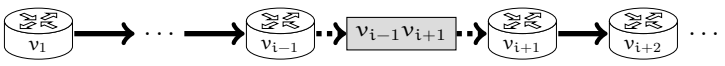


Figure 4.12: Example of the transformation. Solid edges have weight 2 whereas dotted edges have weight 2.

Let p be a path on H from s to t such that sr-cost(min-seg(p)) is maximum. We start by proving that p does not cross any of the new nodes, that is, only crosses nodes in $V(G) \cap V(H)$. Suppose that p visits the node sequence $(v_1, \ldots, v_l)$ and let i be the smallest index such that $v_i \notin V(G)$. Since p is a path from s to t and $s, t \in V(G)$ we have that $1 < i < l$. Thus we can write $v_i = v_{i-1}v_{i+1}$ and $p = (v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_l) = (v_1, \ldots, v_{i-1}, v_{i-1}v_{i+1}, v_{i+1} \ldots, v_l)$. We need to consider three cases.

Case 1: $v_{i+2} \in V(G)$. In this case path p has the form:



By construction, every link of G that is traversed requires an adjacency segment in its minimal segmentation (because of ECMP). Therefore, the minimal segmentation of this path is

$$\langle (v_1, v_2), (v_2, v_3), \ldots, (v_{i-2}, v_{i-1}), xy, (v_{i+1}, v_{i+2}), \ldots \rangle.$$

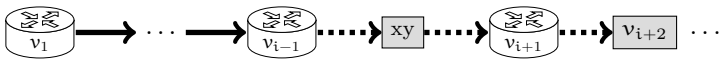
On the other hand, if we remove element xy from the path, we obtain a simple path p' with the same minimal segmentation except that xy is replaced by the

adjacency segment (x, y) giving

$$\langle (v_1, v_2), (v_2, v_3), \ldots, (v_{i-2}, x), (x, y), (y, v_{i+2}), \ldots \rangle.$$

This segmentation costs one more than the segmentation of p. Since p' is also a path from s to t, this contradicts the fact that p is a path from s to t that maximizes $\mathsf{sr\text{-}cost}(\mathsf{min\text{-}seg}(p))$.

Case 2: $v_{i+2} \notin V(G)$. In this case, p has the following form:



The minimal segmentation of p will be

$$\langle (v_1, v_2), (v_2, v_3), \ldots, (v_{i-2}, x), xy, v_{i+2}, \ldots \rangle.$$

As before, if we remove xy from p we obtain a simple path p' from s to t with minimal segmentation equal to

$$\langle (v_1, v_2), (v_2, v_3), \ldots, (v_{i-2}, x), (x, y), v_{i+2}, \ldots \rangle.$$

This segmentation costs one more than the minimal segmentation of p so as above, we conclude that p does not minimize $\mathsf{sr\text{-}cost}(\mathsf{min\text{-}seg}(p))$.

These two cases cover all possibilities so we conclude that the path p on H that maximizes $\mathsf{sr\text{-}cost}(\mathsf{min\text{-}seg}(p))$ has all of its nodes in $V(G)$. Therefore, this is also a path in G. Furthermore, because of ECMP, that the minimal segmentation of p will contain all of its edges as adjacency segments. Hence $\mathsf{sr\text{-}cost}(\mathsf{min\text{-}seg}(p)) = 2|E(p)|$ so it follows that if can find a path on H that maximizes $\mathsf{sr\text{-}cost}(\mathsf{min\text{-}seg}(p))$ we can find a path on G that maximizes $E(p)$ which completes the proof.

□

Corollary 4.25. *Problem 2 is NP-hard.*

Proof. Let G, s, t be an instance of Problem 3. Let $m = E(G)$. We build a graph by adding a path $(x_1, \ldots, x_m, s)$ connecting to s whose minimal segmentation has segment cost $2m$ and a path $(t, y_1, \ldots, y_m)$ going out of t whose minimal segmentation also has segment cost $2m$. This is achieved by adding also nodes $x'_1, \ldots, x'_m$ and $y'_1, \ldots, y'_m$ and edges (x_i, x'_i) and (x'_i, x_{i+1}) of cost 1 to ensure that adjacency segments are required to segment the two paths mentioned above. Figure 4.13 illustrates this. Dashed edges represent edges of cost 1 and solid edges represent edge of cost 2.

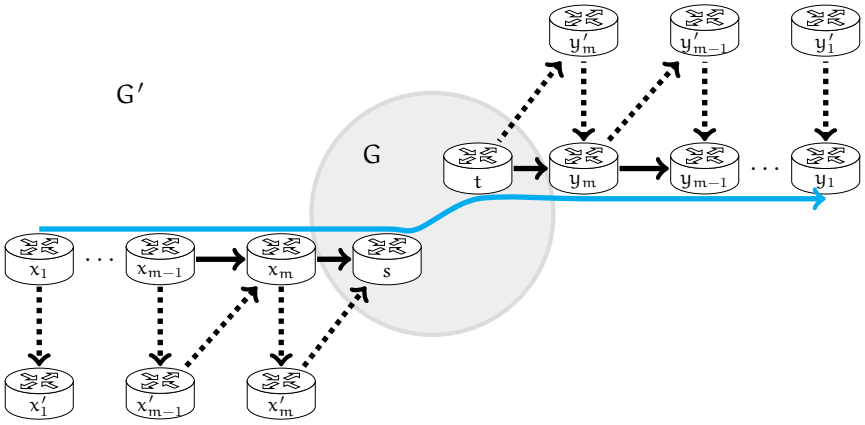


Figure 4.13: Example of the transformation.

By lemma 4.8, the maximum cost of a minimal segmentation in G is at most $2m$. Therefore, the maximum cost of a minimal segmentation of a path in G' must be a path from x_1 to y_1 since just this part of the path will have segment cost at least $4m$. Since the path from x_1 passes by s, to each y_1 we need to pass by t, we will achieve the maximum segment cost by connecting s and t by path with maximum segmentation cost. Thus the solution to Problem 2 on G' is a path of cost $x + 4m$ where x is the cost of the solution to problem 3 on input G, s, t. We conclude that if we could solve Problem 2 in polynomial time then we could also solve Problem 3 in polynomial time. Therefore, by Theorem 4.24, we conclude that 2 is NP-hard.

□

Conclusion

In this chapter we provided a first formalization of segment routing that comprises both node and adjacency segments. We provided an algorithm for computing minimal segmentations. This algorithms opens the possibility of using traditional graph algorithm for solving optimization problems over a network and then using it to segment and implement the solution on a segment routed network. The advantage of this is that it allows to leverage the long lasting graph theory and algorithms that have been developed without needing to extend those results to segment routing. The drawback of course is that this approach yields no guarantees over the number of segments needed in the end. Maybe in the future, when the number of segments in the segment stack becomes less of a limitation most solutions will eventually switch to this approach. However for the time being, we still need solutions that take these constraints into account.

We also developed a reachability theory which constitutes the first steps towards having tools to analyze how well a network is suited for segment routing. We will see in Chapter 7 that this reachability theory makes it possible to provide minimum segment cost cycle covers of a network in polynomial time.

Chapter 5

Optimal sr-paths

Introduction

In this chapter we study several problems related to computing optimal sr-paths between two givens nodes. We start by defining weight functions for sr-paths and propose an algorithm for computing sr-paths of minimal weight with respect to such a weight function and show how we can achieve low latency in a segment routed network. Next we study the reverse problem of finding a path of maximum weight. Finally, we show how to compute sr-paths with maximum capacity. This is useful to be able to route demands on a network without exceeding link capacities.

5.1 Minimum weight sr-path

The problem of computing minimum weight sr-paths appears in some form or another as a sub-problem in each of the three applications of segment routing that we study in this thesis. It also has interesting applications in itself such as computing sr-paths with minimum latency in the network or sr-paths with maximum bandwidth.

In order to be able to make sense of what we mean by a minimum weight sr-path, we need to define weight functions for sr-paths which we call sr-metrics. These will essentially be generalizations of weight function on graphs. To avoid confusion with minimum segment cost sr-paths, we will always be careful to not interchange the words weight and cost. So whenever we mention the weight of a sr-path we mean the value of some sr-metric when evaluated on that sr-path and whenever we mention its cost we are thinking in terms of segments.

Definition 5.1. *Let* G *be a network. A* sr-metric *is a function* $w : (V(G) \times V(G)) \cup E(G) \to \mathbb{R} \cup \{\infty\}$ *such that if there are no paths between* u *and* v *on* G *then* $w(u, v) = \infty$. *Given a sr-path* $\vec{p} = \langle e_1, \dots, e_n \rangle$ *we define the weight* $w(\vec{p})$ *as*

$$w(\vec{p}) = \sum_{i=2}^{n} w(x_{i-1}^2, x_i^1) + \sum_{i : x_i \in E(G)} w(x_i).$$

The way to think about a sr-metric is that $w(u, v)$ for $u, v \in V(G)$ defines what we pay for traversing the shortest paths between u and v and $w(e)$ for

$e \in E(G)$ defines the weight of traversing edge e with an adjacency segment. The problem of finding minimum cost sr-paths with respect to some metric is defined as follows.

Problem 1 (Minimum weight sr-path)

Input: *A network* G*, a sr-metric* w*,* $k \in \mathbb{N}$ *and two distinct nodes* $s, t \in V(G)$*.*
Output: *A sr-path* $\vec{p} \in \vec{\mathcal{P}}_k(s,t)$ *such that* $w(\vec{p})$ *is minimal.*

5.1.1 General algorithm

We propose in this section an algorithm for computing minimum weight sr-paths. The idea of this algorithm comes from the Bellman-Ford shortest path algorithm [12]. The Bellman-Ford algorithm is a dynamic programming (DP) algorithm for computing shortest paths on graphs with both positive and negative edge weights from a given source s. At first sight, this might seem totally different from what we are doing here. However if we look at the original dynamic programming formulation we see that it is actually computing shortest paths of increasing lengths, in terms of edges. In a nutshell, they define DP state spaces such that

$$sol(i, v) = \text{the weight of the shortest path from } s \text{ to } v \text{ using at most } i \text{ edges.}$$

We are going to define a very similar state space in our solution. The main difference is that the i parameter will correspond the segment cost of the sr-path rather the a number of edges. We thus define the following DP state space

$$sol(i, v) = \text{the minimum weight sr-path from } s \text{ to } v \text{ with segment cost at most } i.$$

The solution minimum weight sr-path problem is by definition $sol(k, t)$. For $i = 0$ the only possible sr-path of segment cost 0 from s to any node is $\langle s \rangle$. Thus, we have that $sol(0, s) = 0$ and $sol(0, v) = \infty$ for $v \neq s$.

The next step is to express $sol(i, *)$ in terms of $sol(j, s)$ with $j < i$. The idea for doing this it to analyze the structure of a sr-path of cost i. There are three possibilities to reach a node v with a sr-path of segment cost at most i. These three possibilities are illustrated in Figure 5.1. The first way is to simply not use the extra segment and reach v in the best way using a sr-path of segment cost at most $i - 1$. The second possibility is to reach some node $u \in V$ using a sr-path of segment cost at most $i - 1$ and then use one extra node segment to reach v incurring an extra cost of $w(u, v)$. Finally, we can use a sr-path of cost at most $i - 2$ to any node r and then append to it an adjacency segment (u, v) where u is a in-neighbor of v. Note that nothing prevents r from being equal to u. In this case it simply means that we append the adjacency (u, v) to a path that already ends at node u. The cost increment of doing so is the cost of traversing the shortest path from r to u plus the cost of the link (u, v).

The next theorem formalizes this intuition.

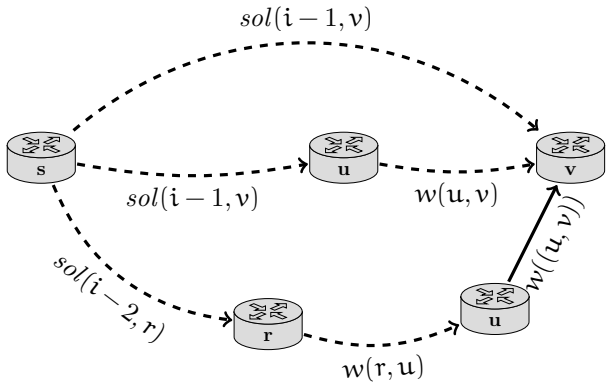


Figure 5.1: Illustration of the *sol* recurrence

Theorem 5.1. *For* $i \geqslant 1$ *and* $x \in V(G)$ *it holds that*

$$sol(i,v) = \min \begin{cases} sol(i-1,v) & \\ sol(i-1,u) + w(u,v) & \text{s.t } y \in V \\ sol(i-2,r) + w(r,u) + w((u,v)) & \text{s.t } r \in V \end{cases}$$

where the third value is only defined for $i \geqslant 2$ *(set to* ∞*) otherwise.*

Proof. Let $\vec{p} = \langle x_1, \ldots, x_n \rangle$ be a minimum weight sr-path from s to v of segment cost at most i. If $\mathsf{sr\text{-}cost}(\vec{p}) < i$ then by definition $w(\vec{p}) = sol(i-1,v)$. Suppose then that $\mathsf{sr\text{-}cost}(\vec{p}) = i$. We consider two cases.

Case 1: $x_n \in V(G)$. In this case $\vec{q} = \langle x_1, \ldots, x_{n-1} \rangle$ is a sr-path from s to some node $u = x_{n-1}^2$ such that $w(\vec{q}) = i - 1$. It must be the case that $w(\vec{q}) = sol(i-1,u)$ or otherwise we could replace it by a better sr-path and obtain a sr-path from s to v with lower weight. Therefore, by definition of w, we have

$$sol(i,v) = w(\vec{p}) = w(\vec{q}) + w(u,v) = sol(i-1,u) + w(u,v).$$

Case 2: $x_n \in E(G)$. This case is similar to the previous one. This time $\vec{q} = \langle x_1, \ldots, x_{n-1} \rangle$ is a sr-path from s to some node $r = x_{n-1}^2$ such that $w(\vec{q}) = i - 2$. Again, it must be the case that $w(\vec{q}) = sol(i-2,r)$. By definition of w, if we let $u = x_n^1$, it holds that

$$sol(i,x) = w(\vec{p}) = w(\vec{q}) + w(x_{n-1}^2, x_n^1) + w(x_n) = sol(i-1,y) + w(r,u) + w((u,v)).$$

□

Since the values of $sol(i,*)$ depend only on $sol(i-1,*)$, we can compute all these values by iterating in increasing order of i. For each i, evaluating one state of the form (i,v) takes $O(|V(G)| + |V(G)| \cdot |\delta^-(v)|)$. Hence, given i, the cost of evaluating state (i,v) all $v \in V(G)$ is $O(|V(G)|^2 + |V(G)| \cdot |E(G)|)$ making the total time complexity be $O(k \cdot |V(G)| \cdot |E(G)|)$. A formalization of this algorithm is provided as Algorithm 4. Since this algorithm runs in polynomial time, we have the following result.

Proposition 5.2. *Problem 1 can be solved in polynomial time.*

Algorithm 4 minWeightSrPath $(g, orig, dest, w, maxSeg)$

```
1: sol ← matrix(maxSeg + 1, g.V(), ∞)
2: sol(0, orig) = 0
3: parent ← matrix(maxSeg + 1, g.V(), null)
4: for i = 1, ..., maxSeg do
5:    for cur ∈ g.V() do
6:       sol(i, cur) ← sol(i − 1, cur)
7:       parent(i, cur) ← parent(i − 1, cur)
8:       for prev ∈ g.V() \ {cur} do
9:          if sol(i − 1, prev) + w(prev, cur) < sol(i, cur) then
10:            [we reach cur using node segment]
11:            sol(i, cur) = sol(i − 1, prev) + w(prev, cur)
12:            parent(i, cur) ← prev
13:      [the third case is only defined for i ⩾ 2]
14:      if i ⩾ 2 then
15:         for e = (orig, dest) ∈ g.inEdges(cur) do
16:            for prev ∈ g.V() \ {cur} do
17:               if sol(i − 2, prev) + w(prev, orig) + w(e) < sol(i, cur) then
18:                  [we reach cur using adjacency]
19:                  sol(i, cur) = sol(i − 2, prev) + w(prev, orig) + w(e)
20:                  parent(i, cur) ← e
21: opt ← sol(maxSeg, dest)
22: k ← maxSeg
23: while k − 1 ⩾ 0 and sol(k − 1, dest) = opt do
24:    k ← k − 1
25: return SrPath(parent, orig, dest, k, opt)
```

5.1.2 Achieving minimum latency with SR

In this section we explore the problem of computing and implementing minimum latency paths in a network. In a typical network, routing is done using shortest path routing with respect to the IGP costs configured on the links. These costs can be, to some extent, arbitrary and need not be related with the link latencies. This means that it can happen that a suboptimal path is used for forwarding packets, with respect to the total time it takes the packets to travel from the origin to its destination.

In Figure 5.2 we illustrate such an example in a real network. The IGP shortest path between routers `a` and `c` is the one shown in the solid blue edges and has a total latency of 101.4 milliseconds. On the other hand, the path $(\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{c})$, represented by the green edges, only has a latency of 58.2 milliseconds, that is, it is 74% faster. This path can be implemented with segment routing by adding, for instance, a detour towards router `d` before forwarding the traffic to router `c`.

Although this example shows that SR can sometimes make it possible to find much more efficient paths, our experiments show that the average gain is actually much lower than the 74% shown above and that this is an outlier result. For every topology in our data set, we computed for every pair of distinct nodes, the minimum latency path between them and computed the ratio between this value and the latency of the IGP shortest path. Figure 5.3 shows a box plot of these ratios. On the x-axis, we either have a single topology or a topology group, and on the y-axis we have the ratio between the nominal latency (latency of

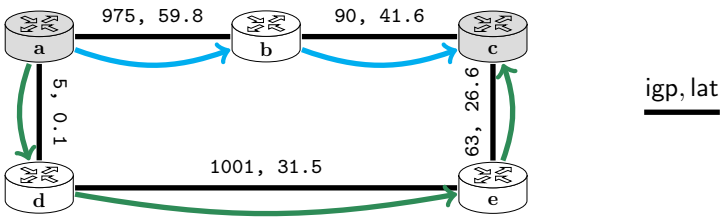


Figure 5.2: A small subgraph of a real network where the IGP shortest path between two nodes paths is 74% slower than the optimal path.

the IGP shortest path) and minimum latency of any path connecting the same origin and destination. We aggregated the results for the Zoo topologies because this data set contains over 200 topologies making it impossible show all of them. The last box shows the aggregated results over all topologies. We can observe that the average gain is only 8%, much lower than our example.

However, it is still interesting to study the problem of finding minimum latency segment routing paths for two reasons. First, even if the gain is not as significant as our above example on average, there is still something to be gained in terms of latency when using SR over IGP shortest path routing as the overhead of doing so is quite small since SR does not require routers to maintain state. Second, as we will see later, this problem appears in some form or another as a subproblem of other problems that we have tackled in this thesis.

Shortest path plus segmentation

The first way to solve this problem with segment routing is to simply use a shortest path algorithm such as Dijkstra's algorithm to compute the minimum latency path between the origin and the destination. Then, in order to implement that path with segment routing, we can use the minimal segmentation algorithm proposed in Chapter 4 to segment that path.

The problem with doing this is that we have no control over the number of segments in the final path. If could be the case that the minimum latency path actually requires a huge amount of segments to be implemented. Figure 5.4 shows the distribution of the number of segments required in the segmentation of the minimum latency paths over all pairs of nodes in the topologies from our data set.

These results indicate that on average, there is some correlation between the IGP shortest paths and the minimum latency paths. For 20% of the pairs over all topologies, the IGP shortest path matches the minimum latency path. With about 5 segments we can already route over 96% of the minimum latency paths. Thus we can say that on average computing the minimum latency path and then segmenting it will yield a path that requires a small segment stack and this is likely to be implementable on the network.

However, there are some topologies where this is far from true. Figure 5.5

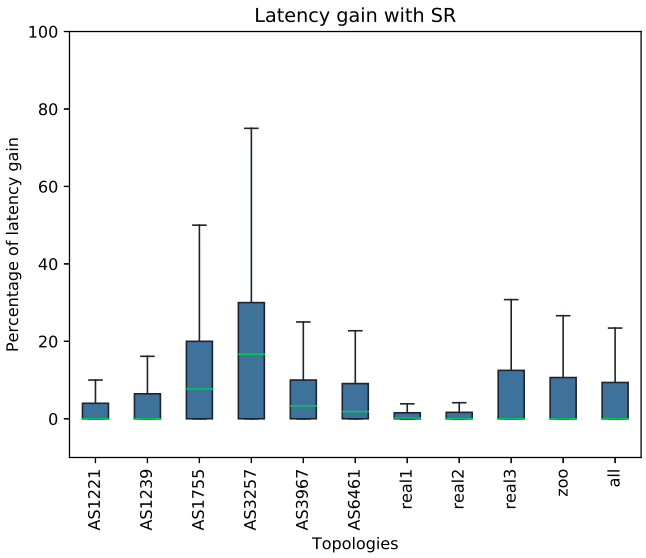


Figure 5.3: Latecy gain in percentage.

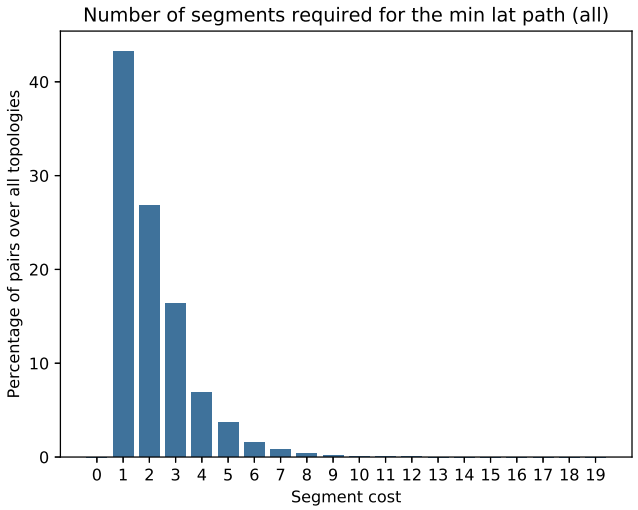


Figure 5.4: Number of segments required in the minimum latency paths over all topologies.

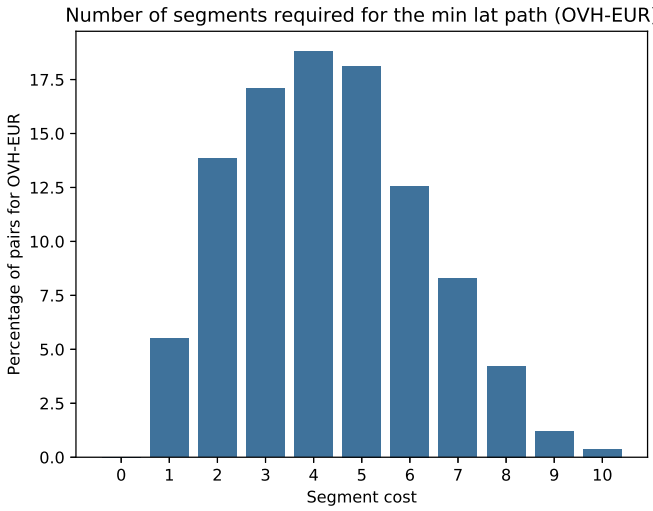


Figure 5.5: Number of segments required in the minimum latency paths for OVH-EUR.

shows the same results but for a single topology OVH-EUR.

For this topology the situation is quite different as we need more than 5 segments for about 26% of the pairs. The reason why we observe this behavior in the OVH-EUR topology is because it contains a lot of link bundles having parallel links. This creates a lot of ECMP making it hard for segment routing to pass by very specific paths.

This motivates the need for a minimum latency path finding algorithm that takes into account the number of segments from the start rather than simply computing a minimum latency path and hope that by chance it will not requite a long segment stack.

Minimum latency SR-Path problem

Another way of approaching this problem is to define a suitable sr-metric and use Algorithm 4 for computing a minimum latency sr-path. In this way we will have full control over the number of segments in the output. The price to pay with this is that the latency of the obtained sr-path will not be the global minimum but only the local minimum over all sr-paths in $\vec{\mathcal{P}}_{\mathtt{k}}$.

To be able to define the problem more formally we need to define what is the latency of a sr-path. This might seem obvious but in the presence of ECMP, a sr-path will correspond to a subnetwork of the original network rather than a simple path. In reality, as we mentioned previously, the behavior of the network may vary making it impossible to know which path amongst all paths defined by the sr-path will actually be used to forward traffic.

Let us illustrate this in a real example. Figure 5.6 shows the shortest path subnetwork between routers a and j in a real network. Both paths have the same IGP cost of 72. The top path has latency 25.36ms and the bottom one has 19.1ms. Whenever we have a sr-path of the form $\langle \ldots, \mathtt{a}, \mathtt{j}, \ldots \rangle$, we do not know

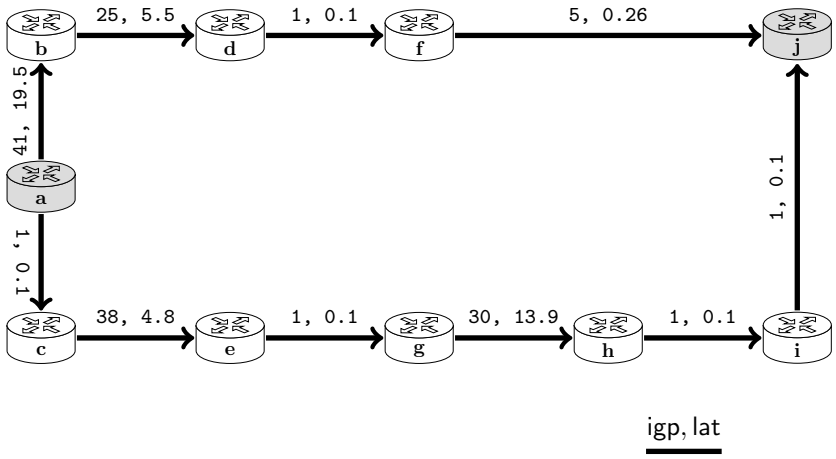


Figure 5.6: The shortest path DAG between routers a and j in a real network.

whether path $(\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{f}, \mathtt{j})$ or $(\mathtt{a}, \mathtt{c}, \mathtt{e}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j})$ will be used to forward traffic between a and j.

If we want to be able to provide guarantees about the latency, we need to consider the worst case scenario where, whenever there is more than one path, we assume the slowest one is used. This corresponds to defining the latency of a sr-path $\langle s_1, s_2 \rangle$ as the *longest path* (w.r.t latencies) of any paths between s_1 and s_2 on $\mathsf{SP}(s_1, s_2)$. This motivates the following definition.

Definition 5.2. *Let* s_1, s_2 *be two nodes of a network* G *and* e *an edge. The* maximum latency sr-metric *is defined as*

$$lat_M(s_1, s_2) = \textit{maximum latency of a path from } s_1 \textit{ to } s_2 \textit{ in } \mathsf{SP}(s_1, s_2)$$

and

$$lat_M(\mathsf{e}) = \mathsf{lat}(\mathsf{e}).$$

With this metric, we have $lat_M(\mathtt{a}, \mathtt{j}) = 25.38$. Note that on single edges this metric is defined as to match that edge's latency. It is important to notice that this metric can be computed efficiently. It is well known that the longest path problem is NP-hard in general [12]. However, in this case we are computing it on an acyclic network (the shortest path subnetwork) which can be done in linear time with a simple dynamic programming algorithm [12].

Another metric that is of interest is to consider that the latency of a sr-path is the average latency over all shortest paths. This corresponds in practice with a situation where the next hops are selected randomly in case of ECMP. It somehow reflects the behavior of selecting the next hops with an unknown hash function when several possible next hops exist.

Definition 5.3. *Let* s_1, s_2 *be two nodes of a network* G *and* e *an edge. The* average latency sr-metric *is defined as*

$$lat_\mu(s_1, s_2) = \textit{average latency among all paths from } s_1 \textit{ to } s_2 \textit{ in } \mathsf{SP}(s_1, s_2)$$

and

$$lat_\mu(\mathsf{e}) = \mathsf{lat}(\mathsf{e}).$$

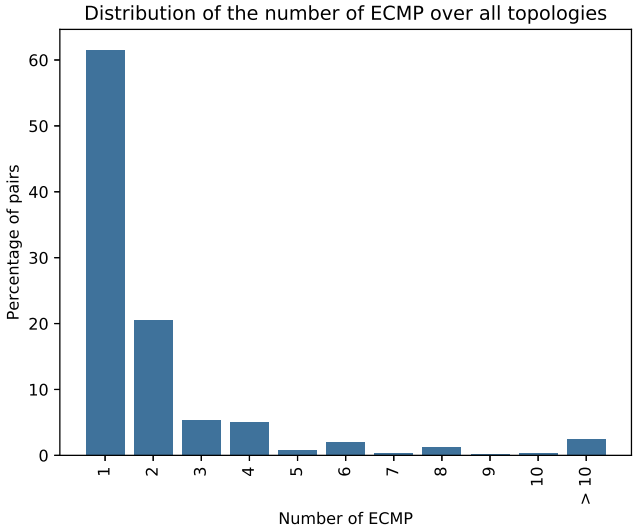


Figure 5.7: Evaluation of the average number of ECMP over all topologies.

This metric can also be computed efficiently. We can use dynamic programming to compute the number of paths and their total cost in linear time. The only complication is that this number grows exponentially which can easily be overcome using arbitrary precision integers.

Figure 5.7 shows the distribution of the number of equal cost shortest path between all pairs of routers over all topologies. We can see about 60% of the time there is a single shortest path between the pairs of nodes. The maximum number of paths found for any pair of nodes was 11150. This value might seem huge but it is actually quite small when we take into the fact the number of paths in a DAG can be as large as $2^{|V|-2}$ where $|V|$ is the number of routers in that topology. This means that when computing average measures, we can probably use normal integers rather than arbitrary precision integers.

5.2 Maximum weight sr-paths

In this section, we briefly discuss the problem of finding maximum weight sr-paths.

Problem 2 (Maximum weight sr-path)

Input: *A network* G*, a sr-metric* w*, an integer constant* $k \geqslant 1$ *and two distinct nodes* $s, t \in V(G)$.
Output: *A sr-path* $\vec{p} \in \vec{\mathcal{P}}_k(s,t)$ *such that* $w(\vec{p})$ *is maximal.*

This problem can be solved in the exact same way as the minimum weight sr-path problem (Problem 1) by replacing the ∞ by $-\infty$ on line 1 and replacing the $<$ by $>$ on lines 9 and 17 of Algorithm 4.

This result might seem unintuitive at first because, as mentioned at the end of Chapter 4, the longest path problem on graphs is NP-hard. Hence, one might think by setting k arbitrarily high and by using an appropriate sr-metric the two problems could become the same. There is however an important nuance. In this problem we are not preventing the sr-path from repeating segments while the longest path problem required simple paths. It is not hard to see that if we drop the simple path constraint then the longest path problem becomes trivial: if the input graph is acyclic then we compute the answer with dynamic programming otherwise the answer is ∞ since we can traverse any cycle an arbitrary amount of times getting longer and longer paths.

The next question one might then ask about this problem is, can this problem be interesting for some sr-metric w? The answer is yes as we will see in Chapter 7.

5.3 Maximum capacity sr-paths

When routing traffic over a network it can be the case that the network is highly loaded. This can cause some network links to become congested such as for instance, shortest path links or the links in the minimum latency path. Motivated by this, we study the problem of finding a sr-path between two nodes whose congestion in minimum. Having such an algorithm can be an interesting building block for dynamically allocating incoming demands while doing a best effort to prevent congestion. Of course, using such an algorithm locally for each incoming demand will lead to an overall sub-optimal solution. We will study the problem of considering all demands at once on Chapter 6.

We start with a very simple definition of demand.

Definition 5.4. *A* demand *in a network* G *is a triple* $(\mathsf{s}, \mathsf{t}, \nu)$ *where* $\mathsf{s}, \mathsf{t} \in \mathsf{V}(\mathsf{G})$ *and* $\nu \in \mathbb{N}^+$. *Given a demand* d *we write* $src(\mathsf{d}) = \mathsf{s}$, $dst(\mathsf{d}) = \mathsf{t}$ *and* $vol(\mathsf{d}) = \nu$.

Before formally defining the problem that we solve in this section, we need to understand how traffic on a sr-path affects link loads since sr-paths can actually correspond to several paths on the original network in case of ECMP. In this section we assume that the network is using a traffic split mechanism. Recall that this means that in the presence of ECMP, the routers evenly split the incoming traffic amongst those equal cost paths. Consider sending one unit of flow from node $\mathtt{a}$ to node $\mathtt{d}$ on the graph shown in Figure 5.8. There are two shortest paths between these nodes: $((\mathtt{a}, \mathtt{c}), (\mathtt{c}, \mathtt{d}))$ and $((\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{d}))$. Therefore, there will be a 50-50 split of the load over these two paths.

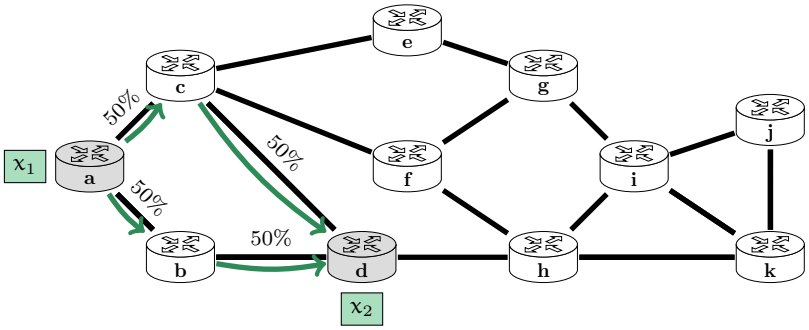


Figure 5.8: Traffic split example between ECMP.

We model this by computing, for each source-destination pair $s, t \in V(G)$, and every edge $e \in E(G)$, what is the fraction of the traffic passing by e when using shortest path routing from s to t. In the previous example, this fraction is 0.5 for edges $(a, b), (a, c), (c, d)$ and (b, d) and 0 on all other edges with $s = a$ and $t = d$. First, need to define the load on the nodes assuming that 1 unit of traffic is sent from s to t.

Definition 5.5. *Let* G *be a network and* $s, t \in V(G)$. *We define for every* $v \in V(G)$

$$\mathsf{load}^{st}(v) = \begin{cases} 1 & \text{if } v = s \\ \displaystyle\sum_{u \in \delta^-(SP(s,t),v)} \frac{\mathsf{load}(u)}{|\delta^+(SP(s,t),u)|} & \text{otherwise} \end{cases}$$

Note that for each $s, t \in V(G)$, $SP(s, t)$ is an acyclic graph so $\mathsf{load}^{st}(v)$ is well defined (it is not defined in terms of itself). The value of $\mathsf{load}^{st}(v)$ corresponds to the amount of traffic that would pass by the router corresponding to node v when routing one unit of traffic from s to t, assuming that in case of ECMP the traffic is split equally. For example, consider the network shown in Figure 5.9 and assume that $s = c$ and $t = i$ (shown in green). Then $SP(s, t) = SP(c, i)$ corresponds to the green edges and the value of $\mathsf{load}^{st}(v)$ is shown in the gray boxes next to each node $v \in V(G)$. We see that, for instance, $\mathsf{load}^{st}(h) = 1/2$. This means that under equal split, half of the traffic load sent with shortest path routing from c to i will pass by h. This corresponds to the definition since

$$\begin{aligned} \mathsf{load}^{ci}(h) &= \sum_{u \in \delta^-(SP(c,i),h)} \frac{\mathsf{load}(u)}{|\delta^+(SP(c,i),u)|} \\ &= \frac{1}{2} \cdot \mathsf{load}^{ci}(f) + \frac{1}{1} \cdot \mathsf{load}^{ci}(d) \\ &= \frac{1}{2} \cdot \frac{1}{3} + \frac{1}{3} = \frac{1}{6} + \frac{1}{3} = \frac{1}{2} \end{aligned}$$

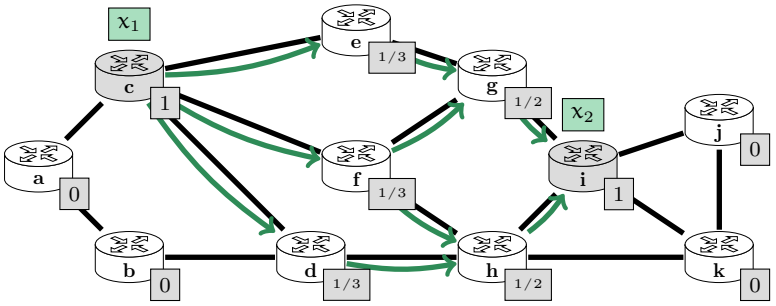


Figure 5.9: Node loads with respect to $s = c$ and $t = i$.

Using the node loads we can now define the edge loads which, as described above, give us the fraction of traffic traversed by a link in the network. These values are necessary to express the link capacity constraints on TE models with segment routing. This is what we study in the next chapter.

Definition 5.6. *Let* G *be a network and* $s, t \in V(G)$*. We define, for every* $e = (u, v) \in E(G)$

$$\tau(s, t, e) = \frac{\mathsf{load}^{st}(u)}{|\delta^+(\mathsf{SP}(s, t), u)|}.$$

In order to characterize the proportion of traffic traversed by a link with respect to a sr-path $\vec{p} = \langle x_1, \ldots, x_n \rangle$ *we define*

$$\tau(\vec{p}, e) = \sum_{i=2}^{n} \tau(x_{i-1}^2, x_i^1, e) + \sum_{i:x_i=e} 1.$$

The first part of the expression of $\tau(\vec{p}, e)$ evaluates the ratio on e for the shortest path subgraphs between consecutive segments, while the second part counts how many times e is used as an adjacency segment on the sr-path $\vec{p}$. This summation is valid because when an adjacency segment occurs, the full unit demand is routed through the edge without split. Note that $\tau(\vec{p}, e)$ may be larger than 1 since a same edge can be used several times in case of cycles in $\mathsf{forw}(\vec{p})$. Figure 5.10 shows the edge ratios $\tau(\langle a, c, i, j \rangle, e)$ for every $e \in E(G)$. The green arrows correspond to the forwarding graph of $\langle a, c, i, j \rangle$.

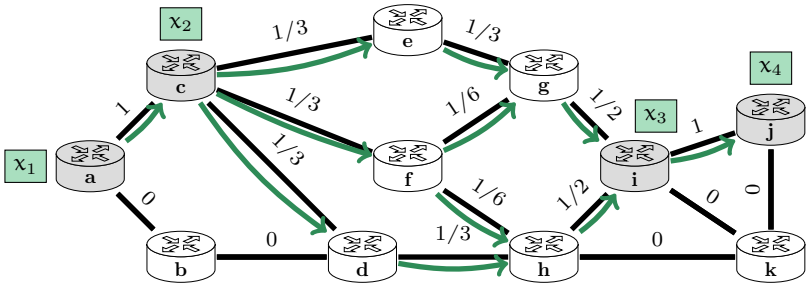


Figure 5.10: Split ratio for sr-path $\langle a, c, i, j \rangle$.

Definition 5.7. *Let* G *be a network and* $\vec{\mathsf{p}}$ *a sr-path on* G. *We say that an edge* $\mathsf{e} \in \mathsf{E}(\vec{\mathsf{p}})$ *is* critical *if the fraction* $\frac{\mathsf{cap}(e)}{\tau(\vec{\mathsf{p}},\mathsf{e})}$ *has minimum value amongst all edges* $\mathsf{e} \in \mathsf{E}(\vec{\mathsf{p}})$. *We define the capacity of* $\vec{\mathsf{p}}$ *as* $\mathsf{cap}(\vec{\mathsf{p}}) = \min_{\mathsf{e}\in\mathsf{E}(\vec{\mathsf{p}})} \frac{\mathsf{cap}(e)}{\tau(\vec{\mathsf{p}},\mathsf{e})}$.

To understand this definition, imagine that we route a demand $\mathtt{d}$ over a sr-path $\vec{\mathsf{p}}$. The bandwidth that this will consume of each edge e is given by $\tau(\vec{\mathsf{p}}, \mathsf{e}) \cdot \mathit{vol}(\mathtt{d})$. Hence, we will exceed the capacity of edge e if and only if

$$\tau(\vec{\mathsf{p}}, \mathsf{e}) \cdot \mathit{vol}(\mathtt{d}) > \mathsf{cap}(\mathsf{e}) \Leftrightarrow \mathit{vol}(\mathtt{d}) > \frac{\mathsf{cap}(e)}{\tau(\vec{\mathsf{p}}, e)}.$$

Hence $\mathsf{cap}(\vec{\mathsf{p}})$ describes the maximum volume that can be routed over $\vec{\mathsf{p}}$ without exceeding the capacity of any edge in $\mathsf{E}(\vec{\mathsf{p}})$.

Lemma 5.3. *Let* G *be a network and* $\vec{\mathsf{p}} = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle$ *a sr-path on* G. *The following holds:*

i) If $\mathtt{x}_n \in \mathsf{V}(\mathsf{G})$ *then* $\mathsf{cap}(\vec{\mathsf{p}}) = \min\left(\mathsf{cap}\langle \mathtt{x}_1, \ldots, \mathtt{x}_{n-1}\rangle, \mathsf{cap}\langle \mathtt{x}^2_{n-1}, \mathtt{x}_n\rangle\right)$

ii) If $\mathtt{x}_n \in \mathsf{E}(\mathsf{G})$ *then* $\mathsf{cap}(\vec{\mathsf{p}}) = \min\left(\mathsf{cap}\langle \mathtt{x}_1, \ldots, \mathtt{x}_{n-1}\rangle, \mathsf{cap}\langle \mathtt{x}^2_{n-1}, \mathtt{x}^1_n\rangle, \mathsf{cap}(\mathtt{x}_n)\right)$

Proof. Immediate from the definition of $\mathsf{cap}(\vec{\mathsf{p}})$ since in the first case we have

$$\mathsf{E}(\vec{\mathsf{p}}) = \mathsf{E}(\langle \mathtt{x}_1, \ldots, \mathtt{x}_{n-1}\rangle) \cup \mathsf{E}(\langle \mathtt{x}^2_{n-1}, \mathtt{x}_n\rangle)$$

and in the second one

$$\mathsf{E}(\vec{\mathsf{p}}) = \mathsf{E}(\langle \mathtt{x}_1, \ldots, \mathtt{x}_{n-1}\rangle) \cup \mathsf{E}(\langle \mathtt{x}^2_{n-1}, \mathtt{x}^1_n\rangle) \cup \mathtt{x}_n.$$

□

With this, we now can define the problem of finding the best sr-path, in terms of capacity, to route a given demand $\mathtt{d}$.

Problem 3 (Maximum capacity sr-path)

Input: *A network* G *and a demand* $\mathtt{d}$ *on* G.
Output: *A sr-path from* $\mathit{src}(\mathtt{d})$ *to* $\mathit{dst}(\mathtt{d})$ *such that* $\mathsf{cap}(\vec{\mathsf{p}})$ *is maximum.*

To solve Problem 3 we are going to proceed in a manner very similar to what we did to compute minimum weights sr-paths. We define a DP state space as follows.

$$\begin{aligned} \mathit{sol}(\mathtt{i}, \nu) = &\text{ the maximum capacity of a sr-path from} \\ &\mathtt{s} \text{ to } \nu \text{ with segment cost at most } \mathtt{i}. \end{aligned}$$

With a case analysis similar to the one we performed above we can obtain a recurrence expressing $\mathit{sol}(\mathtt{i}, \nu)$ in terms of $\mathit{sol}(\mathtt{i}-1, *)$ and $\mathit{sol}(\mathtt{i}-2, *)$ as shown in the next theorem.

Theorem 5.4. *For* $i \geqslant 1$ *and* $x \in V(G)$ *it holds that*

$$sol(i, v) = \max \begin{cases} sol(i-1, v) \\ \max\limits_{u \in V(G)} \min\left(sol(i-1, u), \mathsf{cap}\langle u, v\rangle\right) \\ \max\limits_{r \in V(G), e \in \delta^-(v)} \min\left(sol(i-2, r), \mathsf{cap}\langle r, u\rangle, \mathsf{cap}(e)\right) \end{cases}$$

where the third value is only defined for $i \geqslant 2$ *(set to* $-\infty$*) otherwise.*

Proof. Suppose that $\vec{p} = \langle x_1, \dots x_n\rangle$ is an optimal solution to the sub-problem corresponding to $sol(i, v)$. Thus $\vec{p}$ is a sr-path with $\mathsf{sr\text{-}cost}(\vec{p}) \leqslant i$ from s to u with maximum capacity. If $\mathsf{sr\text{-}cost}(\vec{p}) < i$ then $\vec{p}$ is also the solution of the sub-problem corresponding to $sol(i-1, v)$ so $sol(i, v) = sol(i-1, v)$. Otherwise $\mathsf{sr\text{-}cost}(\vec{p}) = i$ and we consider two cases, depending on whether x_n is a node segment or an adjacency segment.

Case 1: $x_n \in V(G)$. In this case, by Lemma 5.3 we have that

$$\mathsf{cap}(\vec{p}) = \min\left(\mathsf{cap}\langle x_1, \dots, x_{n-1}\rangle, \mathsf{cap}\langle x_{n-1}^2, x_n\rangle\right)$$

Since $\vec{p}$ is optimal, $\langle x_1, \dots, x_{n-1}\rangle$ has segment cost $i-1$ and $x_n = v$ it follows that

$$\min\left(\mathsf{cap}\langle x_1, \dots, x_{n-1}\rangle, \mathsf{cap}\langle x_{n-1}^2, x_n\rangle\right) = \max_{u \in V(G)} \min\left(sol(i-1, u), \mathsf{cap}\langle u, v\rangle\right).$$

Case 2: $x_n \in E(G)$. Then, by Lemma 5.3,

$$\mathsf{cap}(\vec{p}) = \min\left(\mathsf{cap}\langle x_1, \dots, x_{n-1}\rangle, \mathsf{cap}\langle x_{n-1}^2, x_n^1\rangle, \mathsf{cap}(x_n)\right).$$

As in the first case, since $\vec{p}$ is optimal, $\langle x_1, \dots, x_{n-1}\rangle$ has segment cost $i-2$ and $x_n \in \delta^-(v)$ it follows that

$$\mathsf{cap}(\vec{p}) = \max_{r \in V(G), e \in \delta^-(v)} \min\left(sol(i-2, r), \mathsf{cap}\langle r, u\rangle, \mathsf{cap}(e)\right).$$

Since these cover all possibilities we conclude the proof. □

We can compute this recurrence as shown in Algorithm 5. Since the recurrence is based on the values of $\mathsf{cap}\langle u, v\rangle$ where $u, v \in V(G)$ we start by computing those using the definition. The remainder of the algorithm is very similar to Algorithm 4. The only differences are the following. The initialization of `sol` on lines 5 and 6 uses values 0 for impossible solution and ∞ as the capacity of the empty path. The only other changes are the conditions of lines 13 and 21 that are set to match the recurrence provided by Theorem 5.4 and the respective assignments to `sol` that follow these conditions.

This is an important tool when designing online demand allocation. We can use it to accept or refuse demand by keeping track of the link congestions and for each incoming demand d computing the maximum capacity sr-path $\vec{p}$ between $src(d)$ and $dst(d)$. Then if $\mathsf{cap}(\vec{p}) \geqslant vol(d)$ we accept the demand and route it over $\vec{p}$. Otherwise we reject it. We did not explore the problem of online demand routing in this thesis but we believe that this algorithm together with our column generation algorithm for the offline problem that we propose in the next chapter are a good starting point for someone wanting to tackle this problem with segment routing.

Algorithm 5 maxCapSrPath $(g, \tau, \mathsf{orig}, \mathsf{dest}, \mathit{maxSeg})$

```
1: [pre-compute cap⟨u, v⟩ for all u, v and store them as c(u, v)]
2: for u ∈ V(G) do
3:    for v ∈ V(G) do
4:       c(u, v) ← min_{e∈E(SP(u,v))} cap(e) / τ(⟨u, v⟩, e)
5: sol ← matrix(maxSeg + 1, g.V(), 0)
6: sol(0, orig) = ∞
7: parent ← matrix(maxSeg + 1, g.V(), null)
8: for i = 1, ..., maxSeg do
9:    for cur ∈ g.V() do
10:      sol(i, cur) ← sol(i − 1, cur)
11:      parent(i, cur) ← parent(i − 1, cur)
12:      for prev ∈ g.V() \ {cur} do
13:         if min(sol(i − 1, prev), c(prev, cur)) > sol(i, cur) then
14:            [found a better solution by reaching cur using node segment on cur]
15:            sol(i, cur) = min(sol(i − 1, prev), c(prev, cur))
16:            parent(i, cur) ← prev
17:      [the third case is only defined for i ⩾ 2]
18:      if i ⩾ 2 then
19:         for e = (orig, dest) ∈ g.inEdges(cur) do
20:            for prev ∈ g.V() \ {cur} do
21:               if min(sol(i − 2, prev), c(prev, orig), cap(e)) > sol(i, cur) then
22:                  [found better solution by reaching cur using an adjacency on e]
23:                  sol(i, cur) = min(sol(i − 2, prev), c(prev, orig), cap(e))
24:                  parent(i, cur) ← e
25: opt ← sol(maxSeg, dest)
26: k ← maxSeg
27: while k − 1 ⩾ 0 and sol(k − 1, dest) = opt do
28:    k ← k − 1
29: return SrPath(parent, orig, dest, k, opt)
```

We close this chapter by showing, perhaps surprisingly, that the maximum capacity sr-path is not necessarily acyclic. This is one example where we *cannot* use Theorem 4.6 to ignore sr-paths. We will see in Chapter 8 an example where we can leverage this theorem to ignore all cyclic sr-paths in our formulation leading to a faster algorithm.

To see that the maximum capacity sr-path can be cyclic, consider the network shown in Figure 5.11 where we want to route one unit demand from node a to node c. The intuitive solution would be to simply use sr-path $\langle \mathsf{a}, \mathsf{c} \rangle$. However this will lead to using 100% of the capacity of link (a, c). If however we first go to node d and only then back to c, the maximum link utilisation will be link (d, c) which will be used at 50%.

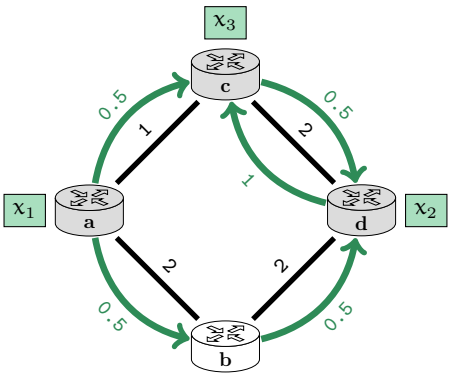


Figure 5.11: Example where the maximum capacity sr-path $\vec{p} = \langle \mathtt{a}, \mathtt{d}, \mathtt{c} \rangle$ from $\mathtt{a}$ to $\mathtt{c}$ is cyclic. The green edges represent $\mathsf{forw}(\vec{p})$ and the values on top of them the amount of traffic demand passing on each given edge.

5.4 Conclusion

In this chapter we proposed a general algorithm for computing minimum weight sr-paths. We showed that we can leverage this algorithm in order to achieve lower latency than with shortest path routing in a segment routed network.

We will see in later chapters that computing minimum weight sr-paths is a common subproblem that occurs designing optimization algorithms for segment routing. By using similar ideas we also proposed an algorithm for computing maximum capacity sr-paths. These two examples show that DP programming is a good fit for solving these kind of problems. We believe that other interesting similar problems will arise in the future and that following this approach will probably be a good starting point for having an efficient algorithm for solving them.

Chapter 6

Traffic engineering with SR

Introduction

In this chapter we propose a column generation algorithm for solving the traffic engineering (TE) problem on a network using segment routing. The traffic engineering problem consists of routing a set of demands over the network whilst minimizing congestion. Routing traffic between a source and a destination along shortest paths allows for no flexibility regarding the way this traffic is send which may lead to a suboptimal usage of the network infrastructure. Consider the four router network show in Figure 6.1. Suppose that we want to route three demands on this network, all from `a` to `b`. Assuming unit IGP weights, all those demands will traverse the link $(\texttt{a}, \texttt{d})$. In contrast, we could route one of them via `b` and the other via `c` as shown on the right. This second solution leads to a much more efficient utilisation of the network. It spreads the traffic thus leading to less congestion. Assuming equal demand volumes, the solution on the right improves the congestion of the link $(\texttt{a}, \texttt{b})$ by 66%. This is a very small example and even here we can already see the limitations of shortest path routing for TE.

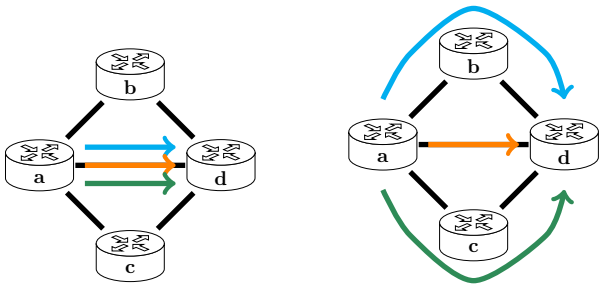


Figure 6.1: Small example where shortest path routing is not ideal for TE.

Attempts at solving the TE problem can be categorized into two main categories. On one hand we could improve the way a set of demands is routed by computing new IGP weights such that the resulting shortest paths between the endpoints of each demand minimize the network congestion [25]. This approach

is limited by the fact that these weights depend of the demands. Having to configure new IGP weights for each set of demands is, to say the least, impractical. Of course, one can try to find weights that are good on average and always use those for each demand set. However, this will clearly lead to suboptimal routing in terms of network capacity.

Another possibility is to use a forwarding mechanism that allows to route traffic over arbitrary paths. This is the approach that we are going to study in this chapter by using SR as a forwarding mechanism.

6.1 Traffic engineering formalization

In this section we provide a formal definition of the TE problem considered in this thesis. This formalization is not yet sr-centric. We define the segment routing variant in a later section.

The usual definition of the problem assumes that a demand matrix is given as input rather than a set of demands. The entry i, j of the matrix contains a positive number representing the volume of the traffic that needs to be routed between routers i and j. We prefer to use a set of demands at it allows for more granularity. With a demand set, we can have several demands between the same pair of routers that are routed in different ways. If for some reason such granularity is not implementable in practice, one can always group the demands between the same source-destination pairs into a single demand. This makes the model with a set of demands more general than the matrix one.

The following definition will be useful to express the load of a link when routing a demand on a given path p.

Definition 6.1. *Let G be a network and $p = (e_1, \dots, e_l)$ a path on G. Given $e \in E(G)$, we denote the number of indexes such that $e_i = e$ by $\mathcal{C}(p, e)$.*

We can now formally define the TE problem. We start with a general definition that is not SR centric where we assume that any path in the network can be used to route any given demand.

Problem 1 (TE min-factor problem)

Input: *A network G and $\mathcal{D}$ a set of n demands $\{d_1, \dots, d_n\}$ on G.*
Output: *The minimum factor $\lambda \geqslant 0$ and set of paths on G, $p_1, \dots, p_n$ such that p_i is a path from s_i to t_i and for each link $e \in E(G)$ it holds*

$$\sum_{i=1}^{n} \mathcal{C}(p_i, e) \cdot vol(d_i) \leqslant \lambda \cdot cap(e)$$

This is far from the only variant of the TE problem. In this version of the problem we aim at minimizing the maximum congestion of any link. The value of

$$\sum_{i=1}^{n} \mathcal{C}(p_i, e) \cdot vol(d_i)$$

gives the total amount of traffic that will be routed through edge e if one uses paths $p_1, \dots, p_n$ to route demands $d_1, \dots, d_n$, respectively. Note that in the

definition of the problem, we did not require the paths to be simple. Therefore we need to count how often an edge is used by the path. This might seem unnecessary at first sight since it is easy to see that there is always a solution where each path p_i is simple. However, we will see that with segment routing this is not true and so, for consistency, we define the problem directly using these factors.

A solution with $\lambda = 1$ is a solution where the capacity of every edge is not exceeded. Clearly the lower λ is, the better as this means that no edge is using more that a ratio λ of its capacity. One could argue that this objective function is not the most interesting in practice because it looks only at the worst case link utilization and fails to provide a global view of the network. Imagine for instance a solution where each link is used at 50% but one link is used at, say, 90%. This version of the problem will prefer, for example, a solution where every link is used at 80% since the maximum utilisation is lower. A lot of variants have been considered in the literature [5]. The reason why we chose this objective function is because most segment routing solutions so far focus on this variant of the problem thus making it easier for us to compare our results with theirs.

A common relaxation of the problem is to consider that each demand can be served by a set of paths rather than a single path. It turns out that this variant of the problem is much easier to solve. We will see that we can solve it in polynomial time whereas Problem 1 is NP-hard [18].

Problem 2 (TE-multipath min-factor problem)

Input: *A network* G *and* $\mathcal{D}$ *a set of* n *demands* $\{d_1, \ldots, d_n\}$ *on* G.
Output: *The minimum factor* $\lambda \geqslant 0$ *and* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *such that each* $\mathcal{P}_i$ *is a non-empty set of pairs* (p, f) *where* p *is a path from* s_i *to* t_i *and* $f \in [0, 1]$ *and for each link* $e \in E(G)$ *it holds*

$$\sum_{i=1}^{n} \sum_{(p,f)\in\mathcal{P}_i} \mathcal{C}(p, e) \cdot f \cdot vol(d_i) \leqslant \lambda \cdot cap(e)$$

This relaxation is often used to provide a lower bound to the optimal value λ^* of Problem 1. In the next section we provide a brief overview about linear programming (LP) which will be the central tool used to solve the problems presented on the remainder of this thesis. We will also show how solve Problem 2 in polynomial time by providing a LP formulation.

6.2 A brief introduction to LP and MIP

In this section we provide a very brief introduction to linear programming and mixed integer programming. The goal is simply to introduce the necessary vocabulary in order to simplify the wording of the remaining of this chapter.

A linear programming is a central tool in optimization that allows one to solve mathematical models whose objective is a linear function and whose constraints are representable by linear inequalities. A linear function is a function $f : \mathbb{R}^n \to \mathbb{R}$ such that

$$f(x_1, \ldots, x_n) = c_1 x_1 + \cdots + c_n x_n$$

for some constants $c_1, \ldots, c_n \in \mathbb{R}$. A linear inequality is an inequality of the form

$$a_1 x_1 + \ldots + a_n x_n \geqslant b$$

for $a_1, \ldots, a_n, b \in \mathbb{R}$. In general, we have not one but m constraints so we are given an $m \times n$ matrix of coefficients A and m values $b_1, \ldots, b_m$. Each row of coefficients represents one constant. The general form of a linear program is the following:

$$\begin{array}{lccccccccc}
\textbf{min} & c_1 x_1 & + & c_2 x_2 & + & \ldots & + & c_n x_n & & \\
\textbf{s.t} & a_{11} x_1 & + & a_{12} x_2 & + & \ldots & + & a_{1n} x_n & \geqslant & b_1 \\
 & a_{21} x_1 & + & a_{22} x_2 & + & \ldots & + & a_{2n} x_n & \geqslant & b_2 \\
 & \vdots & & & & & & & & \vdots \\
 & a_{m1} x_1 & + & a_{m2} x_2 & + & \ldots & + & a_{mn} x_n & \geqslant & b_m \\
 & x_i \in \mathbb{R} & & & & & & & &
\end{array}$$

Numerous problems can be modeled as linear programs. This, combined with the fact that linear programs can be solved efficiently, makes LP a very interesting problem solving tool [3, 13].

Every linear program is associated with another linear program called its *dual* LP. The original problem is called the *primal*. The dual of a LP has one constraint for each variable of the primal and one variable for each constraint of the primal. The objective function is reversed so that if the primal is a minimization problem then the dual is a maximization problem (and vice-versa). The main result about LP duality is the *strong duality theorem* [41] which states that the primal has an optimal solution if and only if the dual also has one and both of them have the same value. The dual of a LP can be obtained by following a systematic procedure. In the case of the LP formulated above, the dual is given by:

$$\begin{array}{lccccccccc}
\textbf{max} & b_1 y_1 & + & b_2 y_2 & + & \ldots & + & b_m y_m & & \\
\textbf{s.t} & a_{11} y_1 & + & a_{21} y_2 & + & \ldots & + & a_{m1} y_m & \leqslant & c_1 \\
 & a_{12} y_1 & + & a_{22} y_2 & + & \ldots & + & a_{m2} y_m & \leqslant & c_2 \\
 & \vdots & & & & & & & & \vdots \\
 & a_{1n} x_1 & + & a_{2n} y_2 & + & \ldots & + & a_{mn} y_m & \leqslant & c_n \\
 & y_i \in \mathbb{R} & & & & & & & &
\end{array}$$

We are going to exploit duality when developing our CG solution. In our algorithm we will denote a function LP-solve that, given an LP, outputs its optimal solution x, the optimal solution of the dual y and the optimal objective value.

In the above formulation we assumed that the domain of the variables was $\mathbb{R}$. Linear programs remain easy to solve as long as the variable ranges are continuous ranges. A lot of very important problems can be modeled with a linear objective function and linear constraints but require integral variable domains. We refer to these problems as integer programs (IP) or mixed integer program (MIP) if some of the variables are continuous and some are integral. The duality theory mentioned above does not extend to integer programming. The general integer programming problem is NP-hard. This makes it much

more challenging to find optimal solutions to MIP's in general. In this chapter's introduction we mentioned that Problem 1 was hard to solve but Problem 2 was easy. We will see that both problems can be formulated with nearly the same model. The only difference between them is that the first requires integer variables whereas the second allows for continuous variables.

A lot of research has been put towards finding efficient algorithms for solving MIP's [55]. For the algorithms developed here, we denote a function MIP-solve that given a MIP (or IP), outputs the optimal solution x and the optimal objective value.

To give a concrete example of a linear program, we model Problem 2. A classic way to formulate Problem 2 as a linear program is to express it as a Multi-commodity flow (MCF) problem [26]. We define variables x_{ed} for each edge $e \in E(G)$ and demand $d \in \mathcal{D}$ such that the value of x_{ed} represents the faction of $vol(d)$ that traverses edge e. The model LP is the following:

MCF-LP$(G, \mathcal{D})$

$$\textbf{min} \quad \lambda$$

$$\textbf{s.t.} \quad \sum_{d \in \mathcal{D}} x_{ed} \cdot vol(d) \leqslant \lambda \cdot cap(e) \quad \forall e \in E(G) \tag{1}$$

$$\sum_{e \in \delta^-(v)} x_{ed} - \sum_{e \in \delta^+(v)} x_{ed} = 0 \quad \forall v \in V(G) \setminus \{s_i, t_i\}, \ \forall d \in \mathcal{D} \tag{2}$$

$$\sum_{e \in \delta^-(s)} x_{ed} - \sum_{e \in \delta^+(s)} x_{ed} = -1 \quad \forall i \in \{1, \dots, r\}, \ \forall d = (s, t, v) \in \mathcal{D} \tag{3}$$

$$\sum_{e \in \delta^-(t)} x_{ed} - \sum_{e \in \delta^+(t)} x_{ed} = 1 \quad \forall i \in \{1, \dots, r\}, \ \forall d = (s, t, v) \in \mathcal{D} \tag{4}$$

$$x_{ed} \in [0, 1]$$

$$\lambda \geqslant 0$$

Constraints (1) ensure that the total demand volume traversing any edge is at most a fraction λ of its capacity. Constraints (2), (3) and (4) are known as *flow constraints* and ensure that each demand d is routed from $src(d)$ to $dst(d)$. More precisely, the group of constraints (3) makes sure that the whole demand exits $src(d)$ and the group of constraints (4) ensure that the whole demand reaches $dst(d)$. The constraints (2) ensure that no demand traffic is lost by requiring that any fraction of the demand that enters an intermediate router also exits it.

Given a solution matrix x to MCF-LP we can easily construct a set of paths for each demand $d = (s, t, v) \in \mathcal{D}$. To do so, we perform a sequence of breadth-first searches (BFS) from s following only edges e such that $x_{ed} > 0$. Each time we reach t, we trace back the path p and reduce the value of x_{ed} by the $\min_{e \in E(p)} x_{ed}$. We continue to do this until one of the searches fails to reach t.

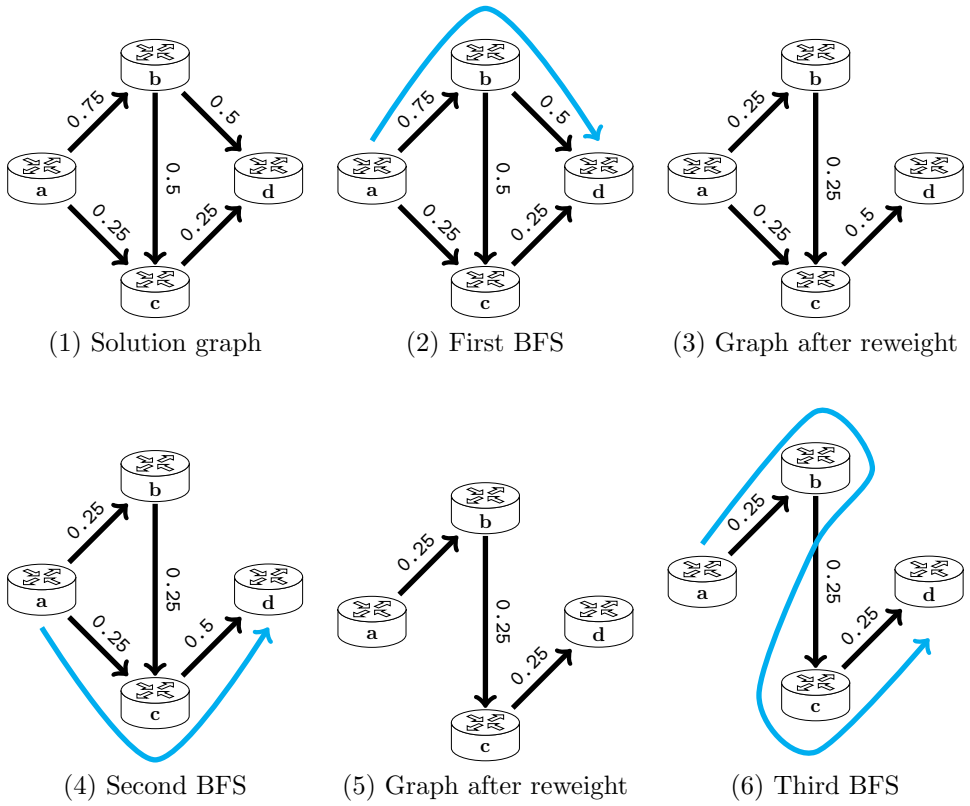


Figure 6.2: Converting the flow to a path set.

At this point we finished computing a set of paths for demand $\mathtt{d}$. Repeating the process for each demand will yield a set of paths over which we can route the demands without exceeding the capacity of any edge by a factor greater than λ.

Figure 6.2 illustrates this process on a small example. The graph represents the values of $x_{\mathtt{ed}}$ for a specific demand $\mathtt{d}$. Each edge if labeled with the value of $x_{\mathtt{ed}}$ (for clarity, we ommit edges with value 0 since they are ignored anyway by the DFS). Then the blue arrows show possible BFS paths. After each path is found, the value of the edges is reduced until no more path exists. In this case we get three paths: $((\mathtt{a},\mathtt{b}),(\mathtt{b},\mathtt{d}))$, $((\mathtt{a},\mathtt{c}),(\mathtt{c},\mathtt{d}))$ and $((\mathtt{a},\mathtt{b}),(\mathtt{b},\mathtt{c}),(\mathtt{c},\mathtt{d}))$.

Using this procedure, we can easily transform a solution of MCF-LP in a solution $\mathcal{P}_1,\ldots,\mathcal{P}_r$ of Problem 2. Algorithm 6 provides a formal description of this process.

Proposition 6.1. *Algorithm 6 runs in polynomial time and computes an optimal solution of Problem 2.*

Proof. Optimality comes from the fact that MCF-LP correctly models Problem 2 [18].

We know that linear problems can be solved in polynomial time [3, 13]. Therefore x can be computed in polynomial time on line 1. It remains to show that our path building process takes polynomial time. Let $\mathtt{d} \in \mathcal{D}$. Building the

Algorithm 6 TE-multipath $(G, \mathcal{D})$

```
1: x, y, λ ← LP-SOLVE(MCF-LP(G, 𝒟))
2: 𝒫 ← ∅
3: for d ∈ 𝒟 do
4:    G_d ← (V, {e ∈ E(G) | x_ed > 0})
5:    p ← BFS(G_d, src(d), dst(d))
6:    while p ≠ ⊥ do
7:       Δ ← min_{e∈E(p)} x_ed
8:       𝒫 ← 𝒫 ∪ {(p, Δ)}
9:       for e ∈ E(p) do
10:         x_ed ← x_ed − Δ
11:      G_d ← (V, {e ∈ E(G) | x_ed > 0})
12:      p ← BFS(G_d, src(d), dst(d))
13: return λ, 𝒫
```

graph G_d on line 4 takes $O(|E(G)| \cdot |\mathcal{D}|)$ and each BFS call takes $O(|G|)$. In the body of the while loop, that is, lines 7 to 12, the most costly line is line 12 which takes $O(|V(G)| \cdot |E(G)|)$. Therefore we only need to prove that the number of iterations of the while loop is polynomial. Since $\Delta = \min_{e \in E(p)} x_{ed}$, at each iteration, at least one edge is removed from G_d at line 11. This means that the number of iterations of the while loop is at most $|E(G)|$. □

By requiring integral variables on model MCF-LP$(G, \mathcal{D})$, that is, by replacing $x_{ed} \in [0, 1]$ by $x_{ed} \in \{0, 1\}$, we obtain a model for solving Problem 1. This illustrates that just by requiring integer variables we can completely change the difficulty of the problem since it takes us from a polynomial solvable problem to a NP-hard one in this case.

6.3 Traffic engineering with segment routing

In this section we discuss the variants of Problems 1 and 2 when instead of using arbitrary paths for routing, we use sr-paths. The SR version of the TE problem is obtained by replacing paths by sr-paths, $\mathcal{C}(p_i, e)$ by $\tau(\vec{p}_i, e)$ and adding a constraint on the segment cost of the sr-paths found. Recall that $\tau(\vec{p}_i, e)$ was defined on Chapter 5 and represents the proportion of the traffic that traverses edge e when routing over sr-path $\vec{p}$.

This last constraint is important to ensure that the paths can be supported by the routers in the network. This leads to the definition of the following two problems which, respectively, correspond to Problems 1 and 2.

Problem 3 (Segment routing traffic engineering)

Input: *A network* G*, a set of demands* $\mathcal{D} = \{d_1, \ldots, d_n\}$ *on* G *and* $k \in \mathbb{N}$.
Output: *The minimum factor* $\lambda \geqslant 0$ *and set of sr-paths on* G*,* $\vec{p}_1, \ldots, \vec{p}_n$ *such that* $\vec{p}_i$ *is a sr-path from* s_i *to* t_i *with* ***sr-cost***$(\vec{p}_i) \leqslant k$ *and for each link* $e \in E(G)$ *it holds*

$$\sum_{i=1}^{n} \tau(\vec{p}_i, e) \cdot vol(d_i) \leqslant \lambda \cdot cap(e).$$

Problem 4 (Segment routing traffic engineering)

Input: *A network* G*, a set of demands* $\mathcal{D} = \{\mathsf{d}_1, \ldots, \mathsf{d}_n\}$ *on* G *and* $\mathsf{k} \in \mathbb{N}$*.*

Output: *The minimum factor* $\lambda \geqslant 0$ *and set of sr-paths on* G*,* $\vec{\mathcal{P}}_1, \ldots, \vec{\mathcal{P}}_n$ *such that* $\vec{\mathcal{P}}_i$ *is a non-empty set of pairs* $(\vec{p}, f)$ *where* $\vec{p}$ *sr-path from* s_i *to* t_i *with* $\mathit{sr\text{-}cost}(\vec{p}_i) \leqslant k$*,* $f \in [0, 1]$ *and for each link* $e \in E(G)$ *it holds*

$$\sum_{i=1}^{n} \sum_{(\vec{p},f)\in\vec{\mathcal{P}}_i} f \cdot \tau(\vec{p}, e) \cdot \mathit{vol}(d_i) \leqslant \lambda \cdot \mathsf{cap}(e).$$

The minimum segmentation algorithm provides a way to translate solutions of Problems 1 and 2 into solutions of Problems 3 and 4, respectively. The advantage of this approach is that makes it possible to leverage existing algorithms for solving these widely studied graph problems. The drawback of course is that since these algorithms are oblivious to segment routing, there is no way of knowing whether or not the output paths will require too many segments for routers to be able to support them.

We first evaluate the segment cost mentioned above. We already did this in the previous chapter when we considered the problem of computing minimum latency sr-paths. This is a way to evaluate how necessary it is to develop dedicated algorithms and how often we can get away by simply segmenting graph centric solutions using the minimum segmentation algorithm proposed in Chapter 4. Perhaps in the future routers will be able to support a high amount of segments and whenever this is the case, it will become fruitless to develop dedicated algorithms.

6.3.1 Existing MIP models and algorithms for SRTE

We now have all the tools needed to describe existing models for the traffic engineering problem with segment routing. The first approach that was developed by Bathia et al and considers only sr-paths of the form $\langle s, x, t\rangle$ to route a demand from s to t [11]. That is, it restricts the set of admissible sr-paths to sr-paths with a single detours towards a given router x on the network. The main advantage of doing this is that we obtain a very efficient model since for each demand there is a single decision to make: which intermediate router to use.

Let $P_1 = \{\langle \mathit{src}(d), x, \mathit{dst}(d)\rangle \mid x \in V(G)\}$. We define binary variables x_{dp} such that $x_{dp} = 1$ if and only if demand d is routed over sr-path p. Their model can be formulated as follows.

SRTE-1DET$(G, \mathcal{D})$

$$\textbf{min} \qquad \lambda$$

$$\begin{aligned}
\textbf{s.t.} \qquad \sum_{p \in P_1} x_{dp} &= 1 && \forall d \in \mathcal{D} \\
\sum_{\vec{p} \in P_1} \sum_{d \in \mathcal{D}} \tau(\vec{p}, e) \cdot vol(d) \cdot x_{dp} &\leqslant \lambda \cdot cap(e) && \forall e \in E(G) \\
x_{dp} &\in \{0,1\} && \forall p \in P_1,\ \forall d \in \mathcal{D}
\end{aligned}$$

The first set of constraints specifies that each demand has to be served by exactly one path. The second set of constraints ensures that no edge carries more than $\lambda \cdot cap(e)$ traffic.

The model that we use for our column generation solution is a generalization of this one where we replace the set of paths considered, P_1 by the set of all sr-paths with at most a given segment cost, $\vec{\mathcal{P}}_k(G)$. Renaud Hartert already had proposed this generalization in his thesis [32]. The only difference between his model and ours is that we also consider adjacency segments in our sr-paths whereas his model only considered node segments. We discuss our model in more detail in the next section.

In his thesis, Hartert also proposed another model called *segment model*. This model is closely related to the IP version of the model that we presented for the multi-commodity flow MCF-LP. In the integral MCF model, we use variables x_{ed} to indicate whether demand d is routed though edge e. Then we use classic flow conservation constraints to ensure that if $x_{ed} = 1$ then we have a path $(e_1, \ldots, e_l)$ starting at $src(d)$ and ending at $src(d)$ such that $x_{e_1 d} = 1, \ldots, x_{e_l d} = 1$. In the segment model, instead of edges we consider the shortest path subgraphs. Concretely, we use variables x^d_{uv} for $d \in \mathcal{D}, u, v \in V(G)$ defined such that $x^d_{uv} = 1$ if and only demand d is routed over the shortest paths between nodes u and v, that is, the subgraph $SP(u, v)$. Then we use the same flow conservation constraints to ensure that whenever $x^d_{uv} = 1$ then there exists a sequence of nodes $(v_1 = s, v_2, \ldots, v_k = t)$ such that $x^d_{v_1 v_2} = x^d_{v_2 v_3} = \ldots = x^d_{v_{k-1} v_k} = 1$ meaning that we route the demand d by following the shortest paths from v_1 to v_2, then from v_2 to v_3 and so on. In other words, we use sr-path $\langle v_1, v_2, \ldots, v_k \rangle$ to route d.

SRTE-SEG$(G, \mathcal{D})$

min λ **s.t.**

$$\sum_{d=1}^{r} \sum_{u,v \in V(G):u \neq v} x_{uv}^{d} \cdot \tau(u,v,e) \cdot vol(d) \leqslant \lambda \cdot cap(e) \quad \forall e \in E(G)$$

$$\sum_{u \in V(G)\setminus\{v\}} x_{uv}^{d} - \sum_{u \in V(G)\setminus\{v\}} x_{vu}^{d} = 0 \quad \forall d = (s,t,v) \in \mathcal{D}, \; \forall v \in V(G) \setminus \{s,t\}$$

$$\sum_{u \in V(G)\setminus\{s\}} x_{us}^{d} - \sum_{u \in V(G)\setminus\{s\}} x_{us}^{d} = -1 \quad \forall i \in \{1,\dots,r\}, \; \forall d = (s,t,v) \in \mathcal{D}$$

$$\sum_{u \in V(G)\setminus\{t\}} x_{ut}^{d} - \sum_{u \in V(G)\setminus\{t\}} x_{ut}^{d} = 1 \quad \forall i \in \{1,\dots,r\}, \; \forall d = (s,t,v) \in \mathcal{D}$$

$$x_{uv}^{d} \in \{0,1\} \quad \forall e \in E(G), \; \forall d \in \mathcal{D}$$

$$\lambda \geqslant 0$$

As it is, this model does not make any guarantees on the number of segments used to route a demand. We can overcome this by adding the following additional constraint limiting the maximum number of node segments in the sr-paths to k:

$$\sum_{u,v \in V(G):u \neq v} x_{u,v}^{d} \leqslant k \quad \forall d \in \mathcal{D}.$$

In all these models, by relaxing the integrability constraints to allow the variables to take any real value in $[0,1]$ we automatically get a polynomial time solvable LP for Problem 4 (over the restricted set of sr-paths that each model considers).

Model comparison

Due to its simplicity, optimal solutions of SRTE-1DET can be computed very efficiently. It it a relaxation of Problem 3 because it significantly restricts the sets of sr-paths that can be used by considering only paths in P_1. The generalization of this model consisting of replacing P_1 by $\vec{\mathcal{P}}_k$ removes this restriction by considering any possible sr-path whose segment cost is at most k to route any given demand. The problem of this model is that it contains a number of variables which is exponential with respect to k since $|\vec{\mathcal{P}}_k| = O(|G|^k)$. This means that even for small values of k it will not scale. Hartert showed in his thesis that even for $k \geqslant 3$ the number of variables is too big to allow good solutions to be found in a reasonable amount of computation time. We will show how we use column generation to overcome this problem the next section.

The segment model presented above overcomes this problem by implicitly representing sr-paths with flow conservation constraints. With this, it manages

to model sr-paths of segment cost up to k using only $|V(G)|^2 \cdot |\mathcal{D}|$ variables. One drawback of this solution is that it cannot represent sr-paths containing adjacency segments. Moreover, it turns out that on the larger topologies this model quickly becomes too large as well. This is not surprising since with one demand per pair of nodes the total number of variables is about $|V(G)|^4$.

Other researchers have proposed heuristic techniques for solving Problem 3. Hartert et al. proposed in DEFO [34,35] a Large Neighborhood Search technique combined with Constraint Programming. Later, Gay et al. proposed in [26] to use standard local search to iteratively improve the current solution. These heuristic approaches are rather efficient but provide no way of knowing how far from the optimal value they end up. Moreover, all the existing approaches only consider node segments in their formulation. They are thus not able to fully exploit the flexibility of segment routing with adjacency segments. Not only that but we have shown in our experiments that, on instances where only a few routing configuration lead to good solution, these heuristic approaches have difficulty of finding them. This is a common drawback of LS since in such cases it becomes very unlikely for the search to be able to reach these very precise configurations.

6.4 The idea behind Column Generation

We now describe how we used Column Generation (CG) [15] to overcome the model size problems faced when using the path model for solving the traffic engineering problem with SR. CG is usually used to solve linear programs with a huge number of variables. The idea is to solve the problem for a subset of the variables and then incrementally insert the missing variables, slowly growing the model. Each time that we add new variables, we solve the model again to check for optimality. One could ask what is the point of doing this since at some point we will have the whole set of variables, so why not just solve the full problem instead? However, we will carefully select the new variables that we introduce so that in general this framework will reach an optimal solution to the original *problem without ever needing to consider all variables.*

To understand the impact of ignoring a variable and how we can prove optimality without considering all variables, we work out a small example. This will allow us to explain the main idea behind CG in a much more intuitive way by providing an example without overwhelming mathematical notations. Suppose that we have the following LP that we want to solve.

Primal

$$
\begin{array}{lrcrcrcrcl}
\textbf{min} & 3x_1 & + & 3x_2 & + & 4x_3 & + & 1x_4 & & \\
\textbf{s.t} & 4x_1 & + & 5x_2 & + & 3x_3 & + & \tfrac{3}{2}x_4 & \geqslant & 6 \\
 & 5x_1 & + & 6x_2 & + & 4x_3 & + & 3x_4 & \geqslant & 7 \\
 & x_i \geqslant 0 & & & & & & & &
\end{array}
$$

The dual of this problem can be obtained by following a systematic procedure. By doing so, we obtain the following linear program.

Dual						
max	$6y_1$	+	$7y_2$			
s.t	$4y_1$	+	$5y_2$	$\leqslant$	3	constraint of x_1
	$5y_1$	+	$6y_2$	$\leqslant$	4	constraint of x_2
	$3y_1$	+	$4y_2$	$\leqslant$	3	constraint of x_3
	$\frac{3}{2}y_1$	+	$3y_2$	$\leqslant$	1	constraint of x_4
	$y_i \geqslant 0$					

Each variable x_i of the primal corresponds to a constraints of the dual. So, for instance, x_2 corresponds to constraint $5y_1 + 6y_2 \geqslant 4$ and x_4 corresponds to the constraint $\frac{3}{2}y_1 + 3y_2 \geqslant 1$ as highlighted in the models.

Because of this correspondence, when we ignore a variable on the primal, this translates in the dual to ignoring the corresponding constraint. For example, if we consider the restricted primal problem on variables x_1 and x_3 only (that we denote by $\mathsf{Primal}(1,3)$), and we take the dual, the dual will be the same as above but will only have the first and third constraints.

$\mathsf{Primal}(1,3)$						
min	$3x_1$	+	$4x_3$			
s.t	$4x_1$	+	$3x_3$	$\geqslant$	6	
	$5x_1$	+	$4x_3$	$\geqslant$	7	
	$x_i \geqslant 0$					

$\mathsf{Dual}(1,3)$					
max	$6y_1$	+	$7y_2$		
s.t	$4y_1$	+	$5y_2$	$\leqslant$	3
	$3y_1$	+	$4y_2$	$\leqslant$	3
	$y_i \geqslant 0$				

Suppose that we solve the restricted primal we obtain the optimal solution $x^* = (1.5, 0)$. By duality, we also obtain an optimal solution y^* of the restricted dual. In this case $y^* = (0.75, 0)$. From here, we want to be able to decide whether x^* is actually optimal for the original primal problem and, if not, what new variable x_2 or x_4 should we consider to improve the solution. By strong duality, we know that x^* is optimal for Primal if and only if y^* is optimal for Dual. Since the only difference between Dual and $\mathsf{Dual}(1,3)$ is that we removed two constraints from Dual, y^* contains values for *all* variables of the dual. Therefore, to check the optimality of y^* for Dual, we can plug its values into the missing constraints to check whether y^* satisfies them. If y^* happens to satisfy all of them, then it must be optimal for the unrestricted dual and by consequence, x^* is optimal for the unrestricted original primal. In this example, for the constraint corresponding to x_2 we have

$$5y_1 + 6y_2 = 5 \cdot 0.75 + 6 \cdot 0 = 3.75 > 3$$

and for the constraint corresponding to x_4 we have

$$2y_1 + 6y_2 = 2 \cdot 0.75 + 6 \cdot 0 = 1.5 > 1.$$

Both of them are violated so we cannot conclude that y^* is optimal for Dual. So we chose one of x_2, x_4 to be added to the model. For the sake of example, we choose x_2 obtaining (we highlighted in green the newly added column):

$$
\begin{array}{lllllllllll}
\mathsf{Primal}(1,\,2,\,3) & & & & & & & & & & \\
\textbf{min} & & 3x_1 & + & 3x_2 & + & 4x_3 & & & & \\
\textbf{s.t} & & 4x_1 & + & 5x_2 & + & 3x_3 & + & \geqslant & 6 \\
 & & 5x_1 & + & 6x_2 & + & 4x_3 & + & \geqslant & 7 \\
 & x_i \geqslant 0 & & & & & & & & &
\end{array}
$$

Solving this yields a primal solution $x^* = (0, 1.2, 0)$ and dual solution $y^*(0.6, 0)$. As before, we check whether this solution is optimal for the original problem by checking whether it satisfies the remaining constraints. In this case we only have one constraint left, the one corresponding to x_4. By plugging into it the values of y^* we get

$$\frac{3}{2}y_1 + 3y_2 = \frac{3}{2} \cdot 0.6 + 3 \cdot 0 = 0.9 \leqslant 1.$$

The constraint is satisfied so we conclude that y^* is optimal for Dual and by strong duality we conclude that $x^* = (0, 1.2, 0, 0)$ is optimal for Primal (note that we added a value of 0 for all ignored variables).

With this process, we were able to solve Primal without ever having to consider x_4. Of course this is just a small illustrative example so the gain is not significant but, on larger models with a huge amount of variables, this can save huge amounts of computation time. This finished our introductory example about column generation.

We are now going to abstract from it and describe the general approach. Lets write the problem we are aiming to solve in the following general form

$$
\begin{array}{lllllllllll}
\mathsf{Primal} & & & & & & & & & & \\
\textbf{min} & & c_1x_1 & + & c_2x_2 & + & \dots & + & c_nx_n & & \\
\textbf{s.t} & & a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & \geqslant & b_1 \\
 & & a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & \geqslant & b_2 \\
 & & \vdots & & & & & & & & \vdots \\
 & & a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & \geqslant & b_m \\
 & x_i \geqslant 0 & & & & & & & & &
\end{array}
$$

whose dual is easily shown to be

$$
\begin{array}{lllllllllll}
\mathsf{Primal} & & & & & & & & & & \\
\textbf{max} & & b_1y_1 & + & b_2y_2 & + & \dots & + & b_my_n & & \\
\textbf{s.t} & & a_{11}y_1 & + & a_{21}y_2 & + & \dots & + & a_{m1}y_m & \leqslant & c_1 \\
 & & a_{12}x_1 & + & a_{22}x_2 & + & \dots & + & a_{m2}y_m & \leqslant & c_2 \\
 & & \vdots & & & & & & & & \vdots \\
 & & a_{1n}x_1 & + & a_{2n}x_2 & + & \dots & + & a_{mn}y_m & \leqslant & c_n \\
 & x_i \geqslant 0 & & & & & & & & &
\end{array}
$$

In general, we need a better way for finding a new variable to introduce. In the previous example we did it by iterating over all remaining variables and

checking the corresponding constraint. Since CG is a framework that we want to use when we have a *huge* amount of variables, this process is not realistic as it will involve checking the same amount of constraints. The idea to overcome this is to express the problem of finding the variable as a another optimization problem which hopefully translates into something that is solvable by an algorithm that is faster than a simple brute-force iterations on the missing variables.

The first step to achieve this is to express the condition for a new variable to be a candidate mathematically. Keeping the notation used in the example, we see that each variable x_i is associated to a column of coefficients $c_i, a_{1i}, a_{2i}, \ldots, a_{mi}$ where the first is the objective function coefficient and the others are the constraints coefficients in the primal. The constraint associated to x_i is the dual is given by

$$a_{1i} + a_{2i} + \ldots + a_{mi} = \sum_{j=1}^{m} a_{ji} y_i \leqslant c_i.$$

Checking whether it is violated, or in other words, whether x_i needs to be considered in the Primal, consists of checking whether

$$\sum_{j=1}^{m} a_{ji} y_i > c_i \Leftrightarrow \sum_{j=1}^{m} a_{ji} y_i - c_i > 0.$$

If we let I be the set of indexes of the variables that we consider at a given point in the Primal, then the problem of finding a new variable can be expressed as finding $i \in \{1, \ldots, n\} \setminus I$ such that

$$\sum_{j=1}^{m} a_{ji} y_i - c_i > 0.$$

Therefore, if we compute the index $i^* \in \{1, \ldots, n\} \setminus I$ for which the value

$$\sum_{j=1}^{m} a_{ji^*} y_{i^*} - c_{i^*}$$

is *maximum* we know that there exists a new variable that we need to consider if and only if

$$\sum_{j=1}^{m} a_{ji^*} y_{i^*} - c_{i^*} > 0.$$

Therefore, we can solve the problem of finding a new variable to add by solving the problem

$$\max_{i \in \{1,\ldots,n\} \setminus I} \quad \sum_{j=1}^{m} a_{ji} y_i - c_i.$$

At first sight this problem might not seem any easier to solve than iterating over all $i \in \{1, \ldots, n\} \setminus I$ and computing the maximum but in practice this problem ofter has a nice structure and is usually solvable by a better, more efficient, algorithm than a simple brute force iteration. This problem is often referred to as the *pricing problem*.

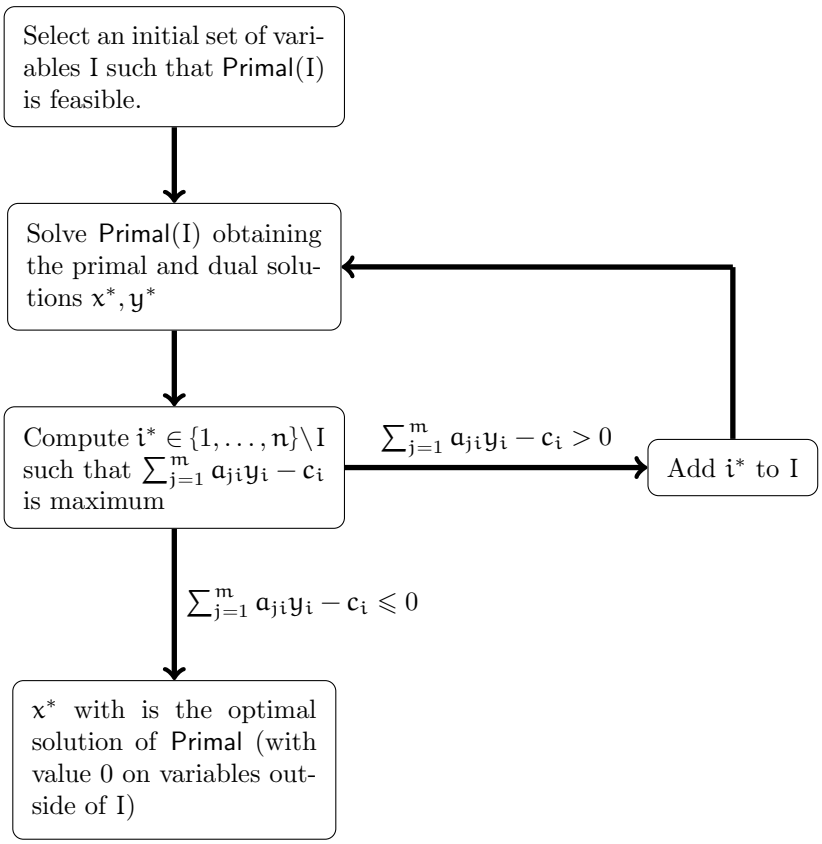


Figure 6.3: The CG framework used in this thesis.

Figure 6.3 provides a schematic vision of the column generation process. CG generation is a very wide field and what we described here is just a very small glimpse into it [15]. It is also important to note that what we did here only applies to *continuous* linear programs and not to integer programming. Therefore, we can only use this to compute solutions of LP that contain no integer variables. In order to obtain optimal integral solution we need to add another layer to the algorithm and perform, for instance, a branch-and-price [10]. We did not explore this in this thesis but we believe that it would be interesting future work to see how fast we can compute optimal solutions using this approach.

In this thesis we instead use an heuristic algorithm to round the fractional solution getting an approximated solution rather than an optimal one.

6.5 CG for the path model

We apply the process that we described above to the path model for the TE problem. As we mentioned before, this model has a huge amount of variables since we have one variable per sr-path in $\vec{\mathcal{P}}_k$ and $|\vec{\mathcal{P}}_k| = O(|G|^k)$. Recall that the

path model is obtained by replacing P_1 by $\vec{\mathcal{P}}_k$ on the formulation of SRTE-1DET.

SRTE-PATH$(G, \mathcal{D})$

min λ

$$\begin{aligned}
\textbf{s.t.} \quad \sum_{\vec{p} \in \vec{\mathcal{P}}_k} x_{d\vec{p}} &= 1 && \forall d \in \mathcal{D} \\
\sum_{\vec{p} \in \vec{\mathcal{P}}_k} \sum_{d \in \mathcal{D}} \tau(\vec{p}, e) \cdot vol(d) \cdot x_{d\vec{p}} &\leqslant \lambda \cdot \mathsf{cap}(e) && \forall e \in E(G) \\
x_{d\vec{p}} &\in \{0,1\} && \forall \vec{p} \in \vec{\mathcal{P}}_k,\ \forall d \in \mathcal{D}
\end{aligned}$$

Our experiments showed that working directly on the path model caused the column generation process described in Figure 6.3 to require adding a lot of columns before proving optimality. It would get stuck with the same optimal value for the restricted problem for huge number of iterations. For this reason we decided to formulate the problem with the same constraints but with the objective of routing a maximum amount of demands for a given factor λ.

Since $x_{d\vec{p}}$ says whether demand d is routed over sr-path $\vec{p}$, the total amount of traffic that is routed by a solution $x_{d\vec{p}}$ if given

$$\sum_{\vec{p} \in \vec{\mathcal{P}}_k} \sum_{d \in \mathcal{D}} vol(d) \cdot x_{d\vec{p}}.$$

Routing a maximum demand volume thus amounts at maximizing this value. Putting it together with the constraints from the path model SRTE-PATH we obtain the following model.

SRTE-DEM$(G, \mathcal{D}, \mathcal{P}, \lambda)$

$$\begin{aligned}
\textbf{max} \quad & \sum_{\vec{p} \in \vec{\mathcal{P}}_k} \sum_{d \in \mathcal{D}} vol(d) \cdot x_{dp} \\
\textbf{s.t.} \quad \sum_{\vec{p} \in \vec{\mathcal{P}}_k} x_{d\vec{p}} &\leqslant 1 && \forall d \in \mathcal{D} \\
\sum_{\vec{p} \in \vec{\mathcal{P}}_k} \sum_{d \in \mathcal{D}} \tau(\vec{p}, e) \cdot vol(d) \cdot x_{d\vec{p}} &\leqslant \lambda \cdot \mathsf{cap}(e) && \forall e \in E(G) \\
x_{d\vec{p}} \in \{0,1\} &\geqslant 0 && \forall \vec{p} \in \vec{\mathcal{P}}_k,\ \forall d \in \mathcal{D}
\end{aligned}$$

In this formulation, λ is a parameter that is given as input. We still want to minimize it so that we end up with a solution of Problem 3. However, we do not do this directly with linear programming, we instead perform a binary search on λ to find the minimum lambda that is able to route *all* demands. The other difference relative to the original path model is that on the first set of constraints we now have an inequality rather than an equality. Both are equivalent. This is the the case because, since the objective now is to route the maximum amount of demands, we always end up with a solution that selects one path per demand. The only reason for using the inequality is that it simplifies a bit the process of finding the dual.

We apply the process described in our CG introduction to this problem. Since CG needs a LP and SRTE-DEM is a MIP, we first relax integrality constraints by replacing $x_{d\vec{p}} \in \{0,1\}$ by $x_{d\vec{p}} \geqslant 0$. Note that, again, the most natural would have been to say $x_{d\vec{p}} \in [0,1]$ but saying $x_{d\vec{p}} \geqslant 0$ is equivalent for this problem because of the first set of constraints. The reason for this choice is again that is makes the process of computing the dual simpler because it leaves the formulation in a standard form. We also need to add the current set of variables as a parameter. Since in our case variables correspond to sr-paths, we pass as an argument a set of sr-paths $\mathcal{P}$ corresponding to the set of variables to which we restrict the problem. The CG process then slowly grow this set until we reach an optimal solution. The linear programming relaxation we obtain is the following.

SRTE-DEM-LP$(G, \mathcal{D}, \mathcal{P}, \lambda)$

$$\begin{array}{lrcll}
\textbf{max} & \displaystyle\sum_{\vec{p}\in\vec{\mathcal{P}}_k}\sum_{d\in\mathcal{D}} vol(d)\cdot x_{dp} & & & \\
\textbf{s.t.} & \displaystyle\sum_{\vec{p}\in\vec{\mathcal{P}}_k} x_{d\vec{p}} & \leqslant & 1 & \forall d \in \mathcal{D} \\
\displaystyle\sum_{\vec{p}\in\vec{\mathcal{P}}_k}\sum_{d\in\mathcal{D}} \tau(\vec{p},e)\cdot vol(d)\cdot x_{d\vec{p}} & & \leqslant & \lambda\cdot cap(e) & \forall e \in E(G) \\
& x_{d\vec{p}} & \geqslant & 0 & \forall \vec{p}\in\vec{\mathcal{P}}_k,\ \forall d\in\mathcal{D}
\end{array}$$

Since SRTE-DEM-LP is already in standard form it is very easy to obtain its dual. In general, computing the dual of a linear program can be achieved by following a systematic procedure. We omit the details here as this is quite a common process. Doing so we obtain the following formulation for the dual.

SRTE-DEM-DUAL$(G, \mathcal{D}, \mathcal{P}, \lambda)$

$$\begin{array}{lrcll}
\textbf{min} & \lambda\cdot\displaystyle\sum_{e\in E(G)} cap(e)\cdot y_e + \sum_{d\in\mathcal{D}} z_d & & & \\
\textbf{s.t.} & z_d + \sum_{e\in E(G)} \tau(\vec{p},e)\cdot vol(d)\cdot y_e & \geqslant & vol(d) & \forall \vec{p}\in\vec{\mathcal{P}}_k \\
& y_e, z_d & \geqslant & 0 & \forall \vec{p}\in\vec{\mathcal{P}}_k,\ \forall d\in\mathcal{D}
\end{array}$$

The next step is to describe the pricing problem. By what we said above, this corresponds to finding a sr-path $\vec{p} \in \vec{\mathcal{P}}_k$ and a demand $d \in \mathcal{D}$ that violate the dual constraints, that is, such that

$$\begin{gathered}
z_d + \sum_{e\in E(G)} \tau(\vec{p},e)\cdot vol(d)\cdot y_e < vol(d) \Leftrightarrow \\
\sum_{e\in E(G)} \tau(\vec{p},e)\cdot vol(d)\cdot y_e < vol(d) - z_d \Leftrightarrow \\
\sum_{e\in E(G)} \tau(\vec{p},e)\cdot y_e < \frac{vol(d)-z_d}{vol(d)}
\end{gathered}$$

If such $\vec{p}$ and d exist then we will add $\vec{p}$ to the $\mathcal{P}$. If not, then, by what we said above, we know that the optimal solution for the restricted sr-path set $\mathcal{P}$ is also optimal for the complete sr-path set $\vec{\mathcal{P}}_k$.

Problem 5 (SRTE pricing)

Input: *A network* G*, a demand set* $\mathcal{D}$*,* $k \geqslant \mathbb{N}$ *and values* $y_e \geqslant 0$ *for each* $e \in E(G)$*.*
Output: *A sr-path* $\vec{p} \in \vec{\mathcal{P}}_k$ *and a demand* $d \in \mathcal{D}$ *such that*

$$\sum_{e \in E(G)} \tau(\vec{p}, e) \cdot y_e < \frac{vol(d) - z_d}{vol(d)}$$

or report that no such path exists.

Theorem 6.2. *Problem 5 can be solved in polynomial time.*

Proof. Let $d \in \mathcal{D}$. Consider the sr-metric c defined for all $u, v \in V(G)$ and $e \in E(G)$ as

$$c(u, v) = \sum_{e \in E(G)} \tau(u, v, e) \cdot y_e$$

and

$$c(e) = y_e.$$

Let know that we can compute a sr-path $\vec{p} = \langle x_1, \ldots, x_l \rangle \in \vec{\mathcal{P}}_k$ that minimizes

$$c(\vec{p}) = \sum_{i=2}^{l} c(x_{i-1}^2, x_i^1) + \sum_{i : x_i \in E(G)} c(x_i)$$

in polynomial time using Algorithm 4 from Chapter 5. We have that

$$\begin{aligned}
c(\vec{p}) &= \sum_{i=2}^{l} c(x_{i-1}^2, x_i^1) + \sum_{i : x_i \in E(G)} c(x_i) \\
&= \sum_{i=2}^{l} \sum_{e \in E(G)} \tau(x_{i-1}^2, x_i^1, e) \cdot y_e + \sum_{i : x_i \in E(G)} y_{x_i} \\
&= \sum_{e \in E(G)} \sum_{i=2}^{l} \tau(x_{i-1}^2, x_i^1, e) \cdot y_e + \sum_{e \in E(G)} \sum_{i : x_i = e} y_e \\
&= \sum_{e \in E(G)} \left(\sum_{i=2}^{l} \tau(x_{i-1}^2, x_i^1, e) \cdot y_e + \sum_{i : x_i = e} y_e \right) \\
&= \sum_{e \in E(G)} \left(\sum_{i=2}^{l} \tau(x_{i-1}^2, x_i^1, e) + \sum_{i : x_i = e} 1 \right) \cdot y_e \\
&= \sum_{e \in E(G)} \tau(\vec{p}, e) \cdot y_e.
\end{aligned}$$

Since $\vec{p}$ has minimum cost, Problem 5 has a solution if and only if

$$c(\vec{p}) = \sum_{e \in E(G)} \tau(\vec{p}, e) \cdot y_e < \frac{vol(d) - z_d}{vol(d)}$$

in which case $\vec{p}$ is a solution. Since Algorithm 4 runs in polynomial time, this means that we can decide in polynomial time whether for a given demand $d \in \mathcal{D}$ there exists a sr-path $\vec{p}$ that violates the dual constraints. Thus by iterating over all demands we get a polynomial time algorithm overall. □

With this theorem we have an example of a problem where finding a new variable to add can be done in a much more efficient way than iterating over all remaining variables. Using the schema in Figure 6.3 we can then compute optimal solutions to the LP relaxation SRTE-DEM-LP of SRTE-DEM.

The solution obtained with this process is optimal for Problem 4 and provides a lower bound form Problem 3. In itself this is already a good result since those bounds are important for evaluating the quality of greedy solution. Until now, the best lower bounds were computed using the MCF. The bounds obtained with the MCF are of lower quality since they do not take into account segment routing as show at the end of this chapter.

6.6 Minimizing the worst link utilization λ

We saw in the previous section how to use CG to compute the maximum demand volume that we can route for a given capacity factor λ. In this section we explain how to use this as a sub-routine to compute the minimum λ for which we can route *all* demands.

Let $\mathcal{V} = \sum_{d \in \mathcal{D}} vol(d)$ be the total volume of demands in $\mathcal{D}$. To find the minimum capacity factor perform a binary search on $\lambda \in [0, \lambda_M]$ where λ_M is a big enough capacity factor that ensure that all demands can be routed. For each λ we solve SRTE-DEM$(G, \mathcal{D}, P, \lambda)$. If the solution is $\mathcal{V}$ then we continue the search with λ as the new upper bound. Otherwise we set λ as the new lower bound and continue. Each time we solve SRTE-DEM$(G, \mathcal{D}, P, \lambda)$ we do not start from scratch. We continue the CG process with the set of paths P from the previous iterations.

Initial sr-path set

We need to select an initial feasible sr-path set $\mathcal{P}$. In this case, feasibility is not an issue as we can select the capacity factor λ_M so that it is large enough to make any solution with at least one sr-path per demand feasible. To compute the initial path set, we use Algorithm 5 to compute the maximum capacity sr-path for each demand.

Generating new paths

We explained above that we can solve the pricing by iterating over all demands $d \in \mathcal{D}$ and computing a minimum cost sr-path for specific weights that depend on the values y, z of the dual solution. There might be several sr-path demand pairs $\vec{p}, d$ that violate the dual constraints and each such path corresponds to a

new candidate variable that might improve the current solution if added. However, if there are a lot of demands, computing the minimum cost sr-path for each and every one of them may be time consuming, even being a polynomial time procedure. Also, adding a lot of new paths makes solving the relaxed problem slower since it will contain more variables. On the other hand, having more paths will maybe make the CG converge faster towards the optimal solution. Hence, we face a trade-off between adding more paths and making each iteration slower or adding fewer paths and having more iterations overall.

To have more control over this, we add a parameter to our algorithm $\mathtt{maxp}$ that limits the number of sr-paths than can be generated in a single iteration. At each CG iteration we loop over all the demands and for each of them check if it has a sr-path to be added. After $\mathtt{maxp}$ demands yield a new sr-path we stop this iteration and solve the new linear program obtained by adding these paths. The order in which we iterate over the demands should be such that we start with demands that have a higher chance of yielding a sr-path that violates the dual constraints, that is, a sr-path $\vec{p}$ such that

$$\sum_{e \in E(G)} \tau(\vec{p}, e) \cdot y_e < \frac{vol(d) - z_d}{vol(d)}.$$

Hence, we sort the demands by decreasing value of $\frac{vol(d)-z_d}{vol(d)}$ since the higher this value is, the more likely we are of finding a sr-path whose weight lower cost. This is important since it helps avoiding useless time consuming invocations of the minimum cost sr-path algorithm.

Rounding heuristic

As we explained above, at the end of the column generation process, we have an optimal solution for the relaxed master problem SRTE-DEM-LP where variables can have fractional values. In practice, this means that each demand might be split over several paths, in other words, we have an optimal segment routing solution for Problem 4. If we seek a solution from Problem 3 we need to ensure that each demand is assigned to exactly one path. If sub-optimal solutions are acceptable, then an efficient way to achieve this is to use some kind of heuristic to assign a path to each demand amongst the paths obtained in the end. We propose to do this by re-solving SRTE-PATH with integrality constraints using a MIP solver. As we discuss later, our experiments showed that in this way we obtain near optimal solutions to Problem 3. This is an heuristic since, with integer variables, there is no guarantee that the restricted optimal solution is the optimal for the unrestricted problem.

By putting all these ideas together we can provide the following formal description of our column generation algorithm for the traffic engineering problem. Algorithm 7 performs one iteration of the column generation algorithm. It starts on line 1 by computing the optimal solution of the linear program SRTE-DEM-LP for the current sr-path set obtaining a primal solution x, the corresponding dual solution y, z and the maximum value $\mathtt{vol}$ that we can be routed on the current path set for the current capacity factor λ. Afterwards, from line 2 to line 5 it computes the sr-metric used for the minimum cost sr-path computation. Then it iterates over all demands and tries to find new paths to add. All those paths are put together and returned.

Algorithm 7 iterate-CG $(\mathsf{G}, \mathcal{P}, \mathcal{D}, \lambda, \mathit{maxp})$

$x, (y, z)_x, \mathit{vol} \leftarrow$ LP-solve(SRTE-DEM-LP$(\mathcal{P}, \mathcal{D}, \lambda)$)
for $x, y \in V$ **do**
  $c(x, y) \leftarrow \sum_{e \in E(\langle x,y \rangle)} \tau(x, y, e) \cdot y_e$
for $e \in E$ **do**
  $c(e) \leftarrow y_e$
$P' \leftarrow \emptyset$
for $d \in \mathcal{D}$ **in decreasing order of** $(\mathit{vol}(d) - z_d)/\mathit{vol}(d)$ **do**
  $p \leftarrow$ mincost-srpath$(s(d), t(d), c)$
  if $c(p) < (\nu(d) - \delta_d)/\nu$ **then**
    $P' \leftarrow P' \cup \{p\}$
  if $|P'| \geqslant \mathit{maxp}$ **then**
    break
return P'

Algorithm 8 binsearch-CG $(\mathcal{D}, \lambda_M, k, \epsilon, \mathit{maxp})$

$\mathcal{P} \leftarrow \{$maxCapSrPath$(g, \tau, s, t, k) \mid (s, t, d) \in \mathcal{D}\}$
$\mathcal{V} \leftarrow \sum_{d \in \mathcal{D}} \mathit{vol}(d)$
$lb \leftarrow 0$
$ub \leftarrow \lambda_M$
while $|lb - ub| > \epsilon$ **do**
  $\lambda \leftarrow (lb + ub)/2$
  $x, (y, z)_x, \mathit{vol} \leftarrow$ column-generation$(\mathcal{P}, \mathcal{D}, \lambda, \mathit{maxp})$
  if $\mathit{vol} = \mathcal{V}$ **then**
    $lb \leftarrow \lambda$
  else
    $ub \leftarrow \lambda$
return ILP-solve(SRTE-DEM$(\mathcal{P}, \mathcal{D})$)

Algorithm 9 performs column generation iterations by calling Algorithm 7 until no new paths are found thus computing an optimal solution of SRTE-DEM-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ for a given capacity factor λ over the full sr-path set $\vec{\mathcal{P}}_k$. Finally, Algorithm 8 uses the column generation as a sub-routine on a binary search to find the smallest value of λ such that the optimal solution of SRTE-DEM-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ is equal to $\mathcal{V}$. In other words, it uses a binary search combined with the column generation to find the smallest capacity factor that allows one to route the whole volume of demands.

6.7 CG experimental results

This section describes the results using the column generation approach described in this chapter and denoted CG4SR here after.

We use Repetita [27] to run all the other solvers: DefoCP [34,35], Bhatia [11] and SRLS [26]. We reuse the demand matrices generated for Repetita [27]. For each topology, they generated 5 demand matrices through the gravity model described in [44]. Demands were normalized so that MCF can merely force all link loads to be below or equal to 90%. The number of demands ranges from 7482 to 98910.

Algorithm 9 column-generation $(\mathcal{P}, \mathcal{D}, \lambda, maxp)$

1: **while** $\mathsf{P}' \leftarrow$ iterate$(\mathcal{P}, \mathcal{D}, \lambda, maxp) \neq \emptyset$ **do**
2: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \mathsf{P}'$
3: **return** LP-solve(SRTE-DEM$(\mathsf{G}, \mathcal{D}, \mathcal{P}, \lambda)$)

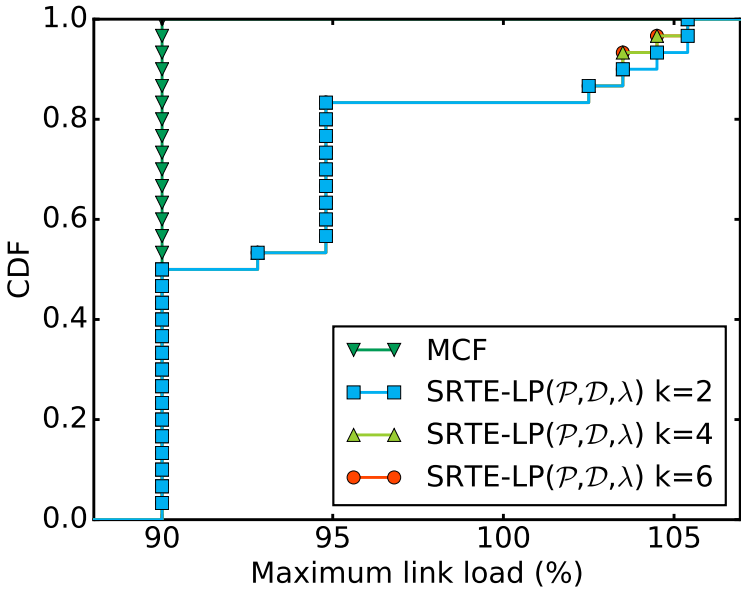


Figure 6.4: Lower bound

6.7.1 Near-optimum evaluation

CG4SR provides a better lower bound for TE over SR than MCF. Traditionally, the value of an optimal MCF solution is used as a lower bound for minimum maximum link utilisation that one can achieve for routing a traffic matrix. However, as mentioned above, this bound is unrealistic as MCF is oblivious to SR. Figure 6.4 studies the quality of the lower bound provided by CG4SR compared to MCF. The load predicted by MCF is always of about 90% because the demand matrices of Repetita [27] were generated artificially to be at this value. However, CG4SR shows that it is strictly impossible to escape network congestion for 5 demand matrices. Moreover, the difference between CG4SR and MCF lower bounds can be as high as 15% in the predicted maximal load. Increasing the number of segments does not get CG4SR lower bound much closer to the MCF.

CG4SR provides solutions whose maximum load is at most 4% more than the optimal solution. We ran CG4SR without enabling adjacency segments and without time limit. The experiment was repeated with limits of 2, 4 and 6 segments to observe the impact of the segment limit on the quality of the solution. Figure 6.5 shows CDFs of the gap (in percents) between the CG4SR upper and lower bounds on all the 30 instances (i.e., 5 demand matrices for each of the 6 topologies) and increasing the limit on the number of segments. This gap is the maximum distance to the actual optimum. We can

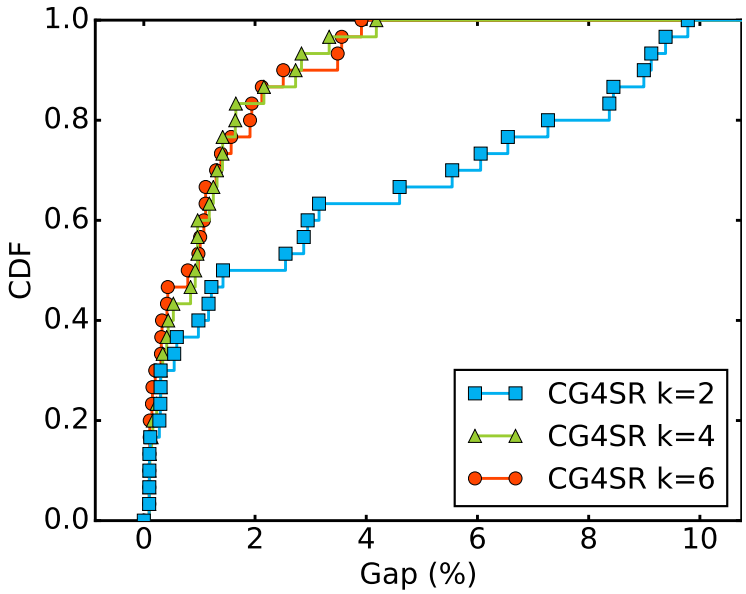


Figure 6.5: Gap

see that increasing the limit from 2 to 4 segments impacts the quality of the solution while increasing the limit from 4 to 6 has little impact. Paths with 4 or 6 segments add more flexibility than paths with 2 segments to SRTE-UTIL-ILP. We can see that this gap is most of the time below 1% of the load and at worst 4% of the load if 4 segments are allowed.

CG4SR is more efficient than Bhatia and MCF. We compared the speed of CG4SR to Bhatia, MCF and MCFP. MCFP is an efficient variant of MCF that is only able to compute the optimal value of MCF, not the actual routing paths. Figure 6.6 describes how fast the different solvers can find their best solution. During these runs, the limit on the number of generated paths at each column generation iteration, *maxp* in Algorithm 7, is fixed to 10. This figure shows a CDF of the execution time on the different topologies and demand matrices for CG4SR, Bhatia and MCF. Because Bhatia only allows two segments, CG4SR is also limited to two segments in this figure. MCF is the slowest one and it runs out of memory for all the demand matrices of the largest topology despite the 120GB available. Bhatia only considers paths that can be expressed with two segments. This significantly reduces the problem size and Bhatia can always get an answer. CG4SR can run with any number of two segments because of the lazy generation of the paths. And this is so effective than we are actually faster than Bhatia with a two-segment limit.

Figure 6.6 also shows that the MCFP variant of MCF can actually compute the optimal value of the MCF quicker than CG4SR. But, as mentioned above, MCFP only provides the maximum link utilisation of the MCF formulation but not a set of paths satisfying it. Hence, in practice, MCFP can only be used to provide lower bounds which, as we showed in Figure 6.5, are worse than the ones provided by our algorithm.

CG4SR scales better than MCF and Bhatia. Our model has fewer

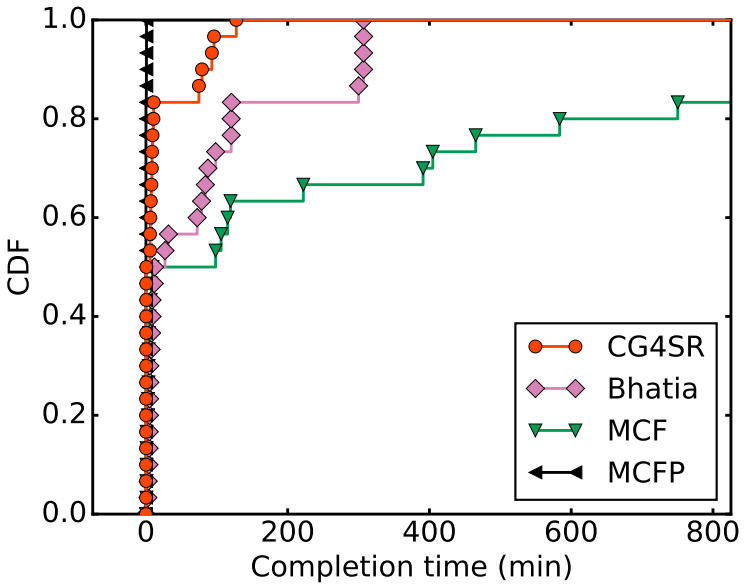


Figure 6.6: Execution time (CG4SR is limited to two segments)

variables than the MCF and Bhatia formulations. The size of our model is the number of paths that were generated. The size of MCF considers how much of each demand can be placed on each edge. Therefore, the number of variables is the number of edges multiplied by the number of demands. Bhatia considers for each demand, two segments through a single node of the graph. Its number of variables is thus the number of nodes multiplied by the number of demands. We observe that CG4SR is more scalable because it considers at worst 60 times fewer possibilities than Bhatia and 200 times fewer than MCF. This explains why CG4SR is faster than Bhatia and MCF. This difference does not change significantly when varying the limit on the number of segments. As can be seen in Figure 6.7, the number of generated columns seems to grow linearly with the number of demands. Given that the restricted path set is initialized with all the direct paths for every demand, the path-finding process (Algo 7) only creates a limited number of additional paths to reach optimality. This also explains why the column generation approach is so efficient in practice, as it only needs to solve the linear program with a number of variables only slightly above the number of demands.

6.7.2 Any-time behavior

The previous section shows that we can produce quality solutions with illimited time budget. This section evaluates the quality of CG4SR solutions over time. We compare CG4SR to the heuristic approaches DEFO, SRLS and also to Bhatia.

CG4SR finds good solution even if only allowed to run for a short amout of time. Figures 6.8, 6.9 and 6.10 show, for each of the cited solvers, a CDF of the gap to the SRTE-LP solution for the cited solvers after, respectively

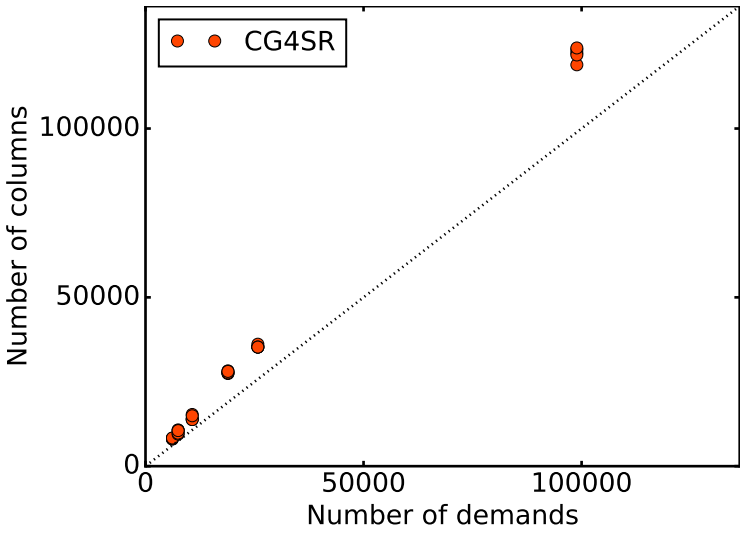


Figure 6.7: Number of generated columns over the size of the demand matrices

1 minute, 5 minutes and 10 minutes. During these runs, the *maxp* parameter (see Algorithm 7) of CG4SR, is fixed to 10. The limit of segments is set to 5, except for Bhatia which limits itself to 1. The quality of a solution is the difference between its current solution and the CG4SR lower bound that was computed without time limit and the same limit on the number of segments.

We see that we are always faster than Bhatia even with limited time spans. SRLS and DEFO are heuristic approaches and therefore are able to quickly find good solutions. Figure 6.8 shows that CG4SR is already comparable to SRLS and better than DEFO for half of the instances after 1 minute. We see that DEFO initially finds better solutions but CG4SR catches up for most of instances by increasing the timeout in Figure 6.9 and Figure 6.10. The largest instance is not yet solved after 10 minutes and that explains why DEFO is still better.

CG4SR is more robust than SRTE over different sets of demands. SRLS produces good results but however this solution is based on local search and can be stuck in a local optimum. We did not observe it on the demand matrices generated by the gravity model. The demand volumes are generally much lower than link capacities. This also means that there are many possible ways to reach good solutions, even if the best solution is hard to find. The gravity model is a good match to the Traffic Engineering problem in ISPs but demand volumes are likely higher in inter-datacenter communication [36]. This also means that there are fewer good solutions. We generated one additional demand matrix for each RocketFuel topology with a low number of large demands requiring 95% of the bandwidth available between their source and destination. Figure 6.11 shows a CDF of the quality of the solution with a time limit of 10 minutes and a limit of six segments as in Figure 6.10. These results confirm that SRLS can be worse than CG4SR when fewer good solutions are available.

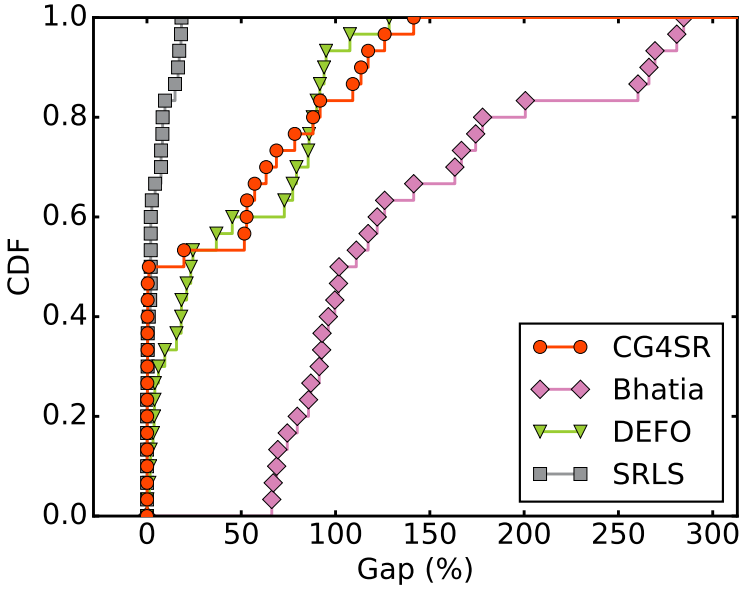


Figure 6.8: Timeout at 1 minute

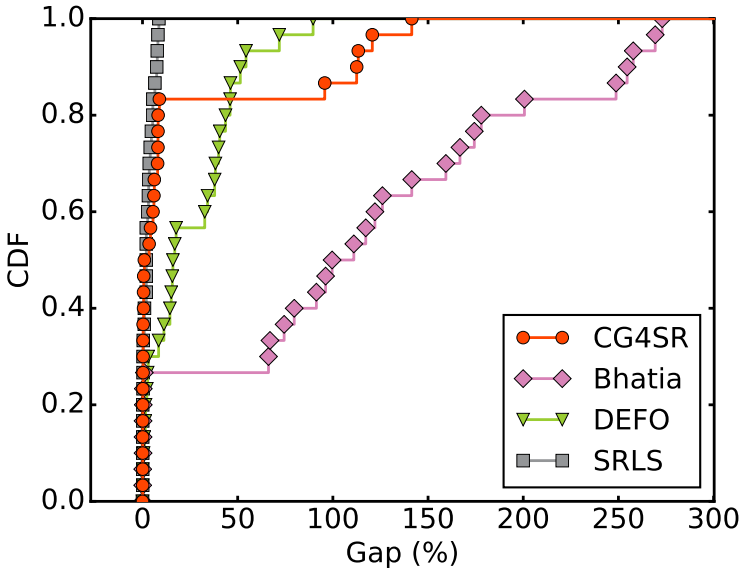


Figure 6.9: Timeout at 5 minutes

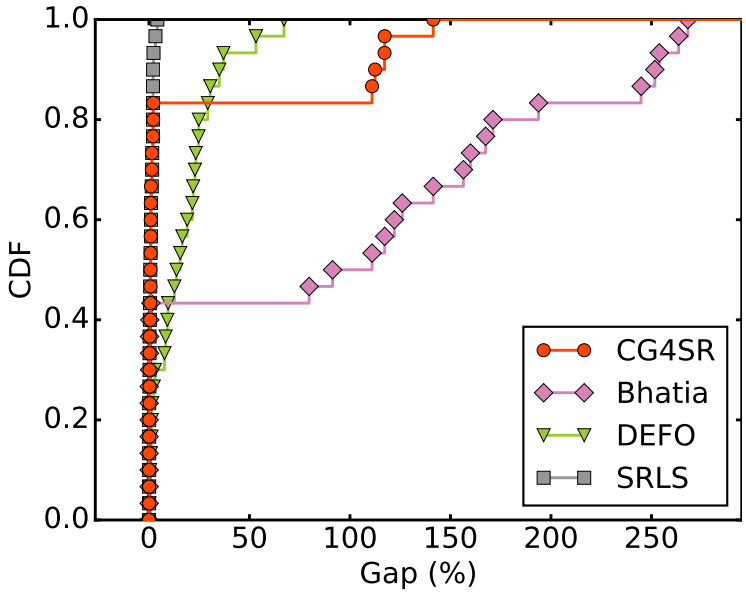


Figure 6.10: Timeout at 10 minutes

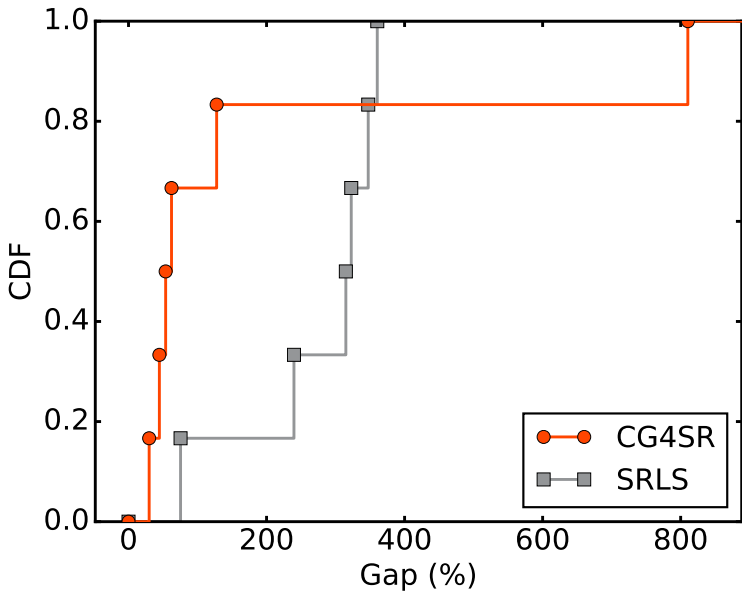


Figure 6.11: Gaps between $\mathsf{SRTE}-\mathsf{LP}(\mathcal{PD}\lambda)$ and SRLS or CG4SR after 10 min with a limit of 6 segments

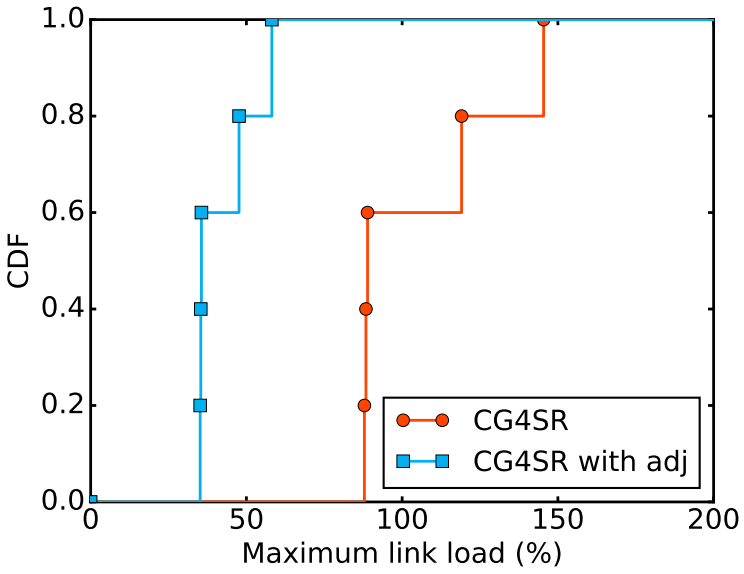


Figure 6.12: The CG4SR solutions with or without adjacency segments. The limit of segments is fixed to 4.

6.7.3 Adjacency segment benefits

Adjacency segments are important in TE and CG4SR is the first to use them. CG4SR is the first SR traffic engineering model to support adjacency segments. We evaluate the benefits of adjacency segments on the inter-datacenter network topology of OVH in Europe (described in [7]). This topology has more parallel links than RocketFuel topologies and thus, the OVH topology can really benefit from adjacency segments. We do not have access to the link weights and capacities of OVH. Therefore, for each link bundle we set the capacity of half of the links to some value and the other half to half of that value. This simulates the link upgrades on the network. For pairs of nodes with a single link between them, we set the capacity to be ten times bigger. Five demand matrices were generated for the OVH topology with the gravity model [44].

Figure 6.12 shows CDFs of the gap (in percents) between the CG4SR upper and lower bounds over the demand matrices of the OVH topology. We do not limit the execution time and we limit the number of segments to 4. This means that we allow at most one detour through a specific link because one segment is needed for the destination and a link detour costs two segments. Even allowing only one link detour halves the load of the maximally loaded link because it utilizes better the parallel links of this topology.

Conclusion

In this chapter we propose the first solution to exploit column generation to solve segment routing problems. We believe that column generation is a good approach for solving segment routing problem and reinforce this belief in the

next chapter. The structure of sr-paths make it amiable to dynamic programming algorithms so we feel that it will often be the case that the pricing problem will have a nice DP optimal substructure.

Our solution improves the state of the art lower bound making it possible to better evaluate the quality of heuristic solutions. We also showed that even tough we are slower on demands generated according to a gravity model, with more constrained demands we can actually be much more efficient than local search.

It still remains an open problem to find an algorithm capable of providing optimal solution to the TE problem over segment routing within reasonable amount of time. We are unsure whether wrapping our solution with a branch-and-price is the right way to go but it is certainly an interesting possibility.

Chapter 7

Network monitoring with segment routing

Introduction

Monitoring is a crucial task for network operations. It is needed to ensure that all resources operate correctly (e.g., no failures) and their configuration meets operator's expectations (no congestion, required quality of service, etc.). Effective monitoring is also fundamental for management tasks like traffic engineering, maintenance and troubleshooting.

Unfortunately, even basic monitoring tasks, like checking for hardware malfunctions, are practically hard, due to the complexity of current networks. Prominently, multi-path routing is widely used, both to spread the load on multiple paths and to aggregate parallel links in bundles. Figure 7.1 shows an overview of the European backbone of a big cloud provider, OVH. It highlights that parallel links are used at the same time between many pairs of routers. While enabling better performance and robustness, multi-path routing also poses significant obstacles to monitoring. For instance, assessing the exact path and performance of each packet becomes complex [2], [20] since such a path depends on (vendor-specific) hash functions used by routers for load-balancing.

As a consequence, not only naive approaches (e.g., based on ping or traceroute) are not sufficient, but also state-of-the-art monitoring techniques tend to be ineffective.

On the one hand, protocol-based approaches use control-plane messages to infer possible failures. For example, link-state routing protocols (like OSPF or IS-IS) or specialized ones (BFD [14]) rely on heartbeat-like mechanisms to check bi-directional connectivity among pairs of adjacent nodes. This approach only ensures detection of failures that affect control plane messages. However, it cannot be used to detect failures that only affect data-plane traffic like:

i) corruption of an optical link that leads to framing errors and packet losses;

ii) malfunctioning of a router interface that considers the link still up but discards all the received packets;

iii) failure of only one link among the parallel ones between two routers.

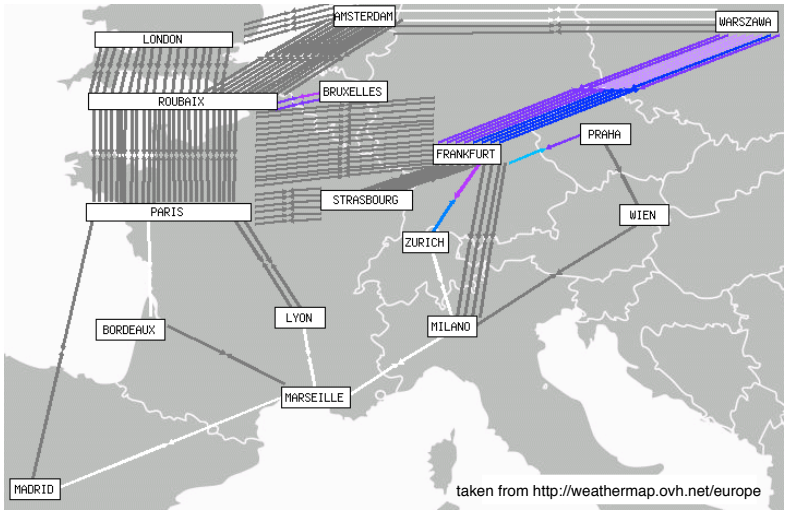


Figure 7.1: European backbone of OVH. In contrast to prior techniques, our algorithm can monitor health and performance of single links in bundles (e.g., between LONDON and ROUBAIX).

On the other hand, probe-based techniques rely on sending data-plane monitoring packets, i.e., probes, between fixed vantage points in the network. Vantage points typically run standard protocols (e.g., IPSLA [8]) to send probes and extract measurements from them. Unfortunately, if the probes are sent over paths used to forward regular traffic, many vantage points may be needed to obtain high coverage, and links not used by current paths (e.g., backup links) cannot be checked at all. Otherwise, probes can be sent over tunnels (e.g., RSVP-TE [3] ones) to enforce specific paths, but this is not scalable. Indeed, even for detecting single-link failures and pinpointing their position, the number of needed tunnels tends to explode, and so does the control-plane overhead (to signal tunnels) [7].

We propose a new technique that ensures full coverage of all network resources from a single vantage point. It is based on sending data-plane probes over carefully-chosen cycles. This way, a single box can both send and receive monitoring probes, avoiding the need for synchronizing and coordinating multiple vantage points, hence minimizing infrastructural costs. By relying on data-plane measurements, we support both detection of hardware failures and resource overloading (e.g., link congestion).

As usual, we start by defining the problems that we tackle in this chapter. To do that we need the two following definitions. Throughout this chapter we assume that the network topology is symmetric.

Definition 7.1. *Let* G *be a symmetric network,* $s \in V(G)$ *and* $k \in \mathbb{N}$*. We denote by* $\mathcal{C}_s^k$ *the set of deterministic sr-cycles from* s *to* s *with segment cost at most* k*.*

Definition 7.2. *Let* G *be a symmetric network,* $s \in V(G)$ *and* $k \in \mathbb{N}$*. A* k sr-cycle cover *of a network* G *with vantage point* s *is a subset* $C \subseteq \mathcal{C}_s^k$ *of*

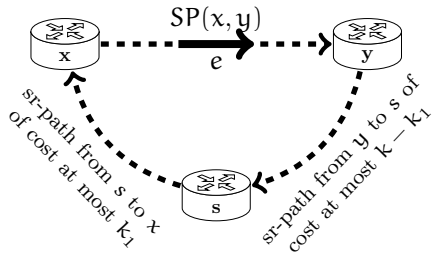


Figure 7.2: First case of edge covering.

deterministic sr-cycles such that for each edge $e \in E(G)$ *there exists* $\vec{c} \in C$ *such that* $e \in E(\vec{c})$.

Below we define the two problems that we tackle in this chapter. Both of them seek to compute a cycle cover. The first aims at minimizing the maximum segment cost of any sr-cycle in the cover while the second wants to minimize the number of cycles for a given segment cost limit.

Problem 1 (Min segment cost cover)

Input: *A symmetric network* G *and* $s \in V(G)$.
Output: *A sr-cycle cover* C *of* G *such that the maximum segment cost of any sr-cycle* $\vec{c} \in C$ *is minimum.*

Problem 2 (Min sr-cycle cover)

Input: *A symmetric network* G, $s \in V(G)$ *and* $k \in \mathbb{N}$.
Output: *A sr-cycle cover* $C \subseteq \vec{\mathcal{C}}_k^s$ *of* G *such that* $|C|$ *is minimum.*

7.1 Minimum segment cost covers

In this section we propose a polynomial time algorithm for solving Problem 1. We have seen in Chapter 4 reachability results that tells us exactly how costly, in terms of segments, it is to deterministically reach a given node in the network. Using these results we can easily establish how costly it is to cover a given edge with a deterministic sr-cycle.

In order to cover a given edge e with a deterministic sr-cycle from s to s with segment cost at most k, we have essentially two options:

1. First we go from s to some node x using a deterministic sr-path of cost, say, k_1. Then travel from x to some node y via a unique shortest path that contains edge e. Finally, we go from y back to s with a deterministic sr-path of cost $k_2 = k - k_1$. This is illustrated in Figure 7.2.

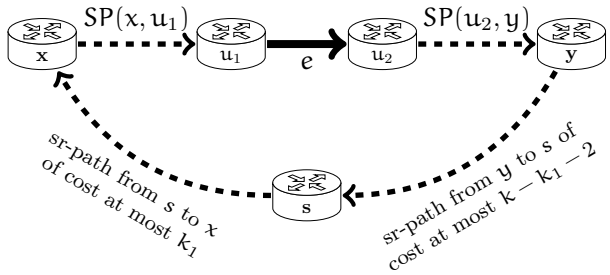


Figure 7.3: Second case of edge covering.

2. The second way to cover e is to use an adjacency segment over it. To do so, we first go from s to some node x with a deterministic sr-path of cost at most, say, k_1. This node x must be such that it contains a unique shortest path to e^1. Otherwise the resulting cycle would not be deterministic. Then we use an adjacency segment on e to traverse it by following the unique shortest path between x and e^1 and then e. Then we go to some node y via a unique shortest path from e^2. Finally, we go from y back to s with a sr-path of cost at most $k_2 = k - k_1 - 2$. The -2 comes from the fact that we spent a segment cost of 2 with the adjacency segment e. This case is illustrated in Figure 7.3.

Note that in both cases, nothings prevents us to have $x = e^1$ or $y = e^2$. We do not require these nodes to be distinct as doing so would provide an incomplete description of all possible cases.

Next we prove formally that those two cases cover all possibilities. The next theorem directly translate into a polynomial time algorithm for computing minimum segment cost covers.

Theorem 7.1. *Let G be a network $s \in V(G)$, $e \in E(G)$ and $k \in \mathbb{N}$ an integer. There exists a deterministic sr-cycle $\vec{c}$ from s to s of cost at most k such that $e \in E(\vec{c})$ if and only if there exist integers $k_1, k_2 \geqslant 1$, $x \in \mathit{reach}(k_1, s)$, y such that $s \in \mathit{reach}(k_2, y)$ and one of the two following conditions holds*

(1) $k_1 + k_2 = k$, $y \in \mathit{sp\text{-}reach}(x)$ *and* $e \in SP(x, y)$

(2) $k_1 + k_2 = k - 2$, $e^1 \in \mathit{sp\text{-}reach}(x)$ *and* $y \in \mathit{sp\text{-}reach}(e^2)$

Proof. Let G be a network $s \in V(G)$, $e \in E(G)$ and $k \geqslant 1$ an integer.

($\Rightarrow$) Assume that there exists a deterministic sr-cycle $\vec{c}$ from s to s with $\mathsf{sr\text{-}cost}(\vec{c}) \leqslant k$ such that $e \in E(\vec{c})$. Write $\vec{c} = \langle x_1, \dots, x_l \rangle$. Since $e \in E(\vec{c})$ there exists $i \in \{1, \dots, l\}$ such that either $x_i = e$ or $i < l$ and $e \in SP(x_i^2, x_{i+1}^1)$.

Case 1: $e \in SP(x_i^2, x_{i+1}^1)$. Let $x = x_i^2$ and $y = x_{i+1}^1$. Then $\vec{p} = \langle x_1, \dots, x_i \rangle$ is a deterministic sr-path from s to x of cost, say, k_1 and $\vec{q} = \langle x_{i+1}, \dots, x_l \rangle$ is a deterministic sr-path from y to s of cost $k_2 \leqslant k - k_1$. Therefore, $x \in \mathsf{reach}(k_1, s)$, $s \in \mathsf{reach}(k_2, y)$. Since $\vec{c}$ is deterministic, there is a unique shortest path from x to y so $y \in \mathsf{sp\text{-}reach}(x)$. Since by hypothesis $e \in SP(x, y)$, condition (1) holds.

Case 2: $e = x_i$. Let $x = x_{i-1}^2$ and $y = x_{i+1}^1$. Then $\vec{p} = \langle x_1, \dots, x_{i-1} \rangle$ is a deterministic sr-path from s to x of cost, say k_1, and $\vec{q} = \langle x_{i+1}, \dots, x_l \rangle$ from y to

s of cost, say k_2, such that $k_1{+}k_2 = \mathsf{sr\text{-}cost}(\vec{p}){+}\mathsf{sr\text{-}cost}(\vec{q}) = \mathsf{sr\text{-}cost}(\vec{c}){-}2 \leqslant k{-}2$. Thus $x \in \mathsf{reach}(k_1, s)$ and $s \in \mathsf{reach}(k_2, y)$. Since $\vec{c}$ is deterministic, there is a unique shortest path from $x = x^2_{i-1}$ and $e^1 = x^1_i$. For the same reason, there exists a unique shortest path from $e^2 = x^2_i$ to $y = x^1_{i+1}$. Thus $e^1 \in \mathsf{reach}(2, x)$ and $y \in \mathsf{sp\text{-}reach}(e^2)$ so that condition (2) holds.

Note that in the first case we have $k_1{+}k_2 \leqslant k$ and in the second $k_1{+}k_2 \leqslant k{-}2$ instead of the equalities. This is not a problem because if we have a solution with $k'_1 + k'_2 < k$ we also have a solution with longer paths. The same is true for $k'_1 + k'_2 < k - 2$. One way to do so is to add the source s enough times so that both paths have the desired segment cost.

($\Leftarrow$) Assume that there exist integers $k_1, k_2 \geqslant 1$, $x \in \mathsf{reach}(k_1, s)$, y such that $s \in \mathsf{reach}(k_2, y)$ and either (1) or (2) holds. Since $x \in \mathsf{reach}(k_1, s)$ there exists a deterministic sr-path $\vec{p} = \langle x_1, \ldots, x_l \rangle$ from s to x with $\mathsf{sr\text{-}cost}(\vec{p}) \leqslant k_1$. In the same way, since $s \in \mathsf{reach}(k_2, y)$, there exists a deterministic sr-path $\vec{q} = \langle y_1, \ldots, y_r \rangle$ from y to s with $\mathsf{sr\text{-}cost}(\vec{q}) \leqslant k_2$.

Case 1: Condition (1) holds so that $k_1 + k_2 = k$, $y \in \mathsf{sp\text{-}reach}(x)$ and $e \in \mathsf{SP}(x, y)$. Let $\vec{c} = \vec{p} + \vec{q} = \langle x_1, \ldots, x_l, y_1, \ldots, y_r \rangle$. This sr-path is deterministic because $y^1_1 = y \in \mathsf{sp\text{-}reach}(x) = \mathsf{sp\text{-}reach}(x^2_l)$. For the other indexes, the unicity of shortest paths comes from the fact that both $\vec{p}$ and $\vec{q}$ are deterministic. Since $\vec{c}$ goes from s to s and has cost $k_1 + k_2 = k$, $\vec{c}$ is a sr-cycle of cost at most k from s to s. It contains e because $e \in \mathsf{SP}(x, y) = \mathsf{SP}(x^2_l, y^1_1)$.

Case 2: Condition (2) holds so $k_1 + k_2 = k - 2$, $e^1 \in \mathsf{sp\text{-}reach}(x)$ and $y \in \mathsf{sp\text{-}reach}(e^2)$. Let $\vec{c} = \langle x_1, \ldots, x_l, e, y_1, \ldots, y_r \rangle$. Since $e^1 \in \mathsf{sp\text{-}reach}(x)$ and $y \in \mathsf{sp\text{-}reach}(e^2)$ we have that $\vec{c}$ is deterministic. As before, for the other indexes determinism comes from the determinism of $\vec{p}$ and $\vec{q}$. Finally, $\mathsf{sr\text{-}cost}(\vec{c}) = \mathsf{sr\text{-}cost}(\vec{p}) + \mathsf{sr\text{-}cost}(\vec{q}) + \mathsf{sr\text{-}cost}(e) = k_1 + k_2 + 2 \leqslant k - 2 + 2 = k$. Clearly $e \in \vec{c}$ so the proof is complete. □

Theorem 7.1 gives necessary and sufficient conditions for the existence of a sr-cycle with segment cost at most k covering a given network edge. This condition is checkable in polynomial time since, if nothing better, we can just loop over all possible candidates $x, y \in V(G)$ and splits of k into k_1, k_2 (recall that $k \leqslant 2|E(G)|$).

This shows that we can solve Problem 1 in polynomial time. For each edge e we compute the smallest k such that there exists a deterministic sr-cycle covering e. Then the maximum value over all these k values will be the minimum segment cost for which a sr-cycle cover is possible.

Algorithm 10 closely follows Theorem 7.1 to build a sr-cycle for a given edge e, source s and segment cost k. On lines 2 to 7 we try to see whether there exists x, y, k_1 and k_2 that satisfy condition (1) from the theorem. If so, we build a cycle accordingly by putting together a deterministic sr-path from s to x of segment cost at most k_1 and a deterministic sr-path from y to s of segment cost at most k_2. Upon failing to find such a cycle, on lines 9 through 15 we try to find x, y, k_1 and k_2 that satisfy condition (2) from the theorem. If we find such values, we build a cycle composed by a deterministic sr-path from s to x of cost at most k_1, followed by an adjacency segment on e and a deterministic sr-path from y to s of cost at most k_2. If we reach line 16 then Theorem 7.1 guarantees that there exists no sr-cycle of segment cost at most k that covers e from source s.

Algorithm 10 build-srcycle (G, s, k, e)

```
1: [look for a cycle that satisfies the first set of conditions from Theorem 7.1]
2: for k_1 ∈ {1, ..., k − 1} do
3:    k_2 ← k − k_1
4:    for x ∈ reach(k_1, s) do
5:       for y ∈ sp-reach(x) do
6:          if s ∈ reach(k_2, y) and e ∈ E(SP(x, y)) then
7:             return build-det-srpath(G, k_1, s, x) ⊕ build-det-srpath(G, k_2, y, s)
8: [look for a cycle that satisfies the second set of conditions from Theorem 7.1]
9: for k_1 ∈ {1, ..., k − 2} do
10:   k_2 ← k − k_1 − 2
11:   for x ∈ reach(k_1, s) do
12:      if e^1 ∈ sp-reach(x) then
13:         for y ∈ sp-reach(e^2) do
14:            if s ∈ reach(k_2, y) then
15:               return build-det-srpath(G, k_1, s, x) ⊕ ⟨e⟩ ⊕ build-det-srpath(G, k_2, y, s)
16: return null
```

Algorithm 11 cover-exists (G, s, k)

```
1: for e ∈ E(G) do
2:    if cycle-exists(G, s, k, e) = null then
3:       return false
4: return true
```

By pre-computing reach and sp-reach as sets with $O(1)$ membership testing, this algorithm runs in $O(k^2 \cdot |V(G)|^2 \cdot |G|)$ since the cost of building a path with Algorithm 14 is $O(k \cdot |G|)$. In practice thought, it will usually run much faster since the reachability sets are often quite smaller than $V(G)$ and a lot of loop iterations are cut-off beforehand by the conditions.

To check whether a cycle cover exists for a given source s and segment cost k we simply loop over all edges $e \in E(G)$ and check whether a cycle exists for each of them using Algorithm 10. Therefore the runtime of this algorithm is $O(k^2 \cdot |V(G)|^2 \cdot |G| \cdot |E(G)|)$. For completeness a formalization of this algorithm is provided as Algorithm 11.

To find k such that a cover exists from source s, we perform a binary search of k to find the smallest k such that a cover exists. This process is described in Algorithm 12. Our initial search interval is $[0, 2|E(G)|]$ because we know from Lemma 4.7 that any path admits a segmentation of cost at most $2|E(G)|$. Once we find this minimum k, we compute a set of sr-cycles that cover all edges. For this we iterate over all edges and compute a sr-cycle covering it with Algorithm 10. We keep a set of covered edges to which we add all edges of every new sr-cycle in the cover. In this way we reduce the total number of cycles in the final solution. This give a total runtime of $O(\log E(G) \cdot k^2 \cdot |V(G)|^2 \cdot |G| \cdot |E(G)|)$. If the source is unknown, we can add an extra iteration over all $v \in V(G)$ to find the one minimizing k as shown in Algorithm 13.

This is quite a high time complexity but it is none the less polynomial since $k \leqslant 2|E(G)|$. Even tough Algorithm 13 runs in a reasonable amount of time in practice as shown on Figure 7.4, there is most likely a lot of space for improving it. We leave finding more efficient algorithms as an open problem on this thesis.

Algorithm 12 min-seg-cover (G, s)

```
1: [perfom a binary search to find the smallest k such that a cover exists]
2: lb ← 0
3: up ← 2|E(G)|
4: while up − lb ⩾ 2 do
5:    k ← (lb+up)/2
6:    if cover-exists (G, s, k) then
7:       up ← k
8:    else
9:       lb ← k
10: [k = up is the smallest segment cost such that a cover exists, build it]
11: covered ← ∅
12: C ← ∅
13: for e ∈ E(G) do
14:    if e ∉ covered then
15:       c⃗ ← build-srcycle (G, s, ub, e)
16:       covered ← covered ∪ E(c⃗)
17:       C ← C ∪ {c⃗}
18: return C, ub
```

Algorithm 13 min-seg-cover (G)

```
1: C*, s*, k* ← null, null, ∞
2: for s ∈ V(G) do
3:    C, k ← min-seg-cover (G, s)
4:    if k < k_min then
5:       C*, s*, k* ← C, s, k
6: return C*, s*, k*
```

Minimum segmentation sr-cycle cover analysis

Figure 7.4 shows that the maximum time taken over any topology to compute a minimum sr-cycle cover with an unknown source was about 15 minutes. This is a reasonable amount of time given that network monitoring covers are often computed only once and only need to change when the topology changes. A 15 minute setup time for a link failure monitoring service seems perfectly usable in practice.

Figure 7.5 shows the runtime of refalgo:min-seg-cover2 with respect to the size of the topology $|G|$. We can observe that it indeed performs much better in practice than its theoretical time complexity.

Next, we analyze the segment cost of the sr-cycle covers produced by the algorithm. This gives a lower bound on the segment cost of *any* cycle cover on the given topologies. This is an important result as it shows whether or not we can expect to be able to implement such a monitoring scheme on a segment routed network. As we can see on Figure 7.6, for more than 70% we can find a cycle cover needing at most a segment cost of 5. This means that in most topologies such a sr-cycle cover is implementable even on a network operating low end routers. Almost all topologies require a segment cost of at most 10 showing that network monitoring with segment routing is realistic for most topologies with high end routers. We wanted to analyze the relationship between the size of the topology and the segment cost required for the sr-cycle

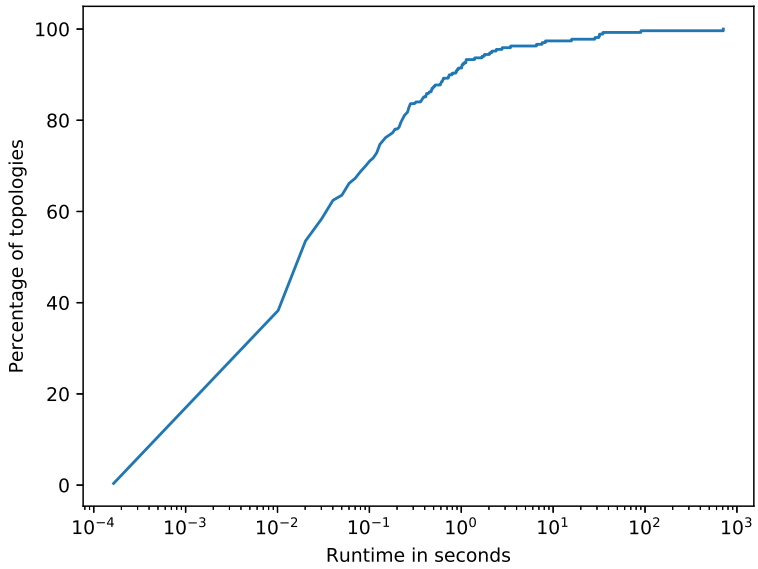


Figure 7.4: Runtime CDF of Algorithm 13 over all topologies.

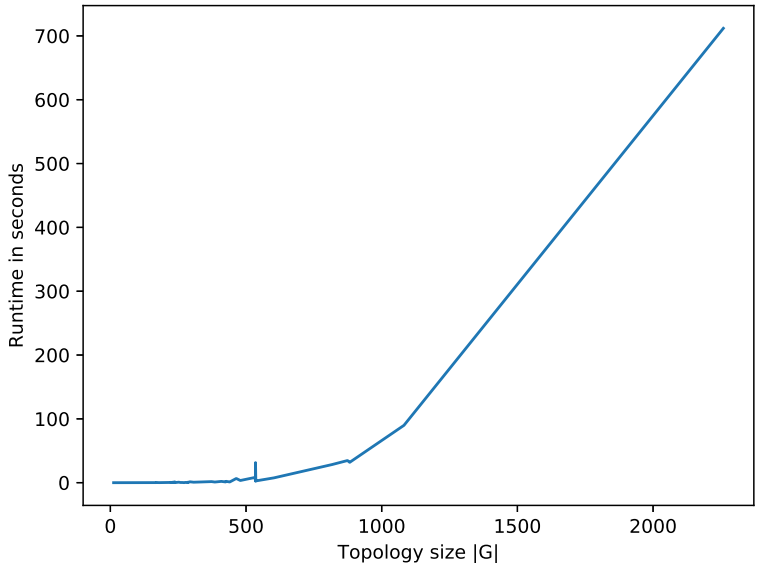


Figure 7.5: Runtime by topology size of Algorithm 13 over all topologies.

Algorithm 14 build-det-srpath (G, k, s, t)

```
1: if k = 1 then
2:    return ⟨s⟩
3: if k = 2 then
4:    if t ∈ N⁺(G, s) then
5:       return ⟨(s, t)⟩    [if multiple edges exist between s and t, any will do]
6:    return ⟨s, t⟩
7: for v ∈ reach(k − 1, s) do
8:    if t ∈ sp-reach(v) then
9:       return ⟨t⟩ ⊕ build-det-srpath (G, k − 1, s, v)
10: for v ∈ reach(k − 2, s) do
11:    for e ∈ δ⁻(G, t) do
12:       if e¹ ∈ sp-reach(v) then
13:          return ⟨e⟩ ⊕ build-det-srpath (G, k − 2, s, v)
14: return null
```

cover. The is shown in Figure 7.7. We can see that the size does not seem to be correlated with the minimum segment cost.

7.2 Column generation cycle cover algorithm

We saw in Chapter 6 that column generation seemed to be a good framework to solve segment routing problems. In this section we propose a column generation algorithm for computing lower bounds on the number of sr-cycles in a minimum sr-cycle cover of a network. We also show how to derive from it an heuristic algorithm for Problem 2.

It is straightforward to express a MIP formulation for Problem 2. We define binary variables $x_{\vec{c}}$ such that $x_{\vec{c}} = 1$ if and only if $\vec{c}$ is used in the cover. These variables will be defined for every $\vec{c} \in \mathcal{C}_s^k$ where k and s are given as input.

In terms of objective function, since we want to minimize the number of cycles in the cover, we can achieve this by minimizing $\sum_{\vec{c} \in \mathcal{C}_s^k} x_{\vec{c}}$. This is so because this sum counts how many cycles are used in the solution. For the constraints it is quite simple as well. We simply need to ensure that for each edge $e \in E(G)$, there is at least one sr-cycle covering it, that is, there is at least one $\vec{c}$ such that $x_{\vec{c}} = 1$ and $e \in E(\vec{c})$. We define the following identify function to simplify the expression of this constraints.

Definition 7.3. *Given a network* G *we define a function* $I : \mathcal{P} \times E(G) \to \{0, 1\}$ *such that* $I(\vec{p}, e) = 1$ *if and only if* $e \in E(\vec{p})$.

$$
\begin{aligned}
&\mathsf{SRCC}(G, k, s) \\
&\textbf{min} \quad \sum_{\vec{c} \in \mathcal{C}_s^k} x_{\vec{c}} \\
&\textbf{s.t.} \quad \sum_{\vec{c} \in \mathcal{C}_s^k} I(\vec{c}, e) \cdot x_{\vec{c}} \quad \geqslant \quad 1 \qquad \forall e \in E(G) \\
&\qquad\qquad x_{\vec{c}} \quad \in \quad \{0, 1\} \quad \forall \vec{c} \in \mathcal{C}_s^k
\end{aligned}
$$

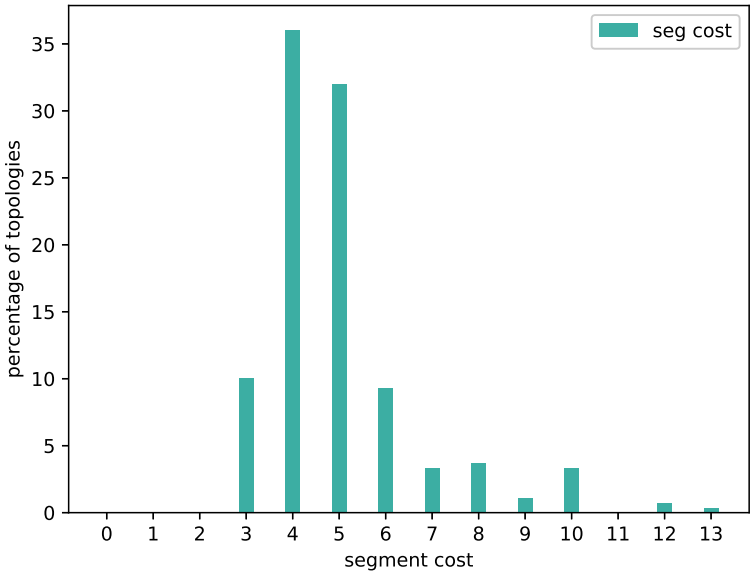


Figure 7.6: Percentage of topologies for each segment cost.

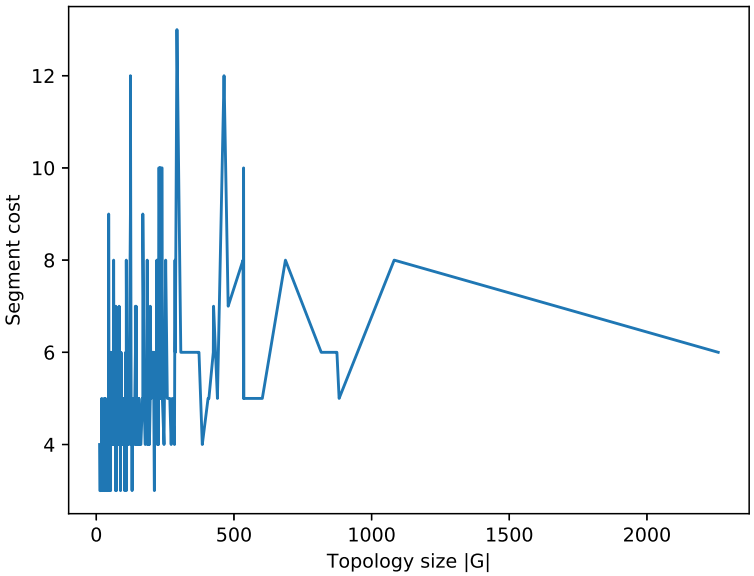


Figure 7.7: Segment cost by topology size over all topologies.

7.2.1 Column generation

We follow the same process that we did in Chapter 6 to develop the column generation algorithm for the minimum sr-cycle cover problem. Recall that CG is a technique for solving a LP and we have a MIP. The first step is then to consider the LP-relaxation of SRCC and compute its dual.

$$\begin{aligned}
&\mathsf{SRCC\text{-}LP}(G, k, s)\\
&\textbf{min} \quad \sum_{\vec{c}\in\mathcal{C}_s^k} x_{\vec{c}}\\
&\textbf{s.t.} \quad \sum_{\vec{c}\in\mathcal{C}_s^k} I(\vec{c}, e)\cdot x_{\vec{c}} \geqslant 1 \quad \forall e \in E(G) \qquad \text{(P1)}\\
&\qquad\qquad x_{\vec{c}} \geqslant 0 \quad \forall \vec{c} \in \mathcal{C}_s^k
\end{aligned}$$

As before, the most natural relaxation would be to set $x_{\vec{c}} \in [0, 1]$ but our objective function guarantees that this is equivalent to $x_{\vec{c}} \geqslant 0$. We chose the second one because it is equivalent and easier to work with. The dual is obtained by following the same systematic procedure as before. By doing so we get the following LP.

$$\begin{aligned}
&\mathsf{SRCC\text{-}DUAL}(G, k, s)\\
&\textbf{max} \quad \sum_{e\in E(G)} y_e\\
&\textbf{s.t.} \quad \sum_{e\in E(G)} I(\vec{c}, e)\cdot y_e \leqslant 1 \quad \forall \vec{c} \in \mathcal{C}_s^k \qquad \text{(D1)}\\
&\qquad\qquad y_e \geqslant 0 \quad \forall e \in E(G)
\end{aligned}$$

The idea to solve SRCC-LP is again to start with a small but feasible subset of sr-cycles $C \subseteq \mathcal{C}_s^k$ and slowly grow it until we can prove optimality. We denote the problems restricted to C by adding C as an argument, that is, by $\mathsf{SRCC\text{-}LP}(G, k, s, C)$ and $\mathsf{SRCC\text{-}DUAL}(G, k, s, C)$. To find a new element $\vec{c}$ to add to C we need to, given an optimal solution y^* of $\mathsf{SRCC\text{-}DUAL}(G, k, s, C)$, find a sr-cycle $\vec{c}$ such that

$$\sum_{e\in E(G)} I(\vec{c}, e)\cdot y_e^* = \sum_{e\in E(\vec{c})} y_e^* > 1.$$

Solving this directly is NP-hard since it can be shown to be equivalent to the longest path problem. Instead of trying to solve this pricing problem directly, we are going to see that by slightly changing the LP formulation we can obtain a polynomial time solvable pricing problem.

Definition 7.4. *Given a network G we define a function $K : \mathcal{P} \times E(G) \to \mathbb{N}$ such that $K(\vec{p}, e)$ equals the number of times e is traversed by $\vec{p}$. Formally, if $\vec{p} = \langle x_1, \ldots, x_l\rangle$*

$$K(\vec{p}, e) = \sum_{i=2}^{l} I(\mathsf{SP}(x_i^2, x_{i-1}^1), e) + \sum_{i=1:x_i=e}^{l} 1.$$

By replacing I by K in the above formulations we still get a model for Problem 2 since, the new constraints corresponding to (P1) with I replaced by K, will still be true if and only if every edge is covered by at least one cycle. With this change the formulations become:

$\mathsf{SRCC\text{-}LP\text{-}K}(G, k, s)$

$$\begin{aligned} \textbf{min} \quad & \sum_{\vec{c} \in \mathcal{C}_s^k} x_{\vec{c}} \\ \textbf{s.t.} \quad & \sum_{\vec{c} \in \mathcal{C}_s^k} K(\vec{c}, e) \cdot x_{\vec{c}} \geqslant 1 \quad \forall e \in E(G) \\ & x_{\vec{c}} \geqslant 0 \quad \forall \vec{c} \in \mathcal{C}_s^k \end{aligned}$$

$\mathsf{SRCC\text{-}DUAL\text{-}K}(G, k, s)$

$$\begin{aligned} \textbf{max} \quad & \sum_{e \in E(G)} y_e \\ \textbf{s.t.} \quad & \sum_{e \in E(G)} K(\vec{c}, e) \cdot y_e \leqslant 1 \quad \forall \vec{c} \in \mathcal{C}_s^k \\ & y_e \geqslant 0 \quad \forall e \in E(G) \end{aligned}$$

The pricing problem now becomes the following one.

Problem 3 (SRCC pricing)

Input: *A network* G, $k \in \mathbb{N}$, *a source node* $s \in V(G)$ *and values* $y_e \geqslant 0$ *for each* $e \in E(G)$.
Output: *A sr-cycle* $\vec{c} \in \mathcal{C}_s^k$ *such that*

$$\sum_{e \in E(G)} K(\vec{c}, e) \cdot y_e > 1$$

or report that no such sr-cycle exists.

In order to solve Problem 3, we define a sr-metric w such that

$$w(u, v) = \sum_{e \in SP(u,v)} y_e$$

and

$$w(e) = y_e.$$

With this metric, it is easy to see can see that

$$w(\vec{c}) = \sum_{e \in E(G)} K(\vec{c}, e) \cdot y_e.$$

Therefore we can solve the pricing problem by computing a sr-cycle $\vec{c}$ from s to s of segment cost at most k such that $w(\vec{c})$ is maximum. We have already shown in Chapter 5 that this problem can be solved in polynomial time. Therefore, we have the following theorem.

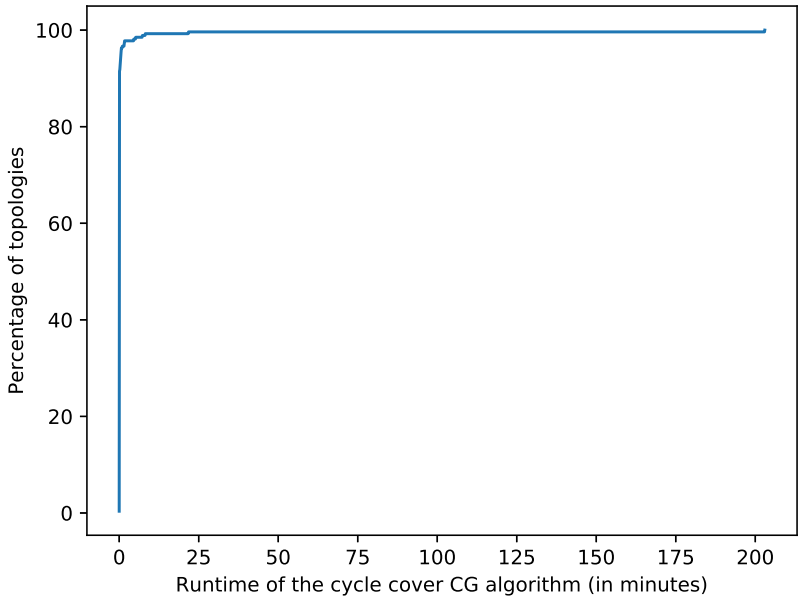


Figure 7.8: CDF of the runtime of the column generation cycle cover algorithm.

Theorem 7.2. *Problem 3 can be solved in polynomial time.*

To start the column generation algorithm we need a feasible solution. In order to obtain one, we run Algorithm 13 which also gives us suitable source node s and lower bound on the segment cost k. If the target segment cost is lower than k then we immediately know that a feasible solution does not exist. If it does, we use the cycles produced by the minimum segment cost cycle cover algorithm as an initial feasible solution for the column generation algorithm.

We evaluated the runtime of the column generation algorithm over all topologies. Figure 7.8 shows a CDF of these runtimes. The slowest topology took about 3 hours so solve. Most topologies are solved in a very short amount of time and 98% of the topologies are solved in under 25 minutes. These values are very reasonable given that the probing cycle only needs to be computed when the topology changes.

In the next section we will evaluate the lower bound provided by the column generation algorithm.

7.2.2 Greedy algorithm

We designed above a column generation algorithm for solving the LP relaxation of Problem 2. In this relaxation, an edge can be covered fractionally by two sr-cycles or more. In this section we propose a greedy algorithm for converting these fractional solutions to proper solutions of Problem 2.

Let C be the final set of sr-cycles. Our algorithm simply consists of selecting elements of C while keeping track of the edges that are already covered until all of them are. At each step, the sr-cycle that we selected is the sr-cycle that

Algorithm 15 greedy-cc (G, C)

```
1: covered ← ∅
2: cover ← ∅
3: while |U| < |E(G)| do
4:     c⃗ ← c⃗ ∈ C such that |E(c⃗) ∪ covered| is maximum
5:     cover ← cover ∪ {c⃗}
6:     covered ← covered ∪ E(c⃗)
7: return cover
```

covers the maximum remaining uncovered edges.

It is a well know result that this yields a logarithmic factor approximation. We prove this in the next theorem for completeness.

Theorem 7.3. *The set of cycles,* cover*, produced by Algorithm 15 is such that*

$$\mathsf{opt}_C \leqslant |\mathsf{cover}| \leqslant \log |E(G)| \cdot \mathsf{opt}_C$$

where opt_C *is the size of a minimum over using only cycles from* C*.*

Proof. Clearly, $\mathsf{opt}_C \leqslant |\mathsf{cover}|$. Let $m = |E(G)|$. Since the optimal solution relative to C uses opt_C sr-cycles, there must be at least one of them that covers at least a fraction $1/\mathsf{opt}_C$ of the edges. If they were all below this ratio, they could not cover all edges. Since we select the sr-cycle that covers the most uncovered edges, the first sr-cycle will cover at least $1/\mathsf{opt}_C$ edges. Hence, after the first iteration, there are at most $m(1 - 1/\mathsf{opt}_C)$ edges left to cover. In the same way, there must be a set that covers at least $1/\mathsf{opt}_C$ of these $m(1 - 1/\mathsf{opt}_C)$ edges. Since we select the sr-cycle covering the most edges, after the second iteration there are at most $m(1 - /\mathsf{opt}_C)^2$ edges left. By repeating this argument, we see that after k iterations there are at most $m(1-1/\mathsf{opt}_C)^k$ edges left. Therefore, after $k = \mathsf{opt}_C \log m$ iterations, there are

$$m(1 - 1/\mathsf{opt}_C)^{\mathsf{opt}_C \log m} = m(1/e)^{\log m} = m \cdot \frac{1}{e^{\log m}} = \frac{m}{m} = 1$$

edges left. Hence $|\mathsf{cover}| \leqslant \log m \cdot \mathsf{opt}_C$. □

Ideally we would want to be able to prove that

$$\mathsf{opt} \leqslant |\mathsf{cover}| \leqslant \log |E(G)| \cdot \mathsf{opt}$$

where opt is the optimal solution over *all* sr-cycles $\mathcal{C}_s^k$ but unfortunately this is not true in general. However, since the fractional solution obtained with the column generation is a lower bound on opt we can still evaluate how far this gets us from the opt in practice. Figure 7.9 shows the sizes of the minimum segment cost covers, the LP lower bound and the sizes of the greedy covers.

We can see that the greedy solution is actually quite close to the lower bound and therefore it must also be close to opt. To provide a better view of how close they are we computed a CDF of relative distance

$$\frac{\mathsf{greedy} - \mathsf{lb}}{\mathsf{lb}}$$

which is shown in Figure 7.10. As a reference, with this metric, a value of 1 means that the greedy solution uses twice the number of sr-cycle compared to

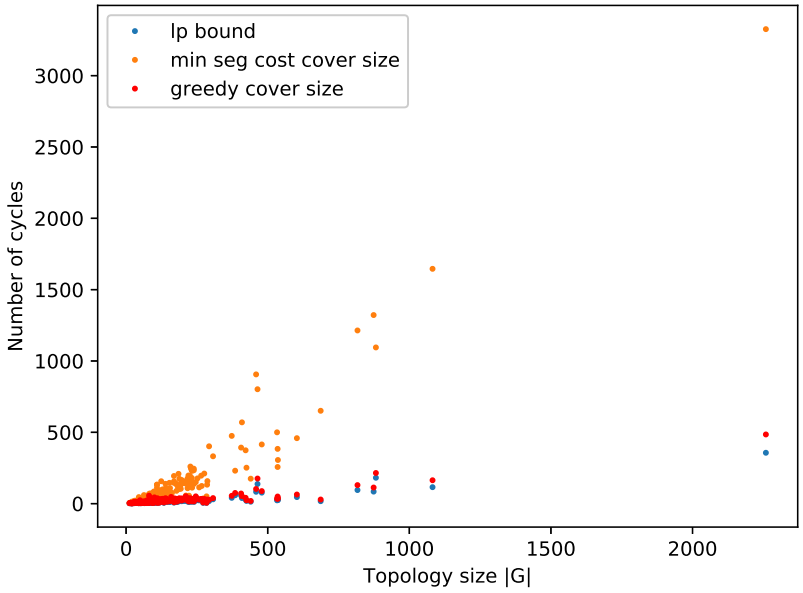


Figure 7.9: Lower bound, min seg cover size and greedy cover size shown by topologies size.

the lower bound. We can see from the CDF that for 90% of the instances we have an increase of at most 50% on the number of sr-cycles. Recall that this lower bound is not the actual number of sr-cycles in the optimal solution but only an estimate. This means that our solution is actually even closer to the optimal solution.

The most important aspect is that by combining the minimum segment cost sr-cycle covers with the column generation and the greedy algorithm we are able to greatly reduce the number of sr-cycles in the minimum segment cost cover. Therefore, our solution is able to find sr-cycle covers that not only use the minimum amount of segments required for *any* sr-cycle cover but also are quite close to the minimum theoretical number of sr-cycles.

7.3 Pinpointing single-link failures

In this section we explain how to use a sr-cycle cover to detect single-link failures.

As mentioned in the introduction of this chapter, the idea is to have node s regularly send monitoring probes over the sr-cycles of in a sr-cycle cover $C \subseteq \vec{\mathcal{C}}_k^s$. Since we are using sr-cycles, if the network is operating without failures, each probe must eventually come back to the vantage point s. If at least one such probe does not come back, we know that at least one of the edges in the cycle associated with it has a failure. We refer to the sr-cycles in the cycle cover C as *probing cycles*.

Let $\vec{c}$ be a cycle with a failure, that is, whose probe did not return. Because we use deterministic cycles, if we map $\vec{c}$ back to G, we get a cycle

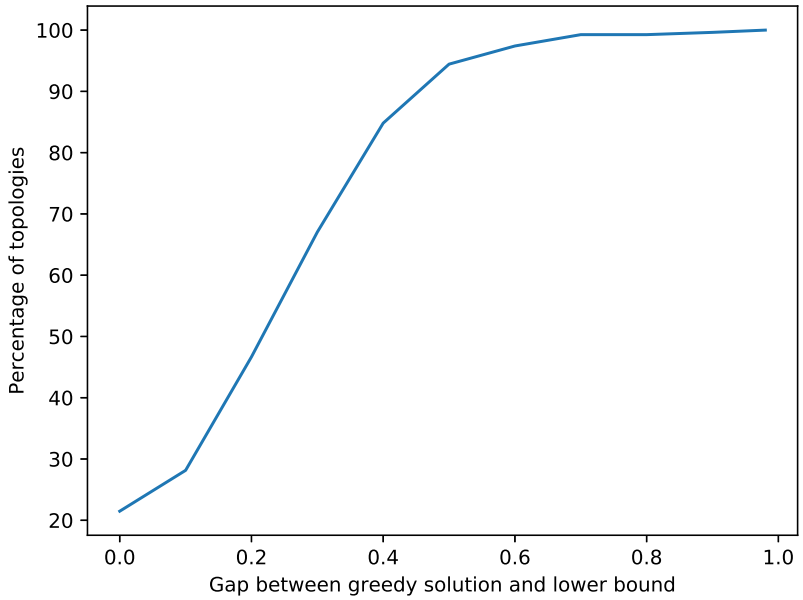
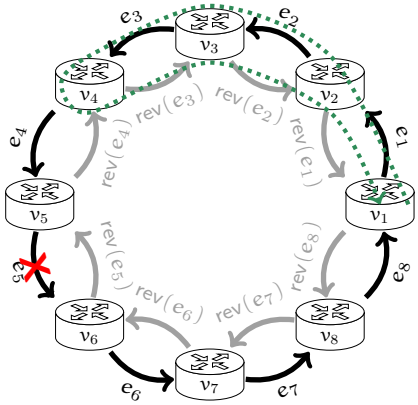


Figure 7.10: CDF of the gap between the greedy solution and the LP lower bound.

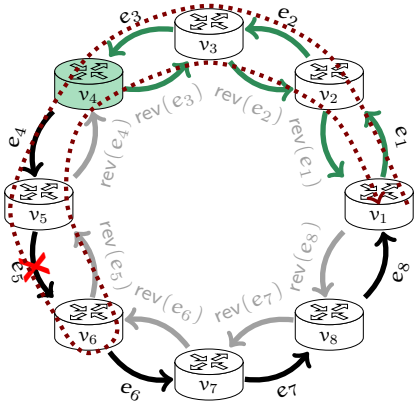
$c = (e_1, e_2, \ldots, e_l)$. Recall that in this chapter we assume that the network G is symmetric so that for each edge e there is a reverse edge $\mathsf{rev}(e)$ and that igp is symmetric if $\mathsf{igp}(e) = \mathsf{rev}(\mathsf{igp}(e))$. The idea to detect the failure is to perform a binary search to find the largest i such that the cycle $c_i = (e_1, \ldots, e_i, \mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_1))$ contains no failure. For each i we send another probe over c_i and check whether it comes back. If it does, then we know that the index we seek is greater than or equal to i. If it does not then the index must be strictly smaller than i. The single-link failure assumption is important for this process to work. Because the probe did not cycle back, we know that one of $e_1, \ldots, e_l$ is faulty. Therefore, since we assume single-link failure, we know that each reverse edge is not faulty. Hence, when we send a binary search probe on c_i, if it does not come back, we know that the problem is in one of $e_1, \ldots, e_i$. We refer to these sr-cycles as *identification cycles*.

Figure 7.11 illustrates this on an example with $l = 8$ and the failure is on edge e_5. We start the search with $i = 4$. The green dotted path illustrates that the probe successfully returned to v_1. Thus we know that edges e_1, e_2 and e_3 are up. The search will select $i = 6$ and this time the probe did not return. We conclude that the error is either in e_4 or e_5. Finally, we send a probe on c_5 which returns. We finally conclude that the problem is on link e_5.

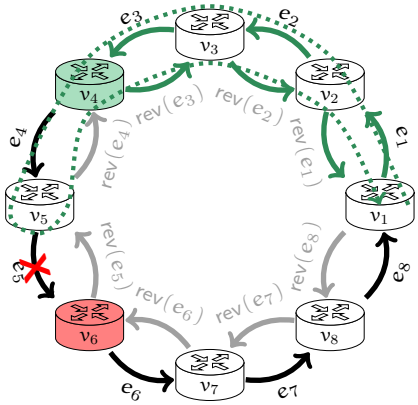
To illustrate why the single link-failure assumption is important, let's see happens with this process when multiple failures occur. Imagine the same example as in Figure 7.11 but assume that edge $\mathsf{rev}(e_3)$ is also down. In this case the first binary search probe will fail to return and the search will stop at $i = 3$ and the algorithm will say that the problem is on edge e_3. However e_3 is up and it is $\mathsf{rev}(e_3)$ that prevents the probe to return. This shows that with multiple



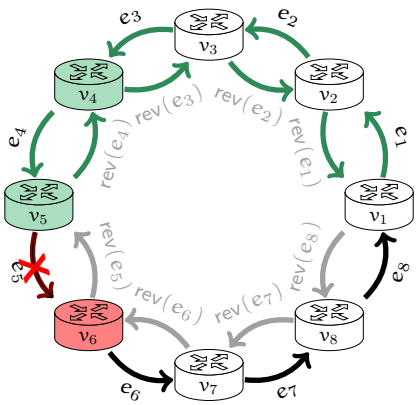
First step of binary search. Success.
Edges e_1, e_2, e_3 are up.



Second step of binary search. Failure.
One of e_4, e_5 is down.



Third step of binary search. Success.
Edges e_4 is up.



End of the search.
Failure detected in e_5.

Figure 7.11: Binary search to identify the failure (single-link failure assumption).

failures, we cannot know exactly which link is down. However, we still can say that either e_3 is down *or* $\mathsf{rev}(e_3)$ is down. Figure 7.12 illustrates this.

This means that in presence of multiple link failures, we cannot tell exactly which link is down but we can still find a set of two links $\{e, \mathsf{rev}(e)\}$ such that we are sure that at least one of them is down.

Computing identification cycles

At each step of the binary search, given i we need to find a way to route a probe over the cycle $c_i = (e_1, \ldots, e_{i-1}, e_i, \mathsf{rev}(e_i), \mathsf{rev}(e_{i-1}), \ldots, \mathsf{rev}(e_1))$. One way do so is to compute a minimum segmentation of c_i and use that segmentation.

There is an alternative solution that avoids having to compute segmentations and uses the segments in $\vec{c} = \langle x_1, \ldots, x_l \rangle$ instead to segment the identification cycles. We will do so by finding an index i such that e_i is between x_j and x_{j+1} and then reverse the elements $x_1, \ldots, x_j$ to obtain a segmentation of $c_i = (e_1, \ldots, e_{i-1}, e_i, \mathsf{rev}(e_i), \mathsf{rev}(e_{i-1}), \ldots, \mathsf{rev}(e_1))$.

Definition 7.5. *Let* G *be a symmetric network and* $\vec{p}$ *be a sr-path on* G. *We define* $\mathsf{rev}(\vec{p}) = \langle \mathsf{rev}(x_l) \ldots, \mathsf{rev}(x_1) \rangle$ *where* $\mathsf{rev}(x_i) = x_i$ *if* $x_i \in V(G)$ *and* $\mathsf{rev}(x_i) = \mathsf{rev}(e)$ *if* $x_i = e \in E(G)$.

Note that we have $\mathsf{rev}(x_i)^1 = x_i^2$ *and* $\mathsf{rev}(x_i)^2 = x_i^1$.

Lemma 7.4. *Let* G *be a symmetric network. Let* $\vec{p}$ *be a deterministic sr-path with* $\mathsf{path}(\vec{p}) = (e_1, \ldots, e_n)$. *If* igp *is symmetric then* $\mathsf{rev}(\vec{p})$ *is a deterministic sr-path with* $\mathsf{path}(\mathsf{rev}(\vec{p})) = (\mathsf{rev}(e_n), \ldots, \mathsf{rev}(e_1)) = \mathsf{rev}(\mathsf{path}(\vec{p}))$.

Proof. We first prove that $\mathsf{rev}(\vec{p})$ is deterministic. Let $i \in \{l, \ldots, 2\}$. We need to prove that there is a unique shortest path from $\mathsf{rev}(x_i)^2 = x_i^1$ to $\mathsf{rev}(x_{i-1})^1 = x_{i-1}^2$. Since $\vec{p}$ is deterministic, we know that there is a unique shortest path from x_{i-1}^2 to x_i^1. By Corollary 2.6 we know then that there is also a unique shortest path from x_i^1 to x_{i-1}^2.

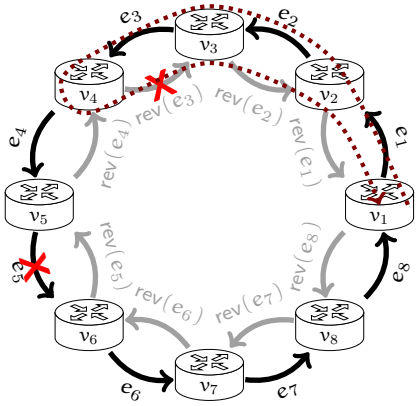
Since $\mathsf{path}(\vec{p}) = (e_1, \ldots, e_n)$, we have that for each i there exist $1 \leqslant k_1^i \leqslant k_2^i \leqslant n$ such that $\mathsf{SP}(x_i^2, x_{i+1}^1) = (e_{k_1^i}, \ldots, e_{k_2^i})$. Therefore, by Corollary 2.6, $\mathsf{SP}(\mathsf{rev}(x_{i+1})^2, \mathsf{rev}(x_i)^1) = \mathsf{SP}(x_{i+1}^1, x_i^2) = (\mathsf{rev}(e_{k_2^i}), \ldots, \mathsf{rev}(e_{k_1^i}))$. Since for adjacency segments $x_i = e$ we have $\mathsf{rev}(x_i) = \mathsf{rev}(e)$ we conclude that $\mathsf{path}(\mathsf{rev}(\vec{p})) = (\mathsf{rev}(e_n), \ldots, \mathsf{rev}(e_1))$. □

To build a segmentation of c_i from $\vec{c}$ we find an index j such that either $e_i = x_j$ or $e_i \in \mathsf{SP}(x_{j-1}^2, x_j^1)$. This index always exists since $\vec{c}$ is a deterministic sr-cycle that traverses edge e_i. Depending on which case occurs, we can build the sr-cycle $\vec{c}_i$ covering c_i as follows:

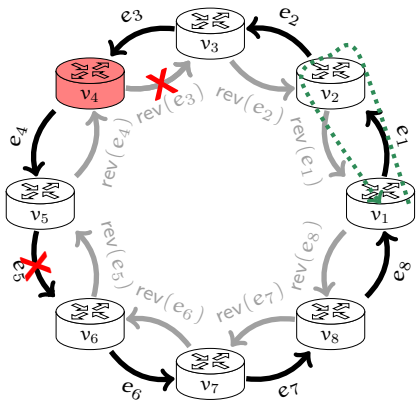
Case 1: $e_i = x_j$. In this case, we can use

$$\begin{aligned}\vec{c}_i &= \langle x_1, \ldots, x_j \rangle \oplus \mathsf{rev}(\langle x_1, \ldots, x_j \rangle) \\ &= \langle x_1, \ldots, x_j \rangle \oplus \langle \mathsf{rev}(x_j), \ldots, \mathsf{rev}(x_1) \rangle \\ &= \langle x_1, \ldots, x_j, \mathsf{rev}(x_j), \ldots, \mathsf{rev}(x_1) \rangle\end{aligned}$$

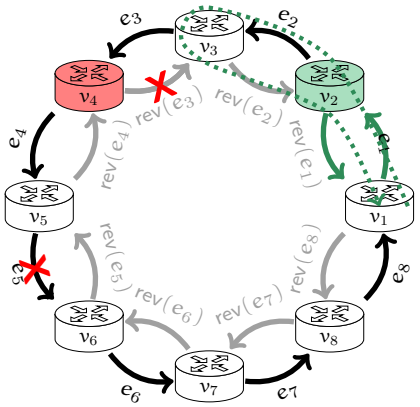
To see that $\vec{c}_i$ is a segmentation of c_i we obeserve that since $\mathsf{path}(\langle x_1, \ldots, x_j \rangle) = (e_1, \ldots, e_i)$, by Lemma 7.4, it holds that $\mathsf{path}(\langle x_1, \ldots, x_j \rangle) = (\mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_1))$



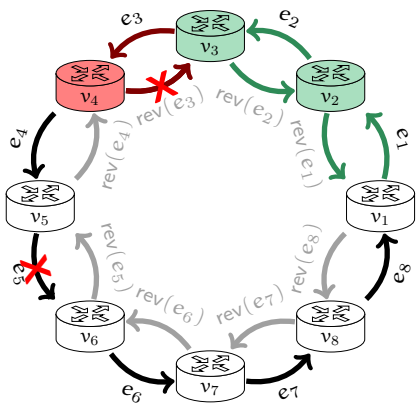
First step of binary search. Failure.
One of $e_1, e_2, e_3, \mathsf{rev}(e_3), \mathsf{rev}(e_2), \mathsf{rev}(e_1)$ is down.



Second step of binary search. Success.
Edges $e_1, \mathsf{rev}(e_1)$ are up.



Third step of binary search. Success.
Edges $e_2, \mathsf{rev}(e_2)$ are up.



End of the search.
Failure in e_3 or $\mathsf{rev}(e_3)$.

Figure 7.12: Binary search to identify the failure. No single-link failure assumption.

and so

$$\begin{aligned}\mathsf{path}(\vec{c}_i) &= \mathsf{path}(\langle x_1, \ldots, x_j\rangle) \oplus \mathsf{path}(\mathsf{rev}\langle x_1, \ldots, x_j\rangle) \\ &= (e_1, \ldots, e_i) \oplus (\mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_1)) \\ &= (e_1, \ldots, e_i, \mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_1)) = c_i\end{aligned}$$

Case 2: $x_j \in \mathsf{SP}(x^2_{j-1}, x^1_j)$. In this case, we can use the sr-path

$$\vec{c}_i = \langle x_1, \ldots x_{i-1}\rangle \oplus e^2_j \oplus \mathsf{rev}(\langle x_1, \ldots, x_{i-1}\rangle)$$

Since $\vec{c}$ is a segmentation of c, there must exist some index $k < i$ such that $\mathsf{path}(\langle x_1, \ldots, x_{j-1}\rangle) = (e_1, \ldots, e_k)$ and $\mathsf{path}(\langle x_{j-1}, \ldots, x_l\rangle) = (e_{k+1}, \ldots, e_i, \ldots, e_n)$. With this notation, $\mathsf{SP}(x^2_{j-1}, e^2_i) = (e_{k+1}, \ldots, e_i)$ so

$$\begin{aligned}\mathsf{path}(\vec{c}_i) &= \mathsf{path}(\langle x_1, \ldots, x_{j-1}\rangle) \oplus \mathsf{SP}(x^2_{j-1}, e^2_i) \oplus \\ &\quad \mathsf{rev}(\mathsf{SP}(x^2_{j-1}, e^2_i)) \oplus \mathsf{path}(\mathsf{rev}(\langle x_1, \ldots, x_{i-1}\rangle)) \\ &= (e_1, \ldots, e_k) \oplus (e_{k+1}, \ldots, e_i) \oplus \\ &\quad \mathsf{rev}((e_{k+1}, \ldots, e_i)) \oplus \mathsf{rev}(\mathsf{path}(\langle x_1, \ldots, x_{i-1}\rangle)) \\ &= (e_1, \ldots, e_k) \oplus (e_{k+1}, \ldots, e_i) \oplus \\ &\quad (\mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_{k+1})) \oplus \mathsf{rev}(e_1, \ldots, e_k) \\ &= (e_1, \ldots, e_k) \oplus (e_{k+1}, \ldots, e_i) \oplus \\ &\quad (\mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_{k+1})) \oplus (\mathsf{rev}(e_k), \ldots, \mathsf{rev}(e_1)) \\ &= (e_1, \ldots e_i) \ \oplus (\mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_1)) \\ &= (e_1, \ldots, e_i, \mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_1)) = c_i\end{aligned}$$

In both cases, we see that $\vec{c}_i$ is a segmentation of cycle c_i. One drawback of this approach is that the probing cycle used during the binary search can have up to a double segment cost of the probing cycles in the cycle cover. This is something that we will have to consider when selecting the maximum segment cost of the cycles in the cycle cover. On a network with high-end routers we can use sr-paths with up to about 10 segments in their segment stack. This means that we should compute probing cycles with up to 5 segments if we want to be sure that the search cycles are supported.

Algorithm 16 formalizes this process. We executed Algorithm 13 on all topologies of our dataset and then for each sr-cycle we computed the maximum segment cost over all identification cycles that could be used in Algorithm 16. Figure 7.13 shows the distribution of these costs. We mentioned in the introduction that high end routers tend to support up to about 10 segments. This figure shows that for about 20% of the topologies identification of sr-cycles sometimes require more than 10 segments. In order to overcome this problem, in the next section we discuss an alternative monitoring scheme which uses a different set of IGP weights for the probing and identification sr-cycles. These new weights are designed with the intent of reducing the maximum segment cost required to identify network failures.

7.4 Dual topology monitoring

As we showed before, the minimum number of segments required to cover a topology can be quite high. One idea to reduce it is to have a separate set

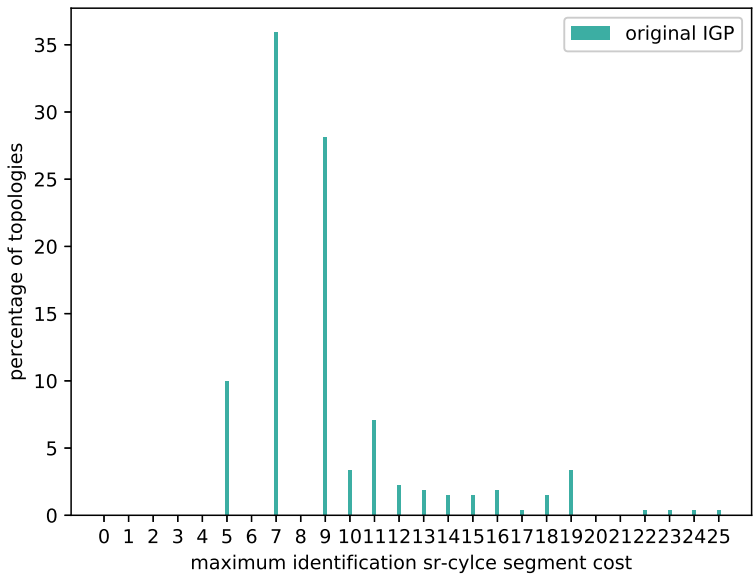


Figure 7.13: Distribution of the maximum segment cost of the probing sr-cycles.

of IGP weights. The ones already configured on the network used to forward traffic and new ones that are used only for sending the monitoring probes.

One idea to compute those weights would be to first compute a minimum cycle cover of the network using some standard existing algorithm [19] (note that here we are talking about cycles, not sr-cycles). Then we could compute a set of weights such that the maximum number of segments required to segment any of those cycles is as small as possible. We did not solve this problem and therefore leave it as an open problem (with a slightly more general formulation).

Problem 4 (Optimal segmentation IGP)

Given a graph G *and a set* P *of paths on* G *compute a IGP weight function* $\mathsf{igp} : \mathbb{E}(\mathsf{G}) \to \mathbb{N}$ *such that if we compute a minimal segmentation of every path* $\mathsf{p} \in \mathsf{P}$ *whose maximum segment cost amongst those sr-paths is minimal.*

We believe that solving this problem could be very useful in any setting where having a dual weight topology is infeasible in practice. This would be yet another way to leverage existing graph theory to solve the problem and then translate those graph theoretic solutions into segment routing solutions with low segment cost.

Another possibility, which we explore in this thesis, is to compute a set of igp weights such that there is a unique shortest path between any pair of nodes *and* every edge belongs to a unique shortest path. The intuition of why this will make segmentations less costly is that the conditions for needing to add a new segment in the minimum segmentation algorithm is exactly the existence of multiple shortest paths or an edge that does not belong to any shortest path.

Algorithm 16 find-faulty-edge $(g, \vec{c} = \langle x_1, \ldots, x_l \rangle)$

$c = (e_1, \ldots, e_n) \leftarrow \mathsf{path}(\vec{c})$
$L \leftarrow 0, R \leftarrow n$
while $R - L \geqslant 2$ **do**
  $i \leftarrow \frac{L+R}{2}$
  if g.isSymmetric() **then**
    $j \leftarrow \min \left\{ j \in \{1, \ldots, l\} \mid e_i = x_j \vee \left(j < l \wedge e_i \in \mathsf{SP}(x_j^2, x_{j+1}^1) \right) \right\}$
    if $e_i = x_j$ **then**
      $\vec{c}_i \leftarrow \langle x_1, \ldots, x_j, \mathsf{rev}(x_j), \ldots, \mathsf{rev}(x_1) \rangle$
    else
      $\vec{c}_i \leftarrow \langle x_1, \ldots, x_{j-1}, e_i^2, \mathsf{rev}(x_{j-1}), \ldots, \mathsf{rev}(x_1) \rangle$
  else
    $\vec{c}_i \leftarrow \mathsf{min\text{-}segmentation}((e_1, \ldots, e_i, \mathsf{rev}(e_i), \ldots, \mathsf{rev}(e_1))$
  status $\leftarrow \mathsf{send\text{-}probe}(\vec{c}_i)$
  if *status* = *received* **then**
    $L \leftarrow M$
  else
    $R \leftarrow M$
if *single-link failures* **then**
  return e_L
return $\{e_L, \mathsf{rev}(e_L)\}$

Note that our two conditions cannot both coexist in a network with parallel links. With two links between u and v, we cannot have at the same time that both those links belong to some shortest path and a unique shortest path between u and v. We therefore relax the definition as follows.

Definition 7.6. *Let* G *be a graph. A set of IGP weights* $\mathsf{igp} : E(G) \to \mathbb{N}$ *is said to be* complete *if and only if for all* $u, v \in V(G)$ *at least one edge of* $E(G, u, v)$ *belong to a shortest path.*

Definition 7.7. *Let* G *be a graph. A set of IGP weights* $\mathsf{igp} : E(G) \to \mathbb{N}$ *is said to be* ECMP-free *if and only if for all* $u, v \in V(G)$ *there is a unique shortest path between* u *and* v*.*

7.4.1 Computing ECMP-free and complete IGP weights

Definition 7.8. *Let* G *be a graph. A set of IGP weights* $\mathsf{igp} : E(G) \to \mathbb{N}$ *is said to be* total *if and only all simple paths on* G *have a different weight.*

Computing a total weighting of a graph G is trivial. We can simply set $\mathsf{igp}(e) = 2^{\mathsf{idx}(e)}$ for all $e \in E(G)$. Since $\mathsf{idx}(e)$ assigns a unique index between 0 and $|E(G)| - 1$ to the edges of G, this IGP function will assign a different power of two to each edge of G. Therefore, any two distinct simple paths must have a different IGP weight since those weights will correspond to sums of distinct powers of 2.

The following lemma shows, unsurprisingly, that one way to build ECMP-free weights is to compute total weights.

Lemma 7.5. *Let* G *be a graph. If* $\mathsf{igp} : E(G) \to \mathbb{N}$ *is total then it is ECMP-free.*

Proof. Trivial from the definition. Any shortest path is simple and thus any two shortest paths must have a different IGP weight since igp is total. □

The following result shows that we can transform any total IGP weighting igp into a complete one by adding a large enough constant. This constant can by any value above the maximum between the diameter of the graph with respect to igp and the maximum weight of any edge. Being larger than the diameter ensures that shortest paths remain unique and being larger than the maximum weight ensures that every edge belong to a shortest path. The diameter of a network is the greatest distance (in terms of number of edges) between any pair of vertices.

Lemma 7.6. *Let* G *be a network. Let*

$$M \geqslant \max\left(\mathtt{diam}(\mathsf{G}, \mathsf{igp}), \max_{e \in \mathsf{E}(\mathsf{G})} \mathsf{igp}(e)\right).$$

Then, if $\mathsf{igp} : \mathsf{E}(\mathsf{G}) \to \mathbb{N}$ *is total then* $\mathsf{igp}^+ : \mathsf{E}(\mathsf{G}) \to \mathbb{N}$ *defined by* $\mathsf{igp}^+(e) = \mathsf{igp}(e) + M$ *is complete.*

Proof. Let $p_1 = (e_1, \ldots, e_n)$ and $p_2 = (f_1, \ldots, f_m)$ be two paths on G.

We start by proving that $\mathsf{igp}^+(p_1) \neq \mathsf{igp}^+(p_2)$. By definition $\mathsf{igp}^+(p_1) = \mathsf{igp}(p_1) + n \cdot M$ and $\mathsf{igp}^+(p_2) = \mathsf{igp}(p_2) + m \cdot M$. By hypothesis, $\mathsf{igp}(p_1) \neq \mathsf{igp}(p_2)$ Assume without loss of generality that $\mathsf{igp}(p_1) > \mathsf{igp}(p_2)$. If $n = m$ then

$$\mathsf{igp}^+(p_1) = \mathsf{igp}(p_1) + n \cdot M > \mathsf{igp}(p_2) + n \cdot M = \mathsf{igp}(p_2) + m \cdot M = \mathsf{igp}^+(p_2).$$

By definition of M, we have $M \geqslant \mathtt{diam}(\mathsf{G}, \mathsf{igp}) \geqslant \mathsf{igp}(p_1), \mathsf{igp}(p_2) > 0$. Thus, if $n > m$ then,

$$\begin{aligned}\mathsf{igp}^+(p_1) &= \mathsf{igp}(p_1) + n \cdot M > n \cdot M \geqslant (m+1) \cdot M \\ &= M + m \cdot M \geqslant \mathsf{igp}(p_2) + m \cdot M = \mathsf{igp}^+(p_2)\end{aligned}$$

Similarly, if $n < m$ then

$$\begin{aligned}\mathsf{igp}^+(p_2) &= \mathsf{igp}(p_2) + m \cdot M > m \cdot M \geqslant (n+1) \cdot M \\ &= M + n \cdot M \geqslant \mathsf{igp}(p_1) + n \cdot M = \mathsf{igp}^+(p_1).\end{aligned}$$

Thus, in any case, $\mathsf{igp}^+(p_1) \neq \mathsf{igp}^+(p_2)$. We conclude that igp^+ is total and therefore also ECMP-free.

Let $u, v \in \mathsf{V}(\mathsf{G})$. We now show that at least one edge $e \in \mathsf{E}(\mathsf{G}, u, v)$ belongs to a shortest path with respect to igp^+. Let $e \in \mathsf{E}(\mathsf{G}, u, v)$ be such that $\mathsf{igp}(e)$ is minimum. Since igp is total, this edge is unique. By Proposition 2.3, e belongs to a shortest path if and only if e is a shortest path between u and v. Suppose that e is not a shortest path for igp^+. Then there exists a path $p = (e_1, \ldots, e_n)$ from u to v such that $\mathsf{igp}^+(p) < \mathsf{igp}(e)$. Since e is the unique edge between u and v of minimum cost, $n \geqslant 2$. By definition of M, we have $M \geqslant \mathsf{igp}(e)$ so

$$\mathsf{igp}^+(p) = \mathsf{igp}(p) + n \cdot M \geqslant \mathsf{igp}(p) + 2 \cdot M > 2 \cdot M \geqslant 2 \cdot \mathsf{igp}(e) > \mathsf{igp}(e).$$

This contradicts the fact that e is not a shortest path for igp^+. Therefore igp^+ is complete. □

Corollary 7.7. *Let* G *be a graph and* $\mathsf{igp} : \mathsf{E}(\mathsf{G}) \to \mathbb{N}$ *defined such that*

$$\mathsf{igp}(e) = 2^{\mathsf{idx}(e)} + 2^{|\mathsf{E}(\mathsf{G})|}.$$

Then igp *is ECMP-free and complete.*

Proof. We have already observed that $e \mapsto 2^{\mathsf{idx}(e)}$ is total and thus ECMP-free by Lemma 7.5. Completeness then follows from Lemma 7.6 by observing that

$$2^{|\mathsf{E}(\mathsf{G})|} = \left(\sum_{e \in \mathsf{E}(\mathsf{G})} 2^{\mathsf{idx}(e)} \right) + 1 > \max \left(\mathtt{diam}(\mathsf{G}, \mathsf{igp}), \max_{e \in \mathsf{E}(\mathsf{G})} \mathsf{igp}(e) \right).$$

□

The problem with these IGP weights is that they are exponential with respect to the number of edges in the graph. In practice IGP weights are represented with a 16-bit integer and thus is maximum value is $2^{16} - 1 = 65535$. This makes them useless in practice since they can only be implemented on a network with at most 15 edges.

This motivates the following problem.

Problem 5 (Minimum weight complete weighting)

Input: *A network* G.
Output: *A complete and ECMP-free weighting* $\mathsf{igp} : \mathsf{E}(\mathsf{G}) \to \mathbb{N}$ *such that* $\max_{e \in \mathsf{E}(\mathsf{G})} \mathsf{igp}(e)$ *is minimum.*

Any approach for solving Problem 5 that is based on Lemma 7.6 is domed to fail. To see why this is true consider $\mathcal{K}_n$, the complete graph on n nodes. It is not hard to see that $\mathcal{K}_n$ contains an exponential number of simple paths.

Any permutation of n elements corresponds to a simple path on $\mathcal{K}_n$ of length $n-1$ and there are $n!$ permutations of n elements. Since a total weight must assign a different weight to *every* simple path, this shows that a total weight on the complete graph $\mathcal{K}_n$ will be such that at least one simple path of length $n-1$ has a weight of at least $n!$. Therefore this path must contain one edge of weight $\frac{n!}{n-1}$ since otherwise the total weight of the path would be lower than $n!$.

This shows that total weightings require exponential weights on any graph with an exponential number of paths. Most graphs have an exponential number of simple paths with respect to its size. Hence, using Lemma 7.6 is bound to provide weights that are very high. In contrast, it is not hard to see that $e \mapsto 1$ is an ECMP-free and complete weighting of $\mathcal{K}_n$. This shows that such exponential bounds do not apply for the weight that we are looking for, only for total ones.

If we only care about the practical applicability of the weights, we can relax Problem 5 into the following one.

Problem 6 (Implementable complete weighting)

Input: *A network* G.
Output: *A complete and ECMP-free weighting* $\mathsf{igp} : \mathsf{E}(\mathsf{G}) \to \mathbb{N}$ *such that* $\max_{e \in \mathsf{E}(\mathsf{G})} \mathsf{igp}(e) \leqslant 2^{16} - 1$.

7.4.2 Prime-based complete IGP

We propose a partial solution to Problem 6. Our solution is able to find a solution for 97.7% of the instances in our dataset. Let $m = |\mathsf{E}(\mathsf{G})|$ be the number

of edges in the graph and $\mathbb{P}_m = \{\pi_0, \ldots, \pi_{m-1}\}$ a set of m prime numbers such that $\pi_i \leqslant \pi_{i+1}$. It is well known that two sets of distinct prime numbers have a different product. Our idea is based on the fact that $\log(x \cdot y) = \log(x) \cdot \log(y)$. If we could set any real valued IGP weights, one solution would be to use $e \mapsto \log(\pi_{\mathsf{idx}(e)})$ because the unicity of products between prime numbers would translate into unique sums and thus unique path weights. To simplify the notations, we write π_e instead of $\pi_{\mathsf{idx}(e)}$. In this way, by what we observe above, two distinct paths would necessarily have distinct IGP costs.

Since we need integers we will use truncated logarithms instead. These logarithms are defined as

$$\overline{\ln}^s(x) = \lfloor 10^s \cdot \ln(x) \rfloor.$$

For instance, $\overline{\ln}^4(5) = \lfloor 10^4 \cdot 1.60943 \rfloor = 16094$. The idea is, starting from $s = 1$, to grow s until the IGP weights defined by $e \mapsto \overline{\ln}^s(\pi_e)$ are ECMP-free. A priori, nothing guarantees that doing so will eventually achieve it but we will prove shortly that it does. Since we also want the IGP weights to also be complete, we use a slight different set of IGP weights $\mathsf{igp}^s : E(G) \to \mathbb{N}$ defined as $\mathsf{igp}^s(e) = \overline{\ln}^s(\pi_e) + \overline{\ln}^s(\pi_m)$. The intuition behind the addition of the largest $\overline{\ln}^s(\pi_m)$ term for achieving completeness is similar to the addition of the constant M from Lemma 7.6. We now prove that by growing s, igp^s eventually becomes ECMP-free and complete.

Proposition 7.8. *Let A and B be two distinct subsets of $\mathcal{P}_m$, $P = \prod_{x \in \mathcal{P}_x} x$ and $q \in \mathcal{P}_m$. If $s > \log_{10}(2m \cdot q^m P)$, then*

$$\sum_{x \in A} \left(\overline{\ln}^s(x) + \overline{\ln}^s(q) \right) \neq \sum_{x \in B} \left(\overline{\ln}^s(x) + \overline{\ln}^s(q) \right).$$

Proof. Given any s and $X \subseteq \mathcal{P}_m$, we have

$$10^s \cdot \ln\left(\prod_{x \in X} x \right) \geqslant \sum_{x \in X} \overline{\ln}^s(x) \geqslant 10^s \cdot \ln\left(\prod_{x \in X} x \right) - |X| \tag{7.1}$$

Let $q \in \mathcal{P}_m$. Write $a = \prod_{x \in A} x$ and $b = \prod_{x \in B} x$. Assume without loss of generality that $q^{|A|}a > q^{|B|}b$ (they are not equal because they contain distinct prime numbers). Then by (7.1)

$$\sum_{x \in A} \overline{\ln}^s(x) + \overline{\ln}^s(q) \geqslant \sum_{x \in A} 10^s \ln(xq) - 2 \geqslant 10^s \ln(q^{|A|}a) - 2|A|$$

and $\sum_{x \in B} \overline{\ln}^s(x) + \overline{\ln}^s(q) \leqslant 10^s \ln(q^{|B|}b)$. Therefore

$$\begin{aligned}
\sum_{x \in A} \overline{\ln}^s(x) + \overline{\ln}^s(q) - \sum_{x \in B} \overline{\ln}^s(x) + \overline{\ln}^s(q) &\geqslant \\
10^s \ln(q^{|A|}a) - 2|A| - 10^s \ln(q^{|B|}b) &\geqslant \\
10^s \ln\left(q^{|A|}a / q^{|B|}b\right) - 2|A| &\geqslant \\
10^s \ln\left(q^{|A|}a / (q^{|A|}a - 1)\right) - 2|A| &\geqslant \\
10^s / (q^{|A|}a) - 2|A| \geqslant 10^s / (q^m P) - 2m &
\end{aligned}$$

Which is positive as long as $s > \log_{10}(2m \cdot q^m P)$. □

Corollary 7.9. *If* $s > \log_{10}(2m \cdot q^m P)$ *then* igp^s *is ECMP-free.*

Proof. Let p_1, p_2 be two paths on G and $q = \pi_m$. By Proposition 7.8,

$$\mathsf{igp}^s(p_1) = \sum_{e \in E(p_1)} \left(\overline{\ln}^s(e) + \overline{\ln}^s(\pi_m) \right) \neq \sum_{e \in E(p_2)} \left(\overline{\ln}^s(e) + \overline{\ln}^s(\pi_m) \right) = \mathsf{igp}^s(p_2).$$

Therefore igp^s is total and thus, by Lemma 7.6, this means that it will also be ECMP-free. □

Proposition 7.10. *The weights* igp^s *are complete for any* $s \geqslant 1$.

Proof. Let $u, v \in V(G)$. Assume that the shortest path p from u to v is not a single edge. Let e be any edge between u and v. Then,

$$w(p) = |E(p)| \cdot \overline{\ln}^s(\pi_m) + \sum_{e \in E(P)} \overline{\ln}^s(\pi_e) > |E(p)| \cdot \overline{\ln}^s(\pi_m) \geqslant$$
$$2 \cdot \overline{\ln}^s(\pi_m) \geqslant \overline{\ln}^s(\pi_e) + \overline{\ln}^s(\pi_m) = w(e)$$

Therefore p cannot be a shortest path proving that igp^s is complete. □

These results provide a theoretical guarantee that the above mentioned process will eventually converge towards ECMP-free and complete weights. However, even for small graphs the value of $\log_{10}(2m \cdot q^m P)$ is large and thus, if $s > \log_{10}(2m \cdot q^m P)$, then 10^s will be really huge. But this a theoretical bound on how long we need to wait to reach a *total* function. But we do not care about totality, we only want ECMP-freeness (completeness is true for any s). Therefore, our algorithm will iterate over s and, at each step, check whether ECMP-freeness holds with the hope that this will occur a long time before totality occurs. Our experiments show that this is indeed the case. Figure 7.14 shows the distribution of the value of s over all topologies from out dataset. We can see that this value is indeed much smaller than the theoretical bound.

We analyzed the percentage of topologies for which this process results in weights that go above the maximum configurable value $2^{16} - 1$. It turns out that for this happens only for 2.3% of the topologies as show in the CDF in Figure 7.15. The threshold value is shown with a dotted line.

Algorithm 17 provides a pseudo-code implementation of this algorithm described above. The first step of the algorithm is to compute a set of $m = |E(G)|$ prime numbers. We can find these by iterating over the integers $2, 3, 5, 7, 9, \ldots$ and using any primality test algorithm to check which ones are prime numbers. The Prime number theorem [12] tells us that the m-th prime number is close to $m \cdot \ln(m)$ so we find them in a small number of steps. This is not the most efficient way to achieve this but since m is relatively small it is fast enough for our purposes.

7.4.3 Randomized complete IGP

In practice, there seems to be a much simpler solution for generating relatively small ECMP-free and complete IGP weights. We performed some experiments that showed that random weighs tend to be ECMP-free as long as the maximum value is not too small. Therefore, a solution for generating ECMP-free weights

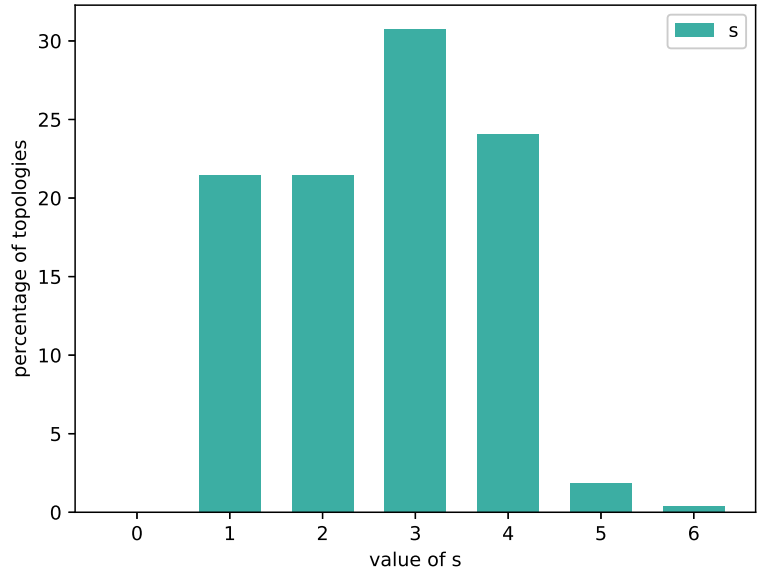


Figure 7.14: Distribution, over all topologies, of the exponent s required for igp^s to be ECMP-free and complete.

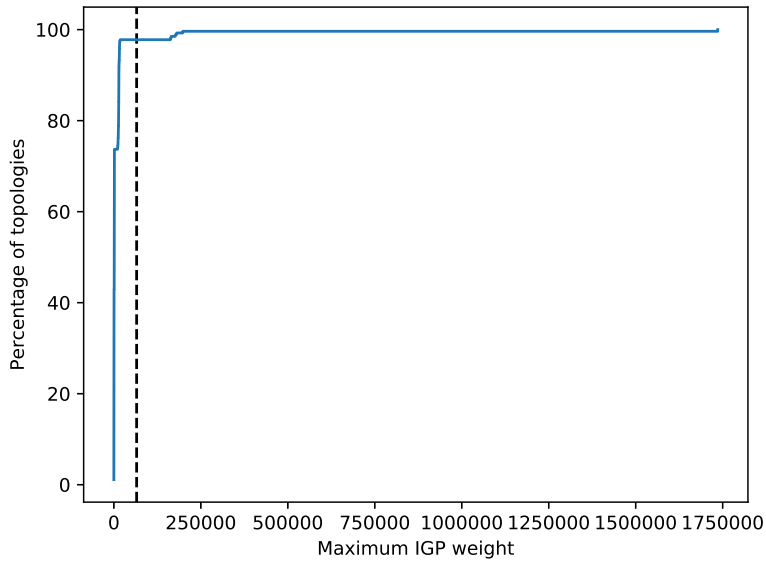


Figure 7.15: CDF of maximum weight obtained over all topologies.

Algorithm 17 primeIGP (G)

```
[compute the first |E(G)| primes]
m ← |E(G)|
P ← {2}
p ← 3
while |P| < m do
    if isPrime(p) then
        P ← P ∪ {p}
    p ← p + 2
[initialize weights]
f ← 1
for e ∈ E(G) do
    w(e) ← ⌊f · log(p_e)⌋ + ⌊f · log(p_m)⌋
[iterate until weights converge]
while not ECMP-free(w) or not complete(w) do
    f ← 10 · f
    for e ∈ E(G) do
        w(e) ← ⌊f · log(p_e)⌋ + ⌊f · log(p_m)⌋
return w
```

Algorithm 18 randomIGP (G, M)

```
w ← random-w(G, M)
while not ECMP-free(w) or not complete(w) do
    w ← random-w(G, M)
return w
```

that are also complete is to randomly generate weights for each edge and then adding the maximum generated weight to each edge to guarantee completeness. Algorithms 18 and 19 formalize this process. Note that without the step of adding the maximum weight, our experiments showed that the algorithm has a low probability of success. This is normal because otherwise there is no control over the weight of a single edge and so it can easily happen that it is assigned a high value making it impossible for it to belong to a shortest path.

In practice we tried it over all topologies with $M = 100$ for each of them we found a solution on average in at most 2 iterations (over 100 runs). For this reason, in practice we strongly recommend using this simple algorithm. However we were unable to prove any bound on the probability of success of this algorithm. Hence we cannot say whether it will work well beyond our dataset. We leave the following problem as another interesting open problem.

Lemma 7.11. *For any network* G *and* $M \in \mathbb{N}$ *the weight function produced by Algorithm 19 is complete.*

Proof. Let w be the random weights produced by the algorithm after the first loop and w' the final weights. Let $M = \max_{e \in E(G)} w(e)$. Let $u, v \in V(G)$ and p be a path from u to v with at least two edges. Then

$$w'(p) = w(p) + |E(p)| \cdot M > 2 \cdot M \geqslant w(e) + M = w'(e)$$

for any edge $e \in E(G)$. Therefore the shortest path between u and v must be a single edge $e \in E(G, u, v)$. □

Algorithm 19 random-w(G, M)

1: **for** $e \in \mathsf{E}(\mathsf{G})$ **do**
2: $\quad w(e) \leftarrow \mathsf{random}(1, \ldots, \mathsf{M})$
3: **for** $e \in \mathsf{E}(\mathsf{G})$ **do**
4: $\quad w'(e) \leftarrow w(e) + \max_{e \in \mathsf{E}(\mathsf{G})} w(e)$
5: **return** w'

Problem 7 (Randomized weighting)

Input: *A network* G *and a integer constant* $\mathsf{M} \in \mathbb{N}$.
Output: *The probability that* random-w(G, M) *outputs an ECMP-free IGP weight function.*

7.4.4 Cycle covers with ECMP-free and complete IGP

In this section we evaluate the benefits of using ECMP-free and complete IGP weights. We start by evaluating the minimum number of segments required in a minimum segment cost sr-cycle cover. Recall that Algorithm 13 computes a sr-cycle cover such that the maximum number of segments in any sr-cycle is as small as possible. In Figure 7.6 we already evaluated the segment cost of the solutions obtained by this algorithm over the original IGP weights. Figure 7.17 shows the distribution of the segment costs with and without special IGP weights. We observe that ECMP-free and complete weights obtain the desired effect of greatly reducing the segment cost of minimum cost sr-cycle covers. For 100% of the topologies we need sr-cycles with segment cost at most 4 whereas before about 55% of the topologies required sr-cycles with segment cost 5 or more.

Next we analyze the benefits of using special IGP weights with respect to the segment cost of the identification sr-cycles. Recall that our network monitoring solution uses probing sr-cycles to periodically check whether every link is still up and one identification sr-cycles to pinpoint a failure when a monitoring probe fails to return to the vantage point. Figure 7.17 shows how much do we gain in terms of segment cost by having ECMP-free and complete IGP weights. We can see that using these weights makes it possible to implement such a monitoring scheme on a lot more networks since the maximum number of segments required becomes 9 rather than the original high value of 19.

Conclusion

In this chapter we proposed a solution for identifying single-link failures in a network. We showed that even if multiple failures occur, our algorithm can still identify a pair of edges e, $\mathsf{rev}(e)$ such that one of them is faulty with certainty.

Our experiments show that our algorithm runs in a reasonable amount of time considering that these cycles need to be installed only once in the network.

Our solution works on the original topology and prides the theoretical minimum number of segments required to implement any cycle cover using segment routing. This was achieved by using our reachability theory developed in Chap-

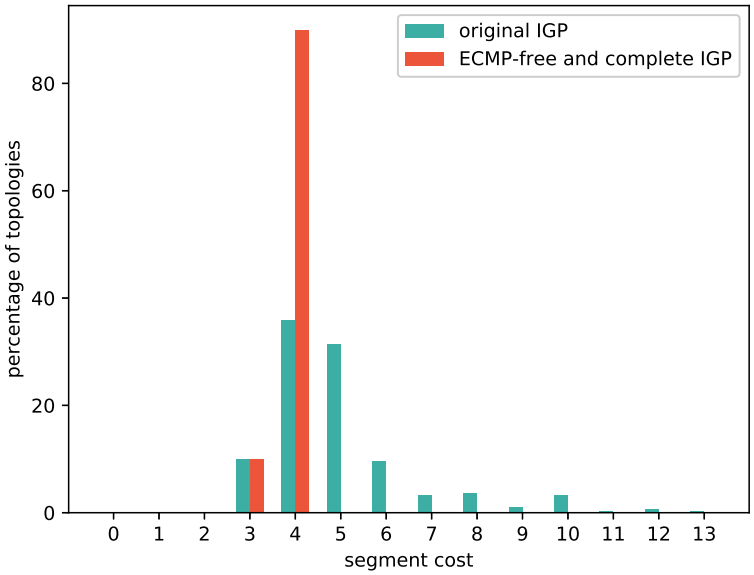


Figure 7.16: Distribution of the maximum segment cost of the probing sr-cycles with and without ECMP-free and complete weights.

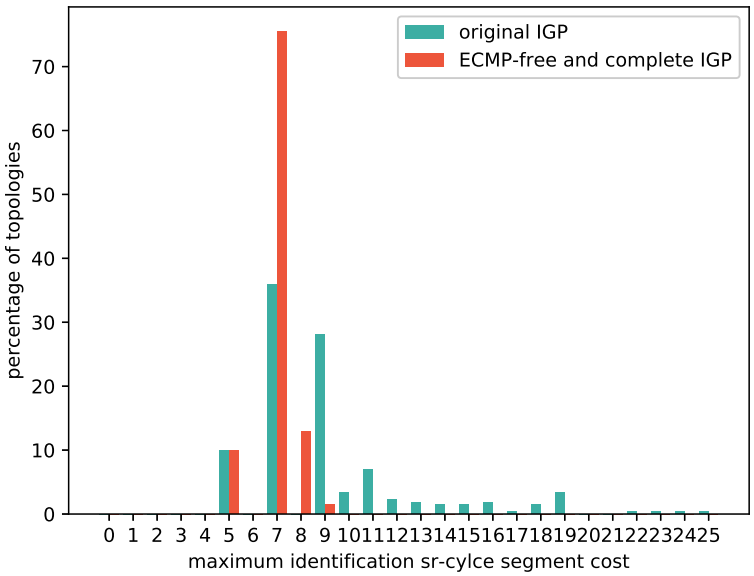


Figure 7.17: Distribution of the maximum segment cost of the identification sr-cycles with and without ECMP-free and complete weights.

ter 4. We also showed that if this segment cost is too high, we can reduce it by working with a dual topology maxing the whole network monitoring process cost at most 9 segments in the worst case, over all topologies from out dataset.

This was a second example where column generation provides good results when applied to solve an segment routing optimization problem.

Chapter 8

Disjoint paths with SR

Introduction

For an Internet Service Provider (ISP), providing disjoint paths to its customers might be one of those advanced connectivity services that generate more revenues. This is what emerged from our discussion with a national ISP that we call "reference ISP". When we met them, this ISP's operators themselves steered the discussion towards possibilities to provide disjoint paths between sites to which a customer is connected. They were motivated by requests from banks and financial customers.

We have quickly realized that the disjoint-path connectivity service has a much bigger market than our reference ISP. An illustration is provided by the NANOG email discussion about a major outage of the Bell network on August 4th, 2017 [42]. The email thread started with Bell's customers complaining that both Internet and mobile connectivity were completely absent in East Canada, affecting banking, ATM, land lines and even 911 services. When a single fibre cut was indicated as the cause of the outage, someone expressed doubts that ISPs really provide geographically diverse circuits, irrespectively of what they promise and sell. The following emails discussed the impossibility to work around this limitation by relying on two providers, as their networks may share the same physical infrastructure (fibres, conduit, etc.), without the ISPs even knowing it – as they do not share information between each other.

The discussion we had with the reference ISP's operators was indeed focused on *providing disjoint paths within a single ISP*, their own. A possible solution [54] to achieve this goal is to deploy two parallel networks, say a red and a blue copy of the same topology, and configure the intra-domain routing protocol (IGP) so that any packet is forwarded in only one of the two networks – i.e., packets that enter the red copy are only forwarded in the red copy. The few links between the two networks are only used if one of the two copies is partitioned. This architecture provides disjoint paths by design, but it is very expensive since the entire network is doubled. The reference ISP's operators were therefore reluctant to deploy it. Of course, they were also aware that MPLS tunnels can be created over arbitrary paths with RSVP-TE [9], including disjoint ones, on an existing infrastructure. However, they were in the process of moving away from MPLS, in order to avoid its operational limitations [48],

its sub-optimal usage of resources [43] and its scalability challenges with respect to the routers' state [20, 33].

Looking at other ISPs, our operators were instead considering Segment Routing (SR). Motivated by this, we dedicate this chapter to the study of the problem of computing and implementing disjoint paths over a network with segment routing. We also provide a solution for leveraging some properties of sr-paths to show how we can provide disjoint paths that are robust to link failures.

8.1 Disjoint paths and network flows

The problem of computing disjoint paths in a graph is one of those ubiquitous problem that has driven a lot of research over the years. Numerous algorithms exist for solving it, most of them being some variant of the more general maximum flow problem [4, 49].

The maximum flow problem is the problem of finding the maximum amount of information that can be sent between two given nodes in a network. It is closely related to the multi-commodity flow problem that we studied in Chapter 6. Instead of being given a set of demands, we are given a source $s \in V(G)$ and a destination $t \in V(G)$ and we are asked what is the maximum amount of traffic that can be routed between those two nodes without exceeding the capacity of any edge. Another way to see it is to imagine that we have an infinite amount of unit demands between s and t and we are asked what is the maximum number of such demands that can simultaneously be routed of the network without exceeding any link capacity.

The maximum flow problem admits an IP formulation that is very similar to the formulation that we gave for the MCF. If we let x_e define the amount of demands routed over edge e it can be shown that the following LP models the problem [4].

$\mathsf{MAX\text{-}FLOW}(G, s, t)$

$$
\begin{aligned}
\textbf{max} \quad & \sum_{e \in \delta^+(s)} x_e & & & \\
\textbf{s.t.} \quad & \sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} x_e & = & 0 & \forall v \in V(G) \setminus \{s, t\} \\
& x_e & \leqslant & \mathsf{cap}(e) & \forall e \in E(G) \\
& x_e & \in & \mathbb{N} &
\end{aligned}
$$

It is possible to show that the linear programming relaxation of this problem obtained by replacing $x_e \in \mathbb{N}$ by $x_e \geqslant 0$ is equivalent as long as the capacities are integral [4]. This makes the maximum flow problem one of those rare cases where the integrality constraints no not increase the difficulty of the problem. Numerous polynomial time algorithms have been developed for solving the maximum flow problem [4, 16, 17, 30]. If we set *unit capacities* on the edges and think about the maximum flow problem as one answering the question of what is the maximum amount of unit demands that we can route from s to t, we see that we actually end up with the *maximum number of disjoint path between* s *and* t. This is the case since each of the routed demands must follow a path that

shares no edges with any of the other demand paths or otherwise some edge would carry at least two units of traffic, thus exceeding its capacity.

In other words, this means that we can model the problem of computing the maximum number of edge-disjoint paths between s and t by replacing $\mathsf{cap}(e)$ by 1 in the MAX-FLOW model. By doing so, we obtain the following model which can also be solved efficiently.

MAX-EDP(G, s, t)

$$
\begin{aligned}
\textbf{max} \quad & \sum_{e \in \delta^+(s)} x_e & & & \\
\textbf{s.t.} \quad & \sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} x_e & = & 0 & \forall v \in V(G) \setminus \{s, t\} \\
& x_e & \leqslant & 1 & \forall e \in E(G) \\
& x_e & \in & \mathbb{N} &
\end{aligned}
$$

This model focuses solely one providing a maximum amount of paths. It does not care about the quality of those paths, for instance, in terms of latency. It turns out that if we wish to minimize total latency of the path set we can still do it in polynomial time. This problem is known, in general, as the minimum cost maximum flow problem [4]. We can easily adapt the above model to obtain a set of n disjoint paths (if they exist) whose total latency is as small as possible by requiring the total flow out of s to be n and minimizing the sum of the latencies of the edges with a non-zero flow as shown in the following model.

MIN-LAT-EDP(G, s, t)

$$
\begin{aligned}
\textbf{min} \quad & \sum_{e \in E(G)} \mathsf{lat}(e) \cdot x_e & & & \\
\textbf{s.t.} \quad & \sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} x_e & = & 0 & \forall v \in V(G) \setminus \{s, t\} \\
& \sum_{e \in \delta^+(s)} x_e & = & n & \\
& x_e & \leqslant & 1 & \forall e \in E(G) \\
& x_e & \in & \mathbb{N} &
\end{aligned}
$$

If we want n to be as large as possible we can simply first compute the optimal solution of MAX-EDP(G, s, t) and set n to that value. Surprisingly, this problem can also be solved in polynomial time [4]. Unfortunately, slight variations of this problem that are of interest for computer networks quickly become NP-hard. For instance, if we want to minimize the maximum latency over all the paths rather than the latency sum, the problem is NP-hard even if $n = 2$ [38, 39]. Another change that, perhaps surprisingly, makes the problem NP-hard is to have two origins s_1, s_2 and two destinations t_1, t_2 and ask for a pair of disjoint paths, one from s_1 to t_1 and another from s_2 to t_2 [53]. One might naively think that we could get away by adding a fake node s linked to

both s_1, s_2 and link t_1, t_2 to some other fake t and then compute the maximum flow from s to t. Unfortunately, nothing in the model forces the flow originating from s_1 to go to t_1 instead of t_2 (and vice-versa). In order to force some origin to destination assignment, we cannot rely on a maximum flow model but rather on a multi-commodity flow model which, as we have already seen, is harder to solve.

8.2 Disjoint sr-paths

In the previous section we talked about the general problem of finding sets of disjoint paths over a network. However, the paths found by these algorithms can be arbitrary and thus hard to implement with segment routing as we will show shortly.

In this section we redefine the problem in terms of segment routing and adapt the MIP models from the previous section accordingly. We start by defining what disjoint sr-path are and the problem of finding a maximum cardinality set of sr-paths.

Definition 8.1. *Let* G *be a network and* $\vec{p}, \vec{q}$ *be two sr-paths on* G*. We say that* $\vec{p}$ *and* $\vec{q}$ *are* disjoint *if* $E(\vec{p}) \cap E(\vec{q}) = \emptyset$*.*

Problem 1 (Maximum edge-disjoint sr-paths problem)

Input: *A network* G*, two nodes* $s, t \in V(G)$ *and* $k \in \mathbb{N}$*.*
Output: *A set of sr-paths* $\{\vec{p}_1, \ldots, \vec{p}_n\} \subseteq \vec{\mathcal{P}}_k(s, t)$ *such that for each* $i \neq j$*,* $\vec{p}_i$ *and* $\vec{p}_j$ *are disjoint and* n *is maximum.*

Recall that in case of ECMP between two consecutive segments of $\vec{p}$, the set $E(\vec{p})$ contains the edges belonging to *all* those ECMP paths. In other words, we are requiring all those shortest paths corresponding to $\vec{p}$ and $\vec{q}$ to be edge-disjoint. This is necessary because we can never be sure where exactly the traffic will flow when using a sr-path. In this way, we ensure that no matter how traffic is routed over ECMPs, no edge will carry packets from two distinct sr-paths $\vec{p}_i$ and $\vec{p}_j$. Figure 8.1 illustrates this with source node a and destination node h. The two sr-paths displayed on it are not disjoint because there might both use edge (e, h) in the event of the green one using path $((a, c), (c, d), (d, e), (e, h))$ and the blue one using $((a, b), (b, e), (e, h))$. This can be prevented by, for instance, forcing the blue path to pass thought node f as shown in Figure 8.2.

If we ignore the segmentation cost constraints from Problem 1, the simplest algorithm to compute a set of edge-disjoint sr-paths is to leverage the minimum segmentation algorithm proposed in Chapter 4 to segment the set of paths produced by the minimum cost flow algorithm. As usual with this kind of approach, this solution has the drawback of granting no control over the segment cost of the output. For this reason, we start by analyzing the distribution of the costs over all topologies. For each topology in our dataset and each pair of distinct nodes, we used a minimum cost maximum flow algorithm to compute the maximum number of disjoint paths between those nodes whose total latency is as small as possible. Then, we segment each of those paths and compute the maximum number of segments required to implement those paths

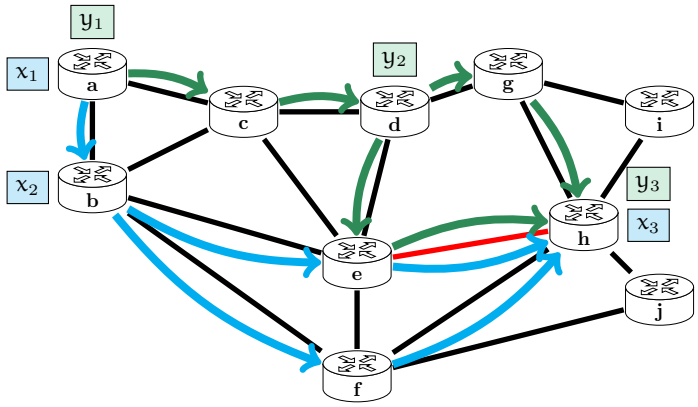


Figure 8.1: The sr-paths $\langle \mathtt{a}, \mathtt{d}, \mathtt{h} \rangle$ and $\langle \mathtt{a}, \mathtt{b}, \mathtt{h} \rangle$ are not edge-disjoint because their edge sets intersect over $(\mathtt{e}, \mathtt{h})$.

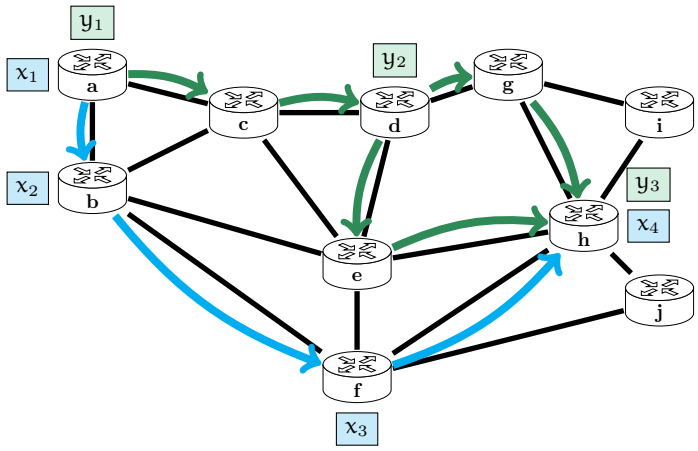


Figure 8.2: The sr-paths $\langle \mathtt{a}, \mathtt{d}, \mathtt{h} \rangle$ and $\langle \mathtt{a}, \mathtt{b}, \mathtt{f}, \mathtt{h} \rangle$ are edge-disjoint.

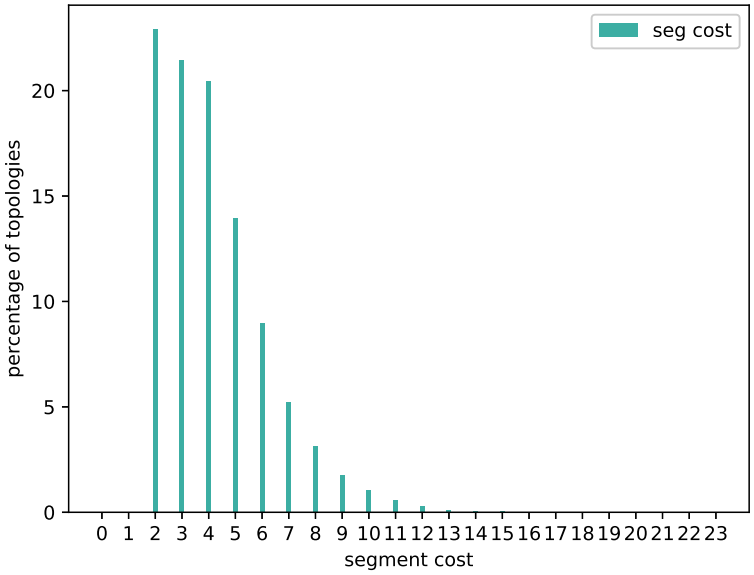


Figure 8.3: Distribution of the maximum segment of maximum cardinality sets of disjoint paths with total minimum latency.

on the network. Figure 8.3 shows the distribution of the segment costs. We observe that for more than 20% of the pairs we need 6 or more segments. This motivated us to propose solutions that are able to limit the number of segments in the output.

8.2.1 Maximum set of disjoint sr-paths

We propose two MIP models for adapting the graph model $\mathsf{MAX\text{-}EDP}(G, s, t)$ to Problem 1. We start by defining an indicator function telling use whether an edge belongs to the shortest paths between two given nodes. That is, let I to denote a function $V(G)^2 \times E(G) \to \{0, 1\}$ defined such that $I(u, v, e) = 1$ if and only if $e \in E(SP(u, v))$.

Our first model is an adaptation of the traffic engineering segment model $\mathsf{SRTE\text{-}SEG}(G, \mathcal{D})$ proposed in Chapter 6. Our demand set contains unit demands between s and t. Each such demand corresponds to a sr-path that is edge-disjoint from the others. We can select the number of demands to be equal to the out-degree of s, since this is an upper bound on the number of disjoint paths from s to any other node. Our objective will by to route a maximum amount of demands in a way such that no two demands are routed over the same edge. We use variables x^d_{uv} saying whether $SP(u, v)$ is used by the sr-path corresponding to demand d. We replace the capacity constraints by disjointness constraints which consists of requiring that for each edge, at most one demand is routed over it. The rest of the model is the same as $\mathsf{SRTE\text{-}SEG}(G, \mathcal{D})$ which uses flow constraints to ensure that paths go from s to t.

SR-EDP-SEG(G, s, t)

$$\textbf{max} \sum_{u \in V(G)} \sum_{d=1}^{r} x_{su}^{d} \qquad \textbf{s.t.}$$

$$\sum_{d=1}^{r} \sum_{u \in V(G)} \sum_{v \in V(G)} x_{uv}^{d} \cdot I(u, v, e) \leqslant 1 \qquad \forall e \in E(G)$$

$$\sum_{u \in V(G) \setminus \{v\}} x_{uv}^{d} - \sum_{u \in V(G) \setminus \{v\}} x_{vu}^{d} = 0 \qquad \forall d, \; \forall v \in V(G) \setminus \{s, t\}$$

$$\sum_{d=1}^{r} \sum_{u \in V(G)} x_{us}^{d} + x_{tu}^{d} = 0 \qquad \forall d \in \{1, \ldots, |\delta^{+}(s)|\}$$

$$\sum_{u \in V(G)} \sum_{v \in V(G)} x_{uu}^{d} \leqslant k \qquad \forall d \in \{1, \ldots, |\delta^{+}(s)|\},$$

$$x_{uv}^{d} \in \{0, 1\} \qquad \forall e \in E(G), \; \forall d \in \{1, \ldots, |\delta^{+}(s)|\}$$

Next, we propose another model whose original idea is due to Bernard Fortz. After we will compare both models. Note that both these models only support node segments in the sr-paths.

A sr-path with only node segments is a sequence $\langle y_1, \ldots, y_l \rangle$ such that each $y_1, \ldots, y_l \in V(G)$. The idea behind Fortz model is to define binary variables x_{uv}^{i} such that $x_{uv}^{i} = 1$ if and only if u and v appear as consecutive segments $y_i = u$ and $y_{i+1} = v$ of a sr-path used in the solution. These variables are defined for $i = 1, \ldots, k-1$ where k is the maximum segment cost that we want to allow the paths to have. Consider for instance that we have a solution where $x_{sa}^{1} = x_{ab}^{2} = x_{bt}^{3} = 1$. This will correspond to using the sr-path $\langle s, a, b, t \rangle$ as shown in Figure 8.4. So, basically, the index i is saying the position at which we use each segment. Note that a more intuitive model would be to drop the i index and use variables x_{uv} to mean that we use the shortest paths between u and v to route traffic. By adding flow conservation constraints similar to the ones used in model MAX-FLOW(G, s, t) we can make sure that these variables actually come together to constitute sr-paths. However, this gives no way of restricting the segment cost of those paths.

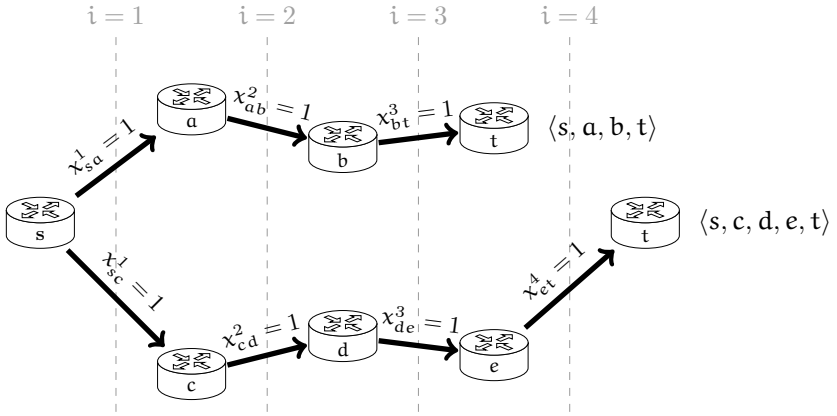


Figure 8.4: Two examples of how the variables x^i_{uv} define sr-paths.

SR-EDP-FORTZ(G, s, t)

$$\textbf{max} \quad \sum_{u \in V(G)} x^1_{su}$$

$$\textbf{s.t.} \quad \sum_{i=1}^{k} \sum_{u \in V(G)} \sum_{v \in V(G)} I(u, v, e) \cdot x^i_{uv} \leqslant 1 \quad \forall e \in E(G) \tag{1}$$

$$\sum_{v \in V(G)} x^{i-1}_{vu} - \sum_{v \in V(G)} x^i_{uv} = 0 \quad \forall u \in V(G) \setminus \{s, t\}, \quad i = 2, \ldots, k \tag{2}$$

$$\sum_{i=1}^{k} \sum_{v \in V(G)} x^i_{vs} + x^i_{tv} = 0 \tag{3}$$

$$\sum_{v \in V(G) \setminus \{s\}} x^1_{vu} + \sum_{v \in V(G) \setminus \{t\}} x^k_{uv} = 0 \quad \forall u \in V(G) \tag{4}$$

$$x_e \in \mathbb{N}$$

Constraints (1) ensure that each edge is used only once making sure that the final paths are indeed edge-disjoint. Recall that setting x^{i-1}_{uv} to 1 means that node u is used as a node segment at position $i-1$ in some sr-path in the solution. Thus, if $u \neq t$ we need to make sure that there is also some node v' coming after u in this sr-path as illustrated in Figure 8.5. This is what constraints (2) ensure, that is, that for each u such that x^{i-1}_{vu} for some v, there is also some element v' such that $x^i_{uv'}$, ensuring that the path does not end at a node $u \neq t$. These constraints are similar to the classical conservation constraints that are commonly used in these kinds of models to ensure connectivity.

Constraints (3) simply make sure that s appears only as the first element of the sr-paths and that t occurs only as the last one. Finally, constraints (4) prevent other nodes to be the starting and end-points of paths by ensuring that x^1_{uv} can only be set if $u = s$ and that x^k_{uv} can only be set if $v = t$.

Figure shows a CDF of the runtime needed to compute optimal solutions of

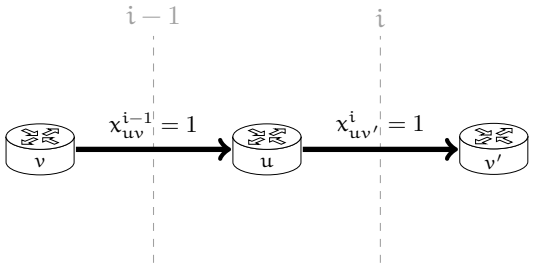


Figure 8.5: If $x_{uv}^{i-1} = 1$.

SR-EDP-SEG(G, s, t) and SR-EDP-FORTZ(G, s, t) using Gurobi. In both cases the maximum number of segments was set to 5. We generated 100 random source-destination pairs and solved both models over these pairs for all instances in our dataset. We can see (in orange) that the segment model is slower than the model proposed by Fortz (in blue). We see that the maximum runtime of the Fortz model is about 3 minutes whereas the maximum runtime of the segment model is about 10 minutes. In both cases this shows that using a MIP solver for computing sets of disjoint sr-paths is feasible in practice in a reasonable amount of time. In order to try to understand whether our sample of 100 pairs is large enough, we computed a box-plot of the run times on the topologies from groups `real` and `rf` of the Fortz model. We selected these because they are the largest ones and we cannot show all results in the box plot. Figure 8.7 shows these. Except for topology `1239`, the runtime does not have a high variance so we can expect that the average runtime is close to the one computed.

We also analyzed how restrictive the segmentation constraints with respect to the existence of disjoint paths. For this we computed the difference between the maximum number of disjoint paths between the sources and the destinations with the maximum number of disjoint sr-paths of segment cost at most 5. Whenever this difference is 0 we know that requiring the path to be implementable with a segment cost of at most 5 posed no restrictions in finding solutions. Table 8.8 shows these results. We can see that for 95% of the pairs the segmentation constraints were not restrictive. This indicates that with about 5 segments we can implement sets of sr-paths that are as large as the theoretical maximum supported on the graph topology.

8.2.2 Minimizing the total latency

For both models it is straightforward to modify the models to minimize the total latency of the sr-paths. In both cases we need to first compute the maximum number of disjoint sr-paths that exist between the source and destination, say P. Then, we need to replace the objective function by a minimization function that adds all latencies together. We also need an additional constraint requiring the total number of paths in the solution to be equal to P. Concretely, in the segment model, SR-EDP-SEG(G, s, t), we can obtain this by replacing the

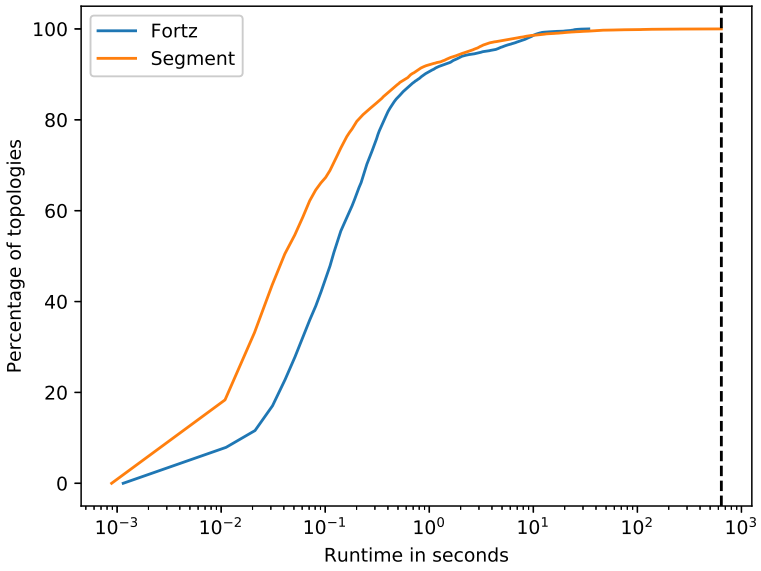


Figure 8.6: CDF over all topologies of the runtime for solving SR-EDP-SEG(G, s, t) and SR-EDP-FORTZ(G, s, t) over 100 randomly selected source-destination pairs.

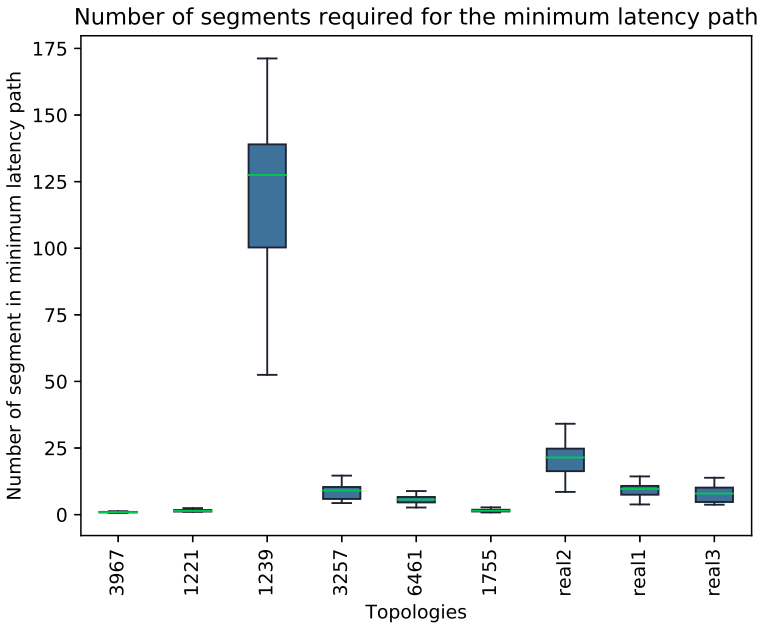


Figure 8.7: CDF over all topologies of the runtime for solving SR-EDP-FORTZ(G, s, t) over 100 randomly selected source-destination pairs.

difference	0	1	2	3	$\geqslant 4$
percentage of s-t	95%	4%	0.4%	0.1%	0.5%

Figure 8.8: Percentage of pairs for each value of the difference between the maximum number of disjoint paths and maximum number of disjoint sr-paths with segment cost at most 5.

objective function by

$$\textbf{minimize} \quad \sum_{d=1}^{r} \sum_{u \in V(G)} \sum_{v \in V(G)} \mathsf{lat}(u,v) \cdot x_{uv}^{d}$$

and adding a constraint

$$\sum_{u \in V(G)} \sum_{d=1}^{r} x_{su}^{d} = P.$$

For the Fortz model, $\mathsf{SR\text{-}EDP\text{-}FORTZ}(G, s, t)$, the change is analogous. The objective function becomes

$$\textbf{minimize} \quad \sum_{i=1}^{k} \sum_{u \in V(G)} \sum_{v \in V(G)} \mathsf{lat}(u,v) \cdot x_{uv}^{i}$$

and we add a constraint requesting P paths starting at the source s:

$$\sum_{u \in V(G)} x_{su}^{1} = P.$$

8.3 Min-max edge-disjoint sr-paths

We now focus on the problem of computing pairs of disjoint paths with a min-max objective function. More specifically, we aim at connecting via disjoint sr-paths a source node s_1 to a destination t_1 and a source node s_2 to a destination t_2 such that the maximum latency among those two paths is as small as possible.

Problem 2 (Min-max edge-disjoint sr-paths)

Input: *A network* G *and* $s_1, s_2, t_1, t_2 \in V(G)$ *such that* $s_1 \neq t_1$ *and* $s_2 \neq t_2$ *and* $k \in \mathbb{N}$.
Output: *Two disjoint sr-paths* $\vec{p}_1 \in \vec{\mathcal{P}}_k(s_1, t_1), \vec{p}_2 \in \vec{\mathcal{P}}_k(s_2, t_2)$ *such that*

$$\max(\mathsf{lat}(\vec{p}_1), \mathsf{lat}(\vec{p}_2))$$

is minimal.

As we mentioned in the previous section, this problem is NP-hard [38, 39].

8.3.1 MIP formulation

We saw above that Fortz model performed better than the segment model for computing maximal sets of disjoint paths. However it is hard to enforce a source to destination assignment with this model. The reason is that different paths are not modeled explicitly. For this reason, we use the segment model for solving Problem 2. With the segment model each path is encoded in the index d of the variables x^d_{uv}. This makes it easy to for the path starting at s_1 to end at t_1 and the path starting at s_2 to end at t_2. The adapted model is the following.

$\mathsf{SR\text{-}2EDP}(G, s_1, s_2, t_1, t_2)$

min λ **s.t.**

$$\sum_{d=1}^{2} \sum_{u \in V(G)} \sum_{v \in V(G)} x^d_{uv} \cdot I(u,v,e) \leqslant 1 \qquad \forall e \in E(G)$$

$$\sum_{u \in V(G)\setminus\{v\}} x^d_{uv} - \sum_{u \in V(G)\setminus\{v\}} x^d_{vu} = 0 \qquad \forall d \in \{1,2\},\ \forall v \in V(G) \setminus \{s_d, t_d\}$$

$$\sum_{u \in V(G)} \sum_{v \in V(G)} \mathsf{lat}(u,v) \cdot x^d_{u,v} \leqslant \lambda \qquad \forall d \in \{1,2\}$$

$$\sum_{u \in V(G)\setminus\{s_d\}} x^d_{s_d u} = 1 \qquad \forall d \in \{1,2\}$$

$$\sum_{u \in V(G)\setminus\{t_d\}} x^d_{u t_d} = 1 \qquad \forall d \in \{1,2\}$$

$$\sum_{d=1}^{2} \sum_{u \in V(G)} x^d_{u s_d} + x^d_{t_d u} = 0 \qquad \forall d \in \{1,2\}$$

$$\sum_{u \in V(G)} \sum_{v \in V(G)} x^d_{uu} \leqslant k \qquad \forall d \in \{1,2\}$$

$$x^d_{uv} \in \{0,1\} \qquad \forall e \in E(G),\ \forall d \in \{1,2\}$$

$$\lambda \geqslant 0$$

8.3.2 Dedicated algorithm

We also proposed a dedicated algorithm for solving this problem. This was actually our original idea that was published in CoNEXT 18 [8]. However we will see that it is actually less efficient and flexible than the MIP formulation. Recall that in our definition of network, we mentioned that each edge is indexed with a unique number between 0 and $|E(G)| - 1$. This is useful for defining parallel edges. These indexes are also important in the context of disjoint paths. From an implementation point of view, we will represent the set of edges corresponding to a sr-path $\vec{p}$, $E(\vec{p})$, as a *bitset* b such that $b_i = 1$ if and only if edge with $\mathsf{idx}(e) = i$ belongs to $E(\vec{p})$.

Bitsets are a very simple and efficient way to represent subsets of $\{0, \ldots, n-1\}$ for some fixed, not too large value of n. Conceptually they are similar to an

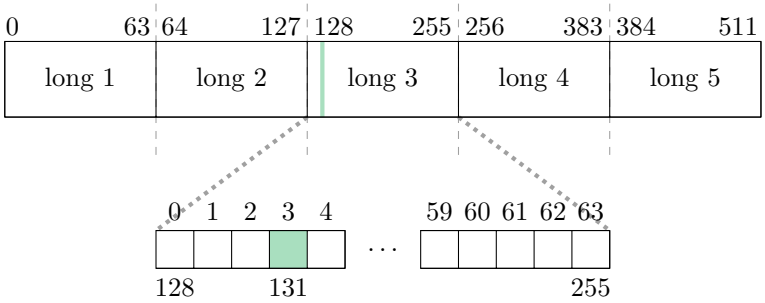


Figure 8.9: Representation of a bitset with $n = 512$.

array of booleans of size n. However, a bitset is represented instead (on a 64-bit machine) with an array of `long`. Each element of the array represents a group of 64 boolean values with its bits. So the bits of the first element in the array will represent elements 0 to 63, the second 64 to 127 and so forth. Figure 8.9 illustrates a bitset for $n = 512$. In this example, element 131 is represented by the forth bit of the third long.

The advantage of this representation over a boolean array representation is that each set operation on a long can be performed in $O(1)$. So for example, to compute the intersection between two bitsets we simply need to loop over the array and perform a bitwise and between corresponding elements. Therefore we only need to perform $n/64$ operations rather n. Even though in big-Oh notation this sill yields the same complexity, $O(n)$, the runtime in practice is 64 times faster which is a gigantic speedup. Using a bitset representation for the set of edges of a sr-path we can then perform set operations over these very efficiently.

In particular, this representation makes it possible to very efficiently check whether or not two sr-paths are disjoint. We exploit this to design an algorithm for solving 2. Given a sr-path $\vec{p}_1$ we can find the minimum latency sr-path $\vec{p}_2$ that is disjoint from it in polynomial time by using the minimum latency sr-path algorithm from Chapter 5. We simply need to adapt it so that it avoids $E(\vec{p}_1)$. To do so, we need to know the set of edges in all sr-paths of the form $\langle x, y\rangle$ where $x, y \in V(G)$. We show how to do this in the next.

Pre-computing the forwarding graphs

Lemma 8.1. *Let* G *be a network and* $x, y \in V(G)$. *Then*

$$E(SP(x, y)) = \bigcup_{e \in \delta^-(SP(x), y)} E(SP(x, e^1)) \cup \{e\}. \tag{8.1}$$

Proof. ($\subseteq$) Let $e \in E(SP(x, y))$. Let $p = (e_1, \ldots, e_n)$ be a shortest path from x to y passing by e. Then $e = e_i$ for some i. Note that $(e_1, \ldots, e_{n-1})$ is a shortest path from x to $e^2_{n-1} = e^1_n$ and, since $e^2_n = y$, $e_n \in \delta^-(SP(x), y)$. Thus, if $i = n$ then e clearly belongs to the right-hand side of (8.1). Otherwise, if $i < n$ then e belongs to the shortest path $(e_1, \ldots, e_{n-1})$ from x to $e^2_{n-1} = e^1_n$. Since $e_n \in \delta^-(SP(x), y))$ we again conclude that e belongs to the rhs of (8.1).

($\supseteq$) Let e be a edge belonging to the rhs of (8.1). There exists $f \in \delta^-(SP(x), y))$ such that either $e = f$ or $e \in E(SP(x, f^1))$. If $e = f$ then $e = f \in \delta^-(SP(x), y)) =$

Algorithm 20 precompute-forwEdges (g)

```
1: for u, v ∈ V(G) do
2:     fwe(u, v) ← Bitset()
3: for u ∈ V(G) do
4:     SP(u) ← dikstra-dag(g, u)
5:     order ← toposort(SP(u))
6:     for v ∈ order do
7:         for e ∈ δ⁻(SP(u), v) do
8:             fwe(u, v) ← fwe(u, e¹) ∪ {e}
9: return fwe
```

$\delta^-(\mathsf{SP}(x,y))$ so it belongs to $\mathsf{SP}(x,y)$. Otherwise, e belongs to a shortest path from x to the origin of f. Since $f \in \delta^-(\mathsf{SP}(x),y)$ we have that $d(x,y) = d(x,f^1) + \mathsf{igp}(f)$. Since e belongs to a shortest path p from x to f^1, we have $d(x,e^1) + \mathsf{igp}(e) + d(e^2,f^1) = d(x,f^1) = \mathsf{igp}(p)$. Then

$$\begin{aligned} d(x,y) &= d(x,f^1) + \mathsf{igp}(f) \\ &= d(x,e^1) + \mathsf{igp}(e) + d(e^2,f^1) + \mathsf{igp}(f) \\ &= d(x,e^1) + \mathsf{igp}(e) + d(e^2,f^2) \\ &= d(x,e^1) + \mathsf{igp}(e) + d(e^2,y) \end{aligned}$$

so that $e \in \mathsf{SP}(x,y)$. □

Using Lemma 8.1 we can leverage the speed of bitsets to efficiently compute $\mathsf{E}(\mathsf{SP}(v,u))$ for all $u,v \in V(G)$. Note that we could always compute them using the definition, that is, computing $\mathsf{SP}(u)$ for all u and then for each v using a breath-first search to extract the subset of edges of $\mathsf{SP}(u)$ that belong to $\mathsf{SP}(u,v)$. Using equation (8.1), we can compute $\mathsf{E}(\mathsf{SP}(u,v))$ by performing $|\delta^-(\mathsf{SP}(u),v)|$ bitset operations. As we mentioned above, in theory this is not faster but it practice it runs faster even due to the usage of bitsets. However, in the next section we will see that applying the same idea to precompute another kind of data that we will need leads to huge gains in runtime. Algorithm 20 shows how we can easily compute this recurrence. For each u we compute the shortest path subnetwork rooted at u and then compute a topological order $v_1, v_2, \ldots, v_n$ to compute $\mathsf{E}(\mathsf{SP}(u,v_i))$ in an order such that when computing $\mathsf{E}(\mathsf{SP}(u,v_i))$ we already computed $\mathsf{E}(\mathsf{SP}(u,e^1))$ for all $e \in \delta^-(\mathsf{SP}(u),v_i)$.

Having pre-computed $\mathsf{E}(\mathsf{SP}(u,v))$ we can easily adapt Algorithm 4 to make sure that the path that is computed avoids all edges in $\mathsf{E}(\vec{p}_1)$. Recall that, according to Chapter 5, the recurrence for the minimum latency path from s_2 to v with segment cost at most i will be

$$sol(i,v) = \min \begin{cases} sol(i-1,v) & \\ sol(i-1,u) + \mathsf{lat}(u,v) & \textbf{s.t } u \in V \\ sol(i-2,r) + \mathsf{lat}(r,e^1) + \mathsf{lat}(e) & \textbf{s.t } u \in V, e \in \delta^-(v) \end{cases}$$

as illustrated by Figure 8.10. In order to guarantee disjointness, we just need to ensure that for each case all pieces are disjoint from $\vec{p}_1$. This means ensuring

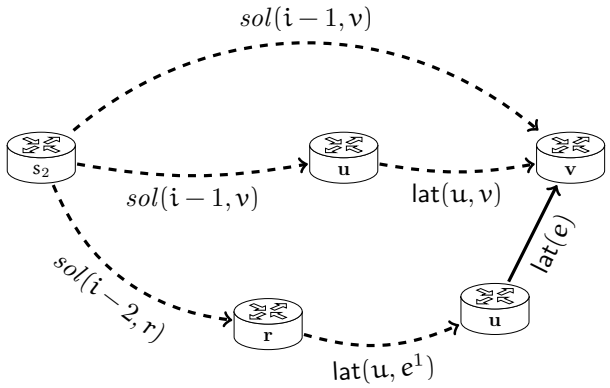


Figure 8.10: Illustration of the *sol* recurrence

that

$$\mathsf{E}(\mathsf{SP}(u, v)) \cap \mathsf{E}(\vec{p}_1) = \emptyset$$

in the second case and that

$$\big(\mathsf{E}(\mathsf{SP}(u, e^1)) \cup \{e\}\big) \cap \mathsf{E}(\vec{p}_1) = \emptyset$$

in the third case. In term of algorithms, this corresponds to adding these conditions to lines 9 and 17 from Algorithm 4, respectively. Note that these changes will have a minor impact over the overall performance of the algorithm since we use bitsets to compute these intersections and $\mathsf{E}(\mathsf{SP}(u, v))$ is given as input for all $u, v \in \mathsf{V}(G)$.

In order to find a pair of paths, we perform a depth-first search on $\vec{p}_1$ and use the above algorithm to maintain the minimum latency sr-path $\vec{p}_2$ that is disjoint from the current partial path $\vec{p}_1$. At each step of the search, we try to extend $\vec{p}_1$ with either a node segment or an adjacency segment. We perform the following steps to avoid exploring useless extensions of $\vec{p}_1$. Let $\vec{p}_1 = \langle x_1, \ldots, x_n \rangle$ be the partial path at given search node, $\vec{p}_2$ the minimum latency sr-path disjoint from the partial path $\vec{p}_1$, l^* the latency of the best solution found so far and x be a node or adjacency segment:

- Let $v = x_n^2$ be the node where $\vec{p}_1$ ends. In the best case, the latency of the completed sr-path $\vec{p}_1$ will be its current latency plus the latency of the minimum latency path between v and t_1 in G. Let's denote that latency by $\mathcal{L}(v, t_1)$. Therefore, if $\max(\mathsf{lat}(\vec{p}_1) + \mathcal{L}(v, t_1), \mathsf{lat}(\vec{p}_2)) \geqslant l^*$ we can stop the search since we will never reach a better solution.

- By Theorem 4.6, there is a solution to Problem 2 where both $\vec{p}_1$ and $\vec{p}_2$ are acyclic. Hence, we can ignore x if $\vec{p}_1 \oplus x$ is cyclic. Checking this can be done efficiently thanks to our bitset representation and forwarding graph edge set pre-computation.

- If $\vec{p}_1 \oplus x$ does not intersect $\vec{p}_2$ then $\vec{p}_2$ remains the minimum latency sr-path that is disjoint from $\vec{p}_1 \oplus x$ so there is not need to re-compute it. Otherwise, we use the algorithm that we described above to compute a

Algorithm 21 disjoint-srpaths (g, s_1, s_2, t_1, t_2)

```
1: p⃗_2 ← min-lat-disjoint-srpath(s_2, t_2, k)
2: l* ← ∞
3: for x ∈ V(G) ∪ E(g) do
4:     disjoint-srpaths-dfs (⟨x⟩, p⃗_2)
5: if l* = ∞ then
6:     return null
7: return p⃗*_1, p⃗*_2
```

new minimum latency sr-path $\vec{p}_2$. If this path does not exist, then it is fruitless to try x as an extension of $\vec{p}_1$.

- If there does not exist a pair of disjoint paths on G, one from x^2 to t_1 and another from s_2 to t_2 then we will never reach a solution by extending $\vec{p}_1$ with x. However, we have seen that checking whether such disjoint paths exists is NP-complete. We use a relaxation of this condition by allowing the path from x^2 to go to t_2 or the path from s_2 to go to t_1 which is equivalent to checking whether the maximum flow between $\{x^2, s_2\}$ and $\{t_1, t_2\}$ is at least 2.

By putting all these ideas together we can formally express Algorithm 21 and 22 for solving Problem 2.

Algorithm comparison

We compared both algorithms in terms of runtime. For this, we generated 100 tuples (s_1, s_2, t_1, t_2) and computed disjoint sr-path using the MIP algorithm and the dedicated algorithm for $k = 3$ and $k = 4$. Figure 8.11 shows the performance profile of the algorithms for $k = 3$. A performance profile shows the CDF of the ratio of the runtime of each algorithm and the minimum runtime amongst the two. We can observe that the MIP model is faster for 64% of the tuples. The MIP model is at most 20 times slower whereas the dedicated algorithm can be up to 53 times slower. This indicates that MIP model is more efficient than the dedicated algorithm. This gets even more evident as we grow k. For $k = 4$, as shown in Figure 8.12, the MIP model performs much better than the dedicated algorithm. It is faster for 80% of the tuples and when it is not, it is barely slower.

8.4 Robustly disjoint sr-paths

In this section we propose a technique to leverage segment routing to provide a failure tolerant disjoint path service. The idea is to connect two sites via a pair of disjoint sr-paths like we did in the previous section and ensure these sites remain connected by two disjoint paths even in case of a link failures. Segment routing is interesting in such a setting because, since it is based on shortest path routing, after a set of network links goes down, these sr-paths will automatically converge towards new paths on G as the routers update theirs routing tables.

The idea is thus to take this into account when building the sr-paths and make sure that the set of paths on G that correspond the sr-paths are disjoint

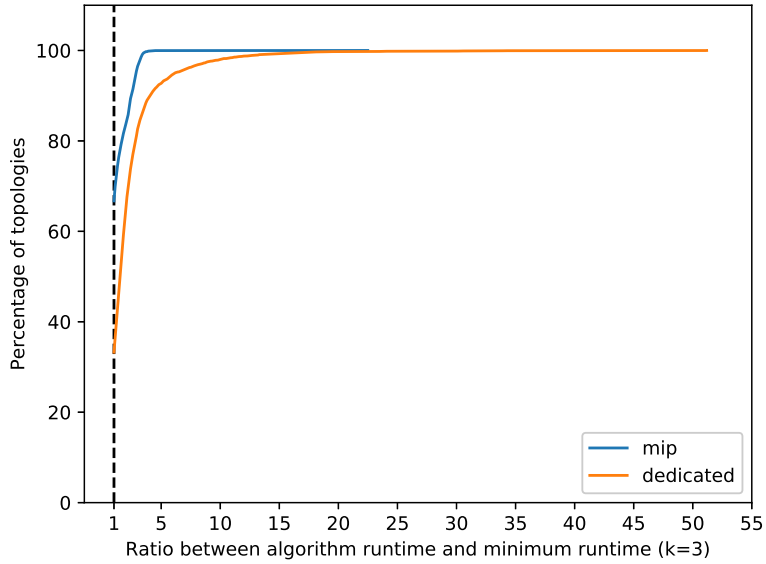


Figure 8.11: Performance profile between the MIP model and the dedicated algorithm for $k = 3$.

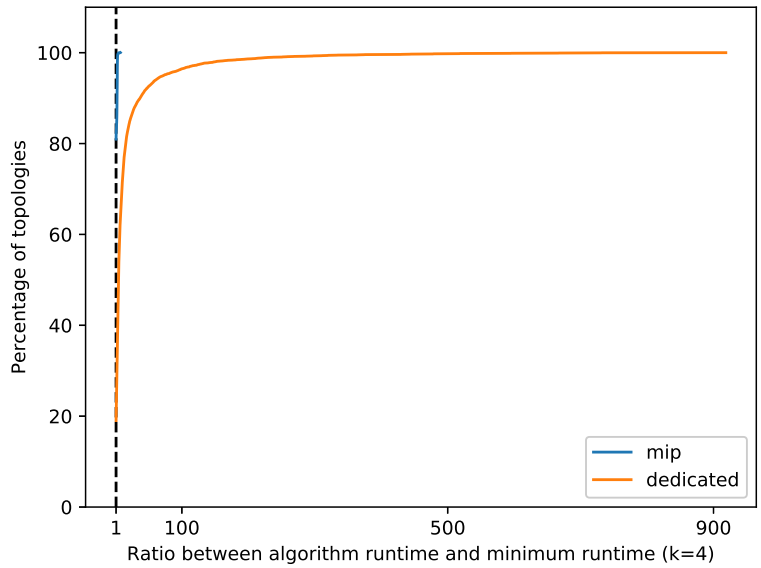


Figure 8.12: Performance profile between the MIP model and the dedicated algorithm for $k = 4$.

after IGP re-convergence for any given failure. If we manage to compute such paths, then we have the guarantee that our solution remains disjoint even in case something goes wrong.

Concretely, we will assume that a set of failure that we want to support is given as input and we will seek pairs of paths that are tolerant to any failure in the given set.

Definition 8.2. *Let* G *be a network. A* failure set *is a set* $\mathsf{F} = \{\mathtt{f}_1, \ldots, \mathtt{f}_m\}$ *such that for all* $\mathtt{i}$, $\mathtt{f}_i \subseteq \mathsf{E}(\mathsf{G})$ *and* $\emptyset \in \mathsf{F}$.

Requiring that the empty set belongs to the failure sets poses no practical restriction and is there just to make the following definitions more elegant.

If we want to support single link failures we can set $\mathsf{F} = \{\{e\} \mid e \in \mathsf{E}(\mathsf{G})\} \cup \{\emptyset\} = \mathsf{F}_{\mathsf{E}}(\mathsf{G})$. If we know specific shared risk link groups [29, 51] we can also add them to F so that our paths become failure tolerant to them. Note that there are limits to the amount of failures that one can add to F. If we add too many failure sets then we have a high chance of over constraining the problem and making it have no admissible solution.

We model a failure of a set of edges $\mathtt{f} \in \mathsf{F}$ as removing those edges from the network. Therefore it can happen that the network becomes disconnected after a failure occurs. This can lead to the fact that the sr-paths might become undefined if they either require to route traffic between disconnected parts of the network or they use an adjacency segment over an element of $\mathtt{f}$. In particular, this means that we cannot use adjacency segments over any link belonging to an edge in F since such a failure would make the path unusable. As a consequence, if $\mathsf{F} = \mathsf{F}_{\mathsf{E}}(\mathsf{G})$ then we cannot use adjacency segments at all. This leads to the following definition.

Definition 8.3. *Let* G *be a network,* $\vec{\mathtt{p}} = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle$ *a sr-path on* G *and* F *a failure set. We say that* $\vec{\mathtt{p}}$ *is* well defined with respect to F *if* $\vec{\mathtt{p}}$ *does not contain any adjacency segment belonging to a set* $\mathtt{f} \in \mathsf{F}$ *and for each* $\mathtt{i} = 2, \ldots, n$ *and* $\mathtt{f} \in \mathsf{F}$, $\mathsf{G} \setminus \mathtt{f}$ *contains at least one path from* $\mathtt{x}^2_{\mathtt{i}-1}$ *to* $\mathtt{x}^1_{\mathtt{i}}$.

Figure 8.13 illustrates this definition. Suppose that the failure set F contains a failure set $\mathtt{f}$ whose elements consist of all edges touching node $\mathtt{i}$ (in both directions). Then any path using node $\mathtt{i}$ as a segment will not be well defined with respect to F. For instance, $\vec{\mathtt{p}} = \langle \mathtt{a}, \mathtt{g}, \mathtt{i}, \mathtt{h} \rangle$ will fail to forward packets from $\mathtt{g}$ to $\mathtt{i}$ if failure $\mathtt{f}$ occurs. This example also illustrates another important aspect of robustly disjoint paths. Imagine that we want to consider node failures and also have disjoint sr-paths that are tolerant to node failures. We can model a node failure with a failure set $\mathtt{f}$ consisting of all edges incident to that node (in both directions). However, this will imply that that node becomes a forbidden node segment for any sr-path in the solution. A corollary of this is that it is impossible to have robustly disjoint paths that are tolerant to the failure of *any* node since this would prevent any sr-path to contain segments altogether.

We can now define the robustly disjoint paths (RDPs).

Definition 8.4. *Let* G *be a network,* $\vec{\mathtt{p}}_1$, $\vec{\mathtt{p}}_2$ *be two sr-paths and* F *a failure set. We say that* $\vec{\mathtt{p}}_1$ *and* $\vec{\mathtt{p}}_2$ *are* robustly disjoint *if they are disjoint and well defined on* $\mathsf{G} \setminus \mathtt{f}$ *for every* $\mathtt{f} \in \mathsf{F}$.

You can see that this definition also ensures that robuslty disjoint paths are disjoint to begin with since we assume that $\emptyset \in \mathsf{F}$.

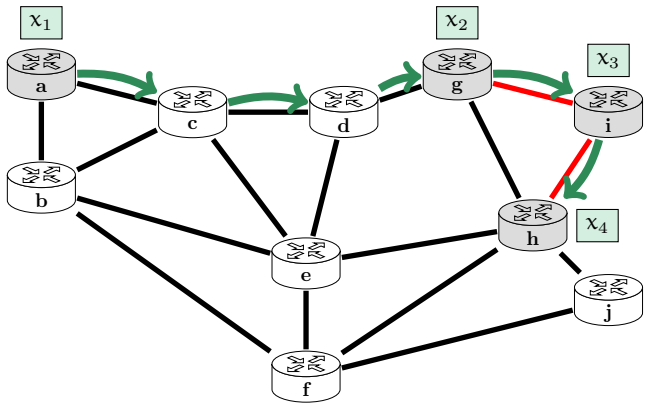


Figure 8.13: The sr-path $\langle \mathtt{a}, \mathtt{g}, \mathtt{i}, \mathtt{h} \rangle$ is not well defined if edges $(\mathtt{g}, \mathtt{i})$, $(\mathtt{i}, \mathtt{g})$, $(\mathtt{i}, \mathtt{h})$, $(\mathtt{h}, \mathtt{i})$ are in F.

Problem 3 (Robustly disjoint sr-path problem)

Input: *A network* G*, a failure set* F*,* $s_1, s_2, t_1, t_2 \in \mathsf{V}(\mathsf{G})$ *and* $k \in \mathbb{N}$*.*
Output: *Two robustly disjoint sr-paths* $\vec{p}_1 \in \vec{\mathcal{P}}_k(s_1, t_1)$*,* $\vec{p}_2 \in \vec{\mathcal{P}}_k(s_2, t_2)$ *with respect to* F*.*

Since Problem 2 is NP-hard, this problem must also be since we get the same problem by setting $\mathsf{F} = \{\emptyset\}$.

It is not hard to adapt the disjoint sr-path MIP model proposed in the previous section to the case of robustly disjoint paths as well as our dedicated algorithm as we will show in the remainder of this section.

8.4.1 Adapting SR-2EDP to RDPs

Adapting model $\mathsf{SR\text{-}2EDP}(\mathsf{G}, s_1, s_2, t_1, t_2)$ to support robustly disjoint paths is very simple. The only thing that we need to change are the disjointness constraints so that they take the failures into account.

Recall that we ensure disjointness by requiting that if $e \in \mathsf{E}(\mathsf{SP}(u, v))$ then at most one of x^1_{uv}, x^2_{uv} is set to 1. To ensure that the sr-paths are robuslty disjoint we need to ensure that if

$$e \in \bigcup_{f \in \mathsf{F}} \mathsf{E}(\mathsf{SP}(\mathsf{G} \setminus f, u, v))$$

then at most one of x^1_{uv}, x^2_{uv} is equal to 1.

For this we define a new indicating function I_F such that $I_\mathsf{F}(u, v, e) = 1$ if and only if e belongs to the shortest paths between u and v on $\mathsf{G} \setminus f$ for some $f \in \mathsf{F}$. The robuslty disjoint path model are obtained by replacing I by I_F.

The cost of this adaptation is of course that pre-computing these I_F functions takes more time, specially if F is very large. However, this is only needed to be done once. We can build the model once and only adapt it for the specific sources and destinations before each computation. Using the MIP model is also more flexible and can easily be extended to compute more than two paths.

8.4.2 Adapting the dedicated algorithm to RDPs

Algorithms 21 and 22 are also easily adaptable to the case of robuslty disjoint paths but this results in a very inefficient algorithm. The problem is that checking whether the paths are disjoint after any failure requires $|\mathsf{F}|$ bit set comparisons meaning that each step of the algorithm is $|\mathsf{F}|$ times slower. For completeness we will describe the needed modifications on the algorithms.

To ensure that the sr-paths computed by the algorithm are well defined, we need to prevent the algorithm from using consecutive node segments u, v such that there is no path from u to v in $\mathsf{G} \setminus \mathsf{f}$. Also, we need to forbid any adjacency segment over an edge belonging to some $\mathsf{f} \in \mathsf{F}$. Both these steps can be achieved by pre-computing the pairs of nodes that cannot appear as consecutive segments and preventing to use adjacency segments on the set $\{e \in \mathsf{f} \mid \mathsf{f} \in \mathsf{F}\}$. Contrary to the next modification, this only slows down the preprocessing step of the algorithm, not the algorithm itself.

What really slows down the algorithm is that we need to also ensure disjointness after any failure $\mathsf{f} \in \mathsf{F}$ occurs. To avoid having to make $|\mathsf{F}|$ shortest path computations we can pre-compute $\mathsf{SP}(\mathsf{G} \setminus \mathsf{f}, u, v)$ for all $\mathsf{f} \in \mathsf{F}$ and $u, v \in \mathsf{V}(\mathsf{G})$. Now instead of checking for intersection between $\mathsf{E}(\vec{p}_1)$ and $\mathsf{E}(\vec{p}_2)$ we need to check for intersection for every $\mathsf{f} \in \mathsf{F}$. This is the reason why the algorithm is not usable in practice for generic failures unless F is quite small. Note that this is not the case for the MIP adaptation above, we need more time to build the model because of the indicating function I_{F} takes longer to compute but that is not really a problem since it needs to be done only once for any given topology. We will see however in the next sections that if $\mathsf{F} = \mathsf{F}_{\mathsf{E}}$ (single-link failures) then we can still obtain a fast algorithm.

8.4.3 The case of single-link failures

Recall from the above discussion that the bottleneck in adapting Algorithms 21 and 22 to the case of robustly disjoint paths is checking whether the paths are disjoint for every failure. We are going to see how we can check this efficiently when the set of failures is the set of edges thus showing that computing robustly disjoint paths that are tolerant to single-link failures can be done efficiently in practice.

The idea is that we will aggregate all the edges in $\mathsf{SP}(\mathsf{G} \setminus \mathsf{f}, u, v)$ for all $\mathsf{f} \in \mathsf{F}$ into a single set that we call *failure set*. And then we show that checking for intersection between the failures sets is enough to ensure robustness when $\mathsf{F} = \mathsf{F}_{\mathsf{E}}$.

Definition 8.5. *Let* G *be a network and* $\vec{p} = \langle x_1, \ldots, x_n \rangle$ *a sr-path on* G*. We define the* failure set of $\vec{p}$ *as*

$$\mathsf{fail}(\vec{p}) = \bigcup_{\mathsf{f} \in \mathsf{F}_{\mathsf{G}}} \mathsf{E}(\mathsf{G} \setminus \mathsf{f}, \vec{p})$$

The next result is a very simple lemma that states that a failure on an edge that is not used by a sr-path does not affect it. This is quite an obvious result but it is important for the theorem that follows.

Lemma 8.2. *Let* G *be a network and* $\vec{p}$ *a sr-path on* G*. If* $e \in \mathsf{E}(\mathsf{G}) \setminus \mathsf{E}(\vec{p})$ *then* $\mathsf{E}(\mathsf{G} \setminus e, \vec{p}) = \mathsf{E}(\mathsf{G}, \vec{p})$.

Proof. Let $e \in \mathsf{E}(\mathsf{G}) \setminus \mathsf{E}(\vec{p})$ and write $\vec{p} = \langle x_1, \ldots, x_n \rangle$. Since $e \notin \mathsf{E}(\vec{p})$ we have that for each $i \in \{2, \ldots, n\}$, $\mathsf{SP}(\mathsf{G}, x_{i-1}^2, x_i^1) = \mathsf{SP}(\mathsf{G} \setminus e, x_{i-1}^2, x_i^1)$. Also, e cannot be an adjacency segment of $\vec{p}$ or else it would belong to $\mathsf{E}(\vec{p})$. Therefore,

$$\begin{aligned}\mathsf{E}(\mathsf{G} \setminus e, \vec{p}) &= \left(\bigcup_{i=2}^{n} \mathsf{E}(\mathsf{SP}(\mathsf{G} \setminus e, x_{i-1}^2, x_i^1))\right) \cup \bigcup_{i : x_i \in \mathsf{E}(\mathsf{G}) \setminus e} x_i \\ &= \left(\bigcup_{i=2}^{n} \mathsf{E}(\mathsf{SP}(\mathsf{G}, x_{i-1}^2, x_i^1))\right) \cup \bigcup_{i : x_i \in \mathsf{E}(\mathsf{G})} x_i \\ &= \mathsf{E}(\mathsf{G}, \vec{p}).\end{aligned}$$

□

The following theorem shows how we can efficiently check disjointness conditions for every failure in the case of single-link failures.

Theorem 8.3. *Let* G *be a network and* $\vec{p}_1, \vec{p}_2$ *be two sr-paths on* G*. Then* $\vec{p}_1, \vec{p}_2$ *are robustly disjoint with respect to* F_{E} *if and only if they are well defined,* $\mathsf{fail}(\vec{p}_1) \cap \mathsf{E}(\vec{p}_2) = \emptyset$ *and* $\mathsf{E}(\vec{p}_1) \cap \mathsf{fail}(\vec{p}_2) = \emptyset$*.*

Proof. ($\Rightarrow$) Suppose that $\vec{p}_1, \vec{p}_2$ are robustly disjoint. Thus for each $f \in \mathsf{F}_{\mathsf{E}}$ we have $\mathsf{E}(\mathsf{G} \setminus f, \vec{p}_1) \cap \mathsf{E}(\mathsf{G} \setminus f, \vec{p}_2) = \emptyset$. Since $\emptyset \in \mathsf{F}_{\mathsf{E}}$ we have in particular that $\vec{p}_1$ and $\vec{p}_2$ are disjoint on G. Suppose that $\mathsf{fail}(\vec{p}_1) \cap \mathsf{E}(\vec{p}_2) \neq \emptyset$. Then there exists $e \in \mathsf{E}(\mathsf{G}, \vec{p}_1)$ such that $\mathsf{E}(\mathsf{G} \setminus e, \vec{p}_1) \cap \mathsf{E}(\mathsf{G}, \vec{p}_2) \neq \emptyset$. Since the paths are disjoint, it cannot be the case that $e \in \mathsf{E}(\mathsf{G}, \vec{p}_2)$. Therefore, by Lemma 8.2 we have that $\mathsf{E}(\mathsf{G} \setminus e, \vec{p}_2) = \mathsf{E}(\mathsf{G}, \vec{p}_2)$. This means that $\mathsf{E}(\mathsf{G} \setminus e, \vec{p}_1) \cap \mathsf{E}(\mathsf{G} \setminus e, \vec{p}_2) = \mathsf{E}(\mathsf{G} \setminus e, \vec{p}_1) \cap \mathsf{E}(\mathsf{G}, \vec{p}_2) \neq \emptyset$ contradicting the fact that the sr-paths are robustly disjoint. The case where $\mathsf{E}(\vec{p}_1) \cap \mathsf{fail}(\vec{p}_2) \neq \emptyset$ is analogous.

($\Leftarrow$) Suppose that $\vec{p}_1, \vec{p}_2$ are well defined and that $\mathsf{fail}(\vec{p}_1) \cap \mathsf{E}(\vec{p}_2) = \emptyset$ and $\mathsf{E}(\vec{p}_1) \cap \mathsf{fail}(\vec{p}_2) = \emptyset$. Assume that the paths are not robustly disjoint for F_{E}. Since they are well defined, this means that either $\mathsf{E}(\vec{p}_1) \cap \mathsf{E}(\vec{p}_2) \neq \emptyset$ or there exists $e \in \mathsf{E}(\mathsf{G})$ such that $\mathsf{E}(\mathsf{G} \setminus e, \vec{p}_1) \cap \mathsf{E}(\mathsf{G} \setminus e, \vec{p}_2)$. Since $\emptyset \in \mathsf{F}_{\mathsf{G}}$ we have that $\mathsf{E}(\vec{p}_1) \subseteq \mathsf{fail}(\vec{p}_1)$ and thus $\mathsf{E}(\vec{p}_1) \cap \mathsf{E}(\vec{p}_2) \subseteq \mathsf{fail}(\vec{p}_1) \cap \mathsf{E}(\vec{p}_2) = \emptyset$. Then, let e be such that $\mathsf{E}(\mathsf{G} \setminus e, \vec{p}_1) \cap \mathsf{E}(\mathsf{G} \setminus e, \vec{p}_2) \neq \emptyset$. Since $\mathsf{E}(\vec{p}_1) \cap \mathsf{E}(\vec{p}_2) = \emptyset$, e cannot belong to both $\mathsf{E}(\vec{p}_1)$ and $\mathsf{E}(\vec{p}_2)$. If it does not belong to $\mathsf{E}(\vec{p}_1)$ then by Lemma 8.2 we have $\mathsf{E}(\mathsf{G} \setminus \mathsf{E}, \vec{p}_1) = \mathsf{E}(\vec{p}_1)$ so $\emptyset \neq \mathsf{E}(\mathsf{G} \setminus e, \vec{p}_1) \cap \mathsf{E}(\mathsf{G} \setminus e, \vec{p}_2) = \mathsf{E}(\vec{p}_1) \cap \mathsf{E}(\mathsf{G} \setminus e, \vec{p}_2) \subseteq \mathsf{E}(\vec{p}_1) \cap \mathsf{fail}(\vec{p}_2) = \emptyset$ which is a contradiction. The other case is analogous and also leads to a contradiction. It follows that the paths are robustly disjoint. □

Using Theorem 8.3 we can transfer the time it takes for checking whether paths remain disjoint to a pre-computation step where we compute the failure sets defined above for each two nodes $u, v \in \mathsf{V}(\mathsf{G})$. Using these we can incrementally build the failure sets as we build $\vec{p}_1$ by noting that if $\vec{p}_1 = \langle x_1, \ldots, x_n \rangle$ then

$$\mathsf{fail}(\vec{p}) = \left(\bigcup_{i=2}^{n} \mathsf{fail}(\langle x_{i-1}^2, x_i^1 \rangle)\right) \cup \bigcup_{i : x_i \in \mathsf{E}(\mathsf{G})} x_i.$$

Therefore, whenever we extend $\vec{p}_1 = \langle x_1, \ldots, x_n \rangle$ with a segment x, we simply need to make the union of the current failure set with the precomputed

failure set $\mathsf{fail}\langle x_n^2, x\rangle$ and x_i if x_i is an adjacency segment. Again, because of the use of bitset, this is a lightweight operation.

These only need to be pre-computed once for each given topology. We propose an efficient algorithm that is able to compute them in under a minute even on the largest topologies. Thus it is reasonable to recompute them whenever the topology changes as these do not happen every minute.

Pre-computing the failure sets

As we showed above, the failure sets for single-link failures are composed of the failure sets of the sr-path $\langle x, y\rangle$. We are now going to describe how we can compute these efficiently. If we followed the definition we would loop over all pairs $x, y \in \mathsf{V}(G)$ and edges $e \in \mathsf{E}(G)$ and for each edge we would compute the shortest path subnetwork from x to y on $G \setminus e$. However this is very costly. Instead, we can take advantage of Lemma 8.1. For each fixed x and $e \in \mathsf{E}(G)$ we will compute the shortest path subnetwork rooted at x. Then we compute a topological order of it and use Equation (8.1) to compute $\mathsf{E}(\mathsf{SP}(G \setminus e, x, y))$.

We also add two ideas to accelerate the algorithm. The first is based on Lemma 8.2. It simply consists of ignoring sets $e \notin \mathsf{E}(\mathsf{SP}(x, y))$ since these will not add any new edges to $\mathsf{fail}\langle x, y\rangle$. The second idea is that we can ignore edges that belong to an ECMP component between x and y since removing them will also not contribute to any new shortest paths because there will always be at least one of the previous shortest paths that remains after the removal of e. A formalization of this is provided as Algorithm 23.

8.4.4 Evaluation of RDPs

Evaluating effective robustness of RDP

Using the above concepts we can compute pairs of sr-paths that are robustly disjoint to single-link failures. This is what our theoretical results guarantee. However, it could very well be the case that our paths are actually robust to a lot more failures. We performed an experiment to evaluate this. For $s = 1, \ldots, 6$, we generated 100 sets of s simultaneous link failures. We consider two kinds of failures sets: selecting random links from the whole topology and selecting random links that are used by the sr-paths. These second failure sets are generated so that each of them affects the paths. So for instance, on a failure set with $s = 3$ we select the first link to be one of the links used by our pair of sr-paths. Then compute the paths resulting from that failure and select the second link randomly among the resulting links. For the third link we do the same. This ensures that we are not biasing our results by the fact that with high probability selecting a random link will not even affect the paths. In general, the n-th failure is selected from the paths resulting from the previous $n - 1$ failures. For each failure set we evaluate the end state of the paths and consider the following four states:

i) *disjoint*: the paths remain disjoint after the failures;

ii) *not disjoint*: both paths remain defined but not disjoint;

iii) *one disconnected*: one path becomes disconnected;

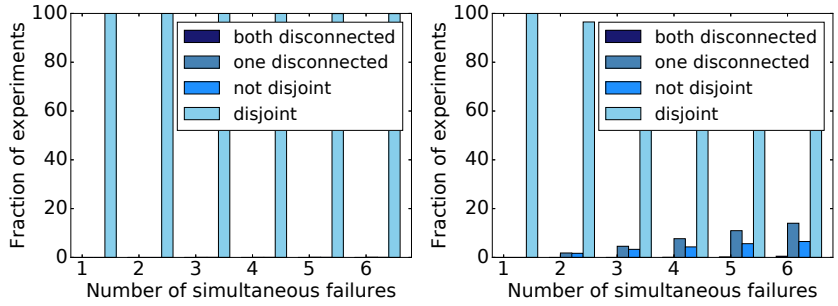


Figure 8.14: Random failures and path link failures over the worst case topoology.

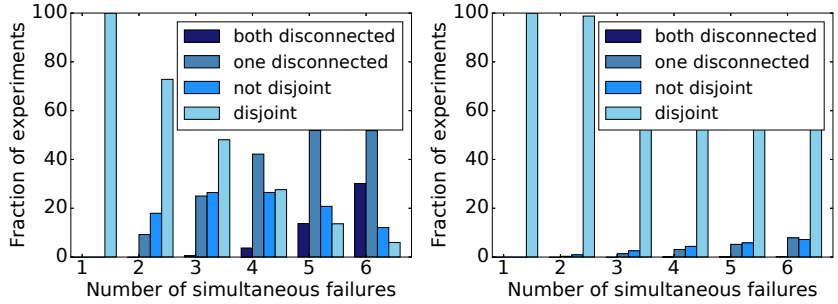


Figure 8.15: Random failures and path link failures over the best case topology.

iv) *both disconnected*: both paths became disconnected.

Figures 8.14 and 8.15 show the results on the topologies in our dataset for which we respectively get the best and worst results in terms of robustness to additional failures (all the other topologies are included between those extremes). For the best case, shown in Figure 8.14 paths remain disjoint with no configuration change in almost all the simulations, including those with 6 simultaneous link failures, and source-destination tuples are disconnected very rarely (0.01% of the time for 6 link failures). For the wost case, shown in Figure 8.15, results remain very good for random failures, but are significantly worse for the worst-case failures. Still, connectivity is often kept between source-destination tuples, e.g., for more than 80% (about 70%, respectively) of the experiments in the presence of 5 (6, respectively) successive on-path failures.

We wondered whether we could evaluate the robustness of a pair of sr-paths algorithmically rather than having to perform such an experiment. The following theorem shows that it is NP-hard to compute the minimum number of failures that a pair of sr-paths can withstand until they cease to be disjoint.

Problem 4 (Minimum cardinality failure)

Input: *A network* G *and two sr-paths* $\vec{\mathsf{p}}_1$ *and* $\vec{\mathsf{p}}_2$.
Output: *The cardinality of a minimal set of links* $\mathsf{f} \subseteq \mathsf{E}(\mathsf{G})$ *such that* $\vec{\mathsf{p}}_1$ *and*

$\vec{p}_2$ *are not disjoint on* $G \setminus f$.

Theorem 8.4. *Problem 4 is NP-hard.*

The following proof resulted from discussions with a student of mine, Simon Tihon, while I was coaching him for algorithmic programming contents.

Proof. To prove that this problem is NP-hard it is enough to prove that the problem is NP-hard for sr-paths of the form $\vec{p}_1 = \langle s_1, t_1 \rangle$ and $\vec{p}_2 = \langle s_2, t_2 \rangle$. This amounts to, given four nodes s_1, s_2, t_1 and t_2, find the minimum numbers of edges that we need to remove so that the shortest paths from s_1 to t_1 intersect the shortest paths from s_2 to t_2.

It is known that the problem of finding the minimum number of edges that need to be removed so that the shortest path between two given nodes becomes strictly larger than a given value d is NP-hard [31]. Let G, s, t, d be an instance of this problem and assume that we can solve our problem is polynomial time. We can assume that the shortest path between s and t has cost lower than or equal to d or otherwise the problem is trivial. e build an instance of Problem 4 by setting $s_2 = s, t_2 = t$ and adding two nodes s_1, t_1. We connect s_1 to s_2 with an link of weight d and t_1 to t_2 with a link of weight 0. Nodes s_1 and t_1 are connected with $K + 1$ parallel edges of weight 0 where K is the value of the minimum cut between s and t on G. Figure 8.16 illustrates this construction.

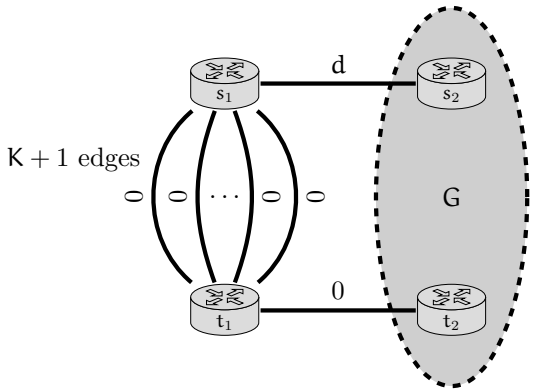


Figure 8.16: Construction used in the problem reduction.

The shortest paths between s_1 and t_1 consists of the parallel links between them whereas the shortest paths between $s_2 = s$ and $t_2 = t$ lie on G since we assumed that the shortest path from s to t has a cost lower than d. Note that the path visiting nodes (s_2, s_1, t_1, t_2) has cost d. Therefore, the shortest path from s_2 to t_2 will intersect the shortest paths from s_1 to t_1 if and only if the shortest path from s_1 to t_1 on G costs more than d. Clearly, the minimum number of edges that we need to remove so that the cost of the shortest path from s to t becomes at least d is at most K since K is the value of a minimum cut (and thus a solution). Thus after removing this set the path are still well defined since we have $K + 1$ parallel edges. □

Evaluating existence and quality of RDPs

In this section we will use the word detour to refer to an intermediate node that is in the segment stack. More concretely, for a sr-path of the form $\langle x_1, x_2, \ldots, x_{n-1}, x_n \rangle$, the detours are the nodes $x_2, \ldots, x_{n-1}$. Clearly a sr-path with r detours has segment cost $r + 2$.

We focus on reasonably well-connected *source-destination tuples*. For each topology, we randomly select 100 tuples (s_1, s_2, t_1, t_2) of two sources s_1, s_2 and two destinations t_1, t_2, such that s_1 and s_2 have a path to t_1 and t_2 even when any edge is removed. Since we try to compute robustly disjoint paths from s_1 to t_1 and from s_2 to t_2, it would indeed make little sense to consider source-destination pairs that are disconnected by a single failure – it is obvious that a service provider cannot offer a robust connectivity service between routers that are poorly connected. We repeat each experiment allowing between 1 and 3 detours ($k = 3, 4, 5$). We stop at 3.

Table 8.17 shows the percentage of these tuples for which robustly disjoint paths exist. Sometimes, only selecting the right IGP paths is sufficient for a given tuple. However, since IGP costs are shared across all paths, they rarely can be used for more than one source-destination tuple, preventing operators to configure robustly disjoint paths for multiple customers or between different sites of the same customer. Adding one detour by specifying an intermediate node with SR allows paths for different tuples to be independent from each other, solving the above issue. It also drastically increases the percentage of tuples with at least one pair of robustly disjoint paths to 71across all the topologies, and to 97two. Allowing more detours provides only slightly more flexibility in our experiments.

Our algorithms are designed to find sr-paths that are both robustly disjoint and have minimal worst-path delay. As table 8.18 shows, the robustly disjoint paths computed by our algorithms have a worst-path delay which is always better than the worst latency across the original IGP shortest paths. We are up to 15% more efficient, on average. Once again, more detours enable to decrease the latency of the computed paths across all the topologies, but just negligibly in most cases.

To assess the benefits of robustly disjoint paths in a real-life scenario, we also analyse a 1-week trace of all the link-state IGP packets exchanged by a router in Real ISP2. Based on this trace, we identified that a total of 5% of the links failed during this period. Some links experienced flapping, confirming observations of previous studies [28, 45]. For example, one of the links failed more than 30 times during the analysed week. We select 100 source-destination pairs in this network, and compute the corresponding robustly disjoint paths for $F = E$ (all single-link failures). We then replayed all the failures that happened during the entire week. The source-destination pairs always have disjoint paths in our simulation, at any moment during the week, even when multiple edges failed simultaneously. This experiment provides a strong indication that the paths computed by our algorithms are robust to real failures, for a long time, in an operational network, without the need for any configuration adjustment.

Topology	Number of detours 0 det	1 det	2 det	3 det
Real ISP 1	83%	100%	100%	100%
Real ISP 2	89%	100%	100%	100%
Real ISP 3	73%	100%	100%	100%
AS 1221	82%	98%	100%	100%
AS 1239	90%	100%	100%	100%
AS 1755	52%	98%	100%	100%
AS 3257	76%	100%	100%	100%
AS 3967	71%	99%	100%	100%
AS 6461	75%	100%	100%	100%
ITZ Cogentco	78%	97%	100%	100%
ITZ Colt	58%	71%	73%	73%
ITZ Deltacom	74%	99%	99%	100%
ITZ Dia	54%	77%	79%	79%
ITZ GtsCe	78%	98%	100%	100%
ITZ Interoute	81%	99%	100%	100%
ITZ Ion	64%	100%	100%	100%
ITZ Tata	86%	100%	100%	100%
ITZ UsCarrier	72%	83%	85%	85%

Figure 8.17: Percentage of tuples for which RDPs exist.

Topology	Number of detours 1 det	2 det	3 det
Real ISP 1	0.97	0.97	0.97
Real ISP 2	0.98	0.98	0.98
Real ISP 3	0.97	0.96	0.96
AS 1221	0.99	0.99	0.99
AS 1239	0.97	0.97	0.97
AS 1755	0.90	0.89	0.88
AS 3257	0.91	0.89	0.88
AS 3967	0.97	0.97	0.97
AS 6461	0.97	0.97	0.97
ITZ Cogentco	0.85	0.84	0.84
ITZ Colt	0.88	0.87	0.86
ITZ Deltacom	0.91	0.90	0.90
ITZ Dia	0.96	0.98	0.98
ITZ GtsCe	0.78	0.77	0.75
ITZ Interoute	0.93	0.91	0.90
ITZ Ion	0.95	0.94	0.94
ITZ Tata	0.90	0.89	0.89
ITZ UsCarrier	0.92	0.92	0.92

Figure 8.18: Average ratio between the RPD latency and the nominal latency.

Algorithm 22 disjoint-srpaths-dfs $(\vec{p}_1, \vec{p}_2)$

if $\max(\mathsf{lat}(\vec{p}_1) + \mathcal{L}(\vec{p}_1.\mathsf{dest}(), t_1) \geqslant l^*$ **then**
  return
[we reached here so if the path is complete, it is a better solution]
if $\vec{p}_1.\mathsf{dest}() = t_1$ **then**
  $l^* \leftarrow \max(\mathsf{lat}(\vec{p}_1), \mathsf{lat}(\vec{p}_2))$
  $\vec{p}_1^*, \vec{p}_2^* \leftarrow \vec{p}_1, \vec{p}_2$
  return
[try extend $\vec{p}_1$ with a node segment]
if $\mathsf{sr\text{-}cost}(\vec{p}_1) + 1 > k$ **then**
  return
for $u \in V(G)$ **do**
  [check whether we can cut with min cost flow]
  $P, l \leftarrow \mathsf{min\text{-}cost\text{-}flow}(g, \{u, s_2\}, \{t_1, t_2\})$
  if $|P| < 2$ **or** $l \geqslant l^*$ **then**
    return
  [check whether adding u will keep $\vec{p}_1$ acyclic]
  if $E(\mathsf{SP}(\vec{p}_1.\mathsf{dest}(), u)) \cap E(\vec{p}_1) = \emptyset$ **then**
    return
  $\vec{p}_1.\mathsf{addLast}(u)$
  if $E(\vec{p}_1) \cap E(\vec{p}_2) = \emptyset$ **then**
    disjoint-srpaths-dfs $(\vec{p}_1, \vec{p}_2)$
  else
    $\vec{p}_2' \leftarrow \mathsf{min\text{-}lat\text{-}disjoint\text{-}srpath}(\vec{p}_1, s_2, t_2, k)$
    if $\vec{p}_2' \neq$ **null then**
      disjoint-srpaths-dfs $(\vec{p}_1, \vec{p}_2')$
  $\vec{p}_1.\mathsf{removeLast}()$
[try extend $\vec{p}_1$ with an adjacency segment]
if $\mathsf{sr\text{-}cost}(\vec{p}_1) + 2 > k$ **then**
  return
for $e \in E(g)$ **do**
  [check whether we can cut with min cost flow]
  $P, l \leftarrow \mathsf{min\text{-}cost\text{-}flow}(g, \{e^2, s_2\}, \{t_1, t_2\})$
  if $|P| < 2$ **or** $l \geqslant l^*$ **then**
    return
  if $(E(\mathsf{SP}(\vec{p}_1.\mathsf{dest}(), e^1)) \cup \{e\}) \cap E(\vec{p}_1) \neq \emptyset$ **then**
    return
  [check whether adding e will keep $\vec{p}_1$ acyclic]
  $\vec{p}_1.\mathsf{addLast}(e)$
  if $E(\vec{p}_1) \cap E(\vec{p}_2) = \emptyset$ **then**
    disjoint-srpaths-dfs $(\vec{p}_1, \vec{p}_2)$
  else
    $\vec{p}_2' \leftarrow \mathsf{min\text{-}lat\text{-}disjoint\text{-}srpath}(\vec{p}_1, s_2, t_2, k)$
    if $\vec{p}_2' \neq$ **null then**
      disjoint-srpaths-dfs $(\vec{p}_1, \vec{p}_2')$

Algorithm 23 precompute-failEdges (G, SP, fwe)

```
1: for u, v ∈ V(G) do
2:     fail(u, v) ← fwe(u, v)
3: for u ∈ V(G) do
4:     [optim 1: only consider edges in SP(u) as the others have no effect]
5:     for f ∈ E(SP(u)) do
6:         [optim 2: check degree to consider only edges not in ECMP components]
7:         if δ⁻(SP(u), f²) ⩽ 1 then
8:             SP(f, u) ← dikstra-dag(G \ f, u)
9:             order ← toposort(SP(u))
10:            for v ∈ order do
11:                for e ∈ δ⁻(SP(u), v) do
12:                    fail(u, v) ← fail(u, e¹) ∪ {e}
13: return fail
```

Chapter 9

Conclusion

In this thesis we proposed a first mathematical formalization of segment routing. So far, segment routing had been used to solve some networking problems such as traffic engineering but the further that these results would go in terms of formalization was showing how to model segment routing in terms of linear constraints.

We go a step further and show interesting results about the structure of these segment routing paths. We showed that these results have practical applications by exploiting them to solve several algorithmic problems related to segment routing.

The most important results were:

- Minimal segmentations can be computed in polynomial time. This result opens the door to solving segment routing problems by ignoring segment routing constraints and solving the problem as a graph problem and only segmenting the resulting paths in the end. This is, of course, not always applicable in practice as solving problem in this way yields results that are sub-optimal in terms of the number of segments used in the final solution. Nevertheless, as the routers' capacity to support segments increases, we believe that this will become the standard approach.
- We can always connect two connected routers with a acyclic sr-path. This result can be leveraged to prune cyclic solutions on problems where we can prove that a cyclic sr-path will always be sub-optimal.
- In some applications, such as network monitoring, we need to be know exactly which network links are used to forward traffic. We defined a notion of determinism for sr-paths that expresses this requirement. We provided bound on the minimum segment cost to connect two nodes of any given network with deterministic sr-paths.
- Computing a cycle cover of a graph that uses a minimal amount of segments to represent the cycles can be achieve in polynomial time.
- Computing minimum cost sr-paths can be done in polynomial time. The algorithm can be leveraged to compute minimum latency sr-paths and maximum capacity sr-paths.

Apart from this more theoretical aspect, we have also showed how to exploit segment routing on several real world applications.

- We showed that we can leverage segmenting for network monitoring of single link failures. We proposed an algorithm that yields a solution with minimal segment cost. We also extended that solution using column generation in order to try to reduce the number probes that are necessary for monitoring.
- We propose an alternative solution for the traffic engineering problem with segment routing based on column generation. Our solution provided near optimal solutions that run faster than previous linear programming models. It also provides a lower bound so we can evaluate how good the solution is.
- Robust and low latency connections are important for applications that require low latency. We showed how we can compute and implement disjoint path using segment routing. We also improved on this solution by exploiting properties of segment routing paths for providing failure tolerance guarantees to our solution.

While writing this thesis we also left some open problems which show that there is still a lot to be done when is comes to segment routing. Our results show that the deployment of segment routing can lead to improved communications with low overhead.

- We showed that we can compute a sr-cycle cover of a network in polynomial time but our complexity is still quite high. How fast can we solve this problem?
- Suppose that we know a set of paths that we want to implement on a network. How to select IGP weights so that the maximum cost of the minimal segmentation of those paths is as small as possible?
- How can we find IGP weights that are ECMP-free and complete and no weight is above a given constant? Could these weights we a solution to the previous problem?
- We also proposed several column generation models but we did not solve them to optimality by doing, for instance, a branch-and-price. How efficiently can this be achieved?

Bibliography

[1]

[2] Ovh europe.

[3] I. Adler, M. G. C. Resende, G. Veiga, and N. Karmarkar. An implementation of karmarkar's algorithm for linear programming. *Mathematical Programming*, 44(1):297–335, May 1989.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[5] A. Altın, B. Fortz, M. Thorup, and H. Ümit. Intra-domain traffic engineering with shortest path routing protocols. *4OR*, 7(4):301, Dec 2009.

[6] F. Aubry, D. Lebrun, Y. Deville, and O. Bonaventure. Traffic duplication through segmentable disjoint paths. In *2015 IFIP Networking Conference (IFIP Networking)*, pages 1–9, May 2015.

[7] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure. Scmon: Leveraging segment routing to improve network monitoring. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[8] F. Aubry, S. Vissicchio, O. Bonaventure, and Y. Deville. Robustly disjoint paths with segment routing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 204–216, New York, NY, USA, 2018. ACM.

[9] D. O. Awduche, L. Berger, D.-H. Gan, D. T. Li, D. V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209, 2001.

[10] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[11] R. Bhatia, F. Hao, M. Kodialam, and T. V. Lakshman. Optimized network traffic engineering using segment routing. In *IEEE INFOCOM 2015*, April 2015.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] G. B. DANTZIG. *Linear Programming and Extensions*. Princeton University Press, 1991.

[14] J. Davidson. Simplifying Networks through Segment Routing. https://blogs.cisco.com/news/simplifying-networks-through-segment-routing.

[15] G. Desaulniers, J. Desrosiers, and M. M. Solomon. *Column generation*, volume 5. Springer Science & Business Media, 2006.

[16] Y. Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 01 1970.

[17] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, Apr. 1972.

[18] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 184–193, Oct 1975.

[19] G. Fan. Integer flows and cycle covers. *Journal of Combinatorial Theory, Series B*, 54(1):113 – 122, 1992.

[20] A. Farrel, O. Komolafe, and S. Yasukawa. An analysis of scaling issues in mpls-te core networks. IETF RFC5439, 2009.

[21] C. Filsfils, F. Clad, et al. IPv6 Segment Routing. In *SIGCOMM'17 demo*, August 2017.

[22] C. Filsfils et al. Segment Routing Architecture. RFC 8402, RFC Editor, July 2018.

[23] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois. The segment routing architecture. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2015.

[24] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois. The segment routing architecture. In *GLOBECOM, 2015 IEEE*. IEEE, 2015.

[25] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional ip routing protocols. *IEEE Communications Magazine*, 40(10):118–124, Oct 2002.

[26] S. Gay, R. Hartert, and S. Vissicchio. Expect the unexpected: Sub-second optimization for segment routing. In *IEEE INFOCOM 2017*, 2017.

[27] S. Gay, P. Schaus, and S. Vissicchio. REPETITA: Repeatable Experiments for Performance Evaluation of Traffic-Engineering Algorithms. *arXiv preprint arXiv:1710.08665*, 2017.

[28] L. D. Ghein. *MPLS Fundamentals*. Cisco Press, 1st edition, 2006.

[29] M. Ghobadi and R. Mahajan. Optical Layer Failures in a Large Backbone. In *IMC*, 2016.

[30] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, Oct. 1988.

[31] P. A. Golovach and D. M. Thilikos. Paths of bounded length and their cuts: Parameterized complexity and algorithms. *Discrete Optimization*, 8:72–86, 2011.

[32] R. Hartert. Fast and scalable optimization for segment routing, 2018.

[33] R. Hartert et al. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *SIGCOMM*, 2015.

[34] R. Hartert, P. Schaus, S. Vissicchio, and O. Bonaventure. Solving segment routing problems with hybrid constraint programming techniques. In *CP 2015*. Springer, 2015.

[35] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. *SIGCOMM CCR*, 45(4), Aug. 2015.

[36] S. Jain et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM CCR*, volume 43. ACM, 2013.

[37] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765 –1775, october 2011.

[38] C. Li et al. The complexity of finding two disjoint paths with min-max objective function. *Discrete Appl. Math.*, 26(1):105 – 115, 1990.

[39] C. Li et al. The complexity of finding two disjoint paths with min-max objective function. *Discrete Appl. Math.*, 26(1):105 – 115, 1990.

[40] P. S. O. B. Mathieu Jadin, François Aubry. CG4SR: Near Optimal Traffic Engineering for Segment Routing with Column Generation. In *INFOCOM*, 2019.

[41] J. Matouek and B. Gärtner. *Understanding and Using Linear Programming (Universitext)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[42] NANOG mailing list. Bell outage. https://mailman.nanog.org/pipermail/nanog/2017-August/091828.html, 2017.

[43] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz. Latency Inflation with MPLS-based Traffic Engineering. In *IMC*, 2011.

[44] M. Roughan. Simplifying the synthesis of Internet traffic matrices. *ACM SIGCOMM CCR*, 35(5), 2005.

[45] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.

[46] T. Schuller, N. Aschenbruck, M. Chimani, M. Horneffer, and S. Schnitter. Traffic engineering using segment routing and considering requirements of a carrier IP network. In *Networking*, 2017.

[47] N. Spring et al. Measuring ISP topologies with Rocketfuel. *ACM SIGCOMM CCR*, 32(4), 2002.

[48] R. Steenbergen. MPLS Autobandwidth. RIPE 64 presentation, 2012.

[49] J. W. Suurballe and R. E. Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984.

[50] J. Tantsura. The critical role of maximum sid depth (msd) hardware limitations in segment routing ecosystem and how to work around those. In *NANOG71*, October 2017. `https://pc.nanog.org/static/published/meetings//NANOG71/daily/day_5.html#talk_1424`.

[51] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: understanding the causes and impact of network failures. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 315–326. ACM, 2010.

[52] S. Vissicchio, L. Vanbever, and J. Rexford. Sweet little lies: Fake topologies for flexible routing. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 3:1–3:7, New York, NY, USA, 2014. ACM.

[53] J. Vygen. Np-completeness of some edge-disjoint paths problems. *Discrete Applied Mathematics*, 61(1):83 – 90, 1995.

[54] R. White and D. Donohue. *Art of Network Architecture, The: Business-Driven Design*. Cisco Press, 2014.

[55] L. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998.