View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UCL Discovery

Robustly Disjoint Paths with Segment Routing

François Aubry
UCLouvain
f.aubry@uclouvain.be

Stefano Vissicchio
University College London
stefano.vissicchio@cs.ucl.ac.uk

Olivier Bonaventure
UCLouvain
olivier.bonaventure@uclouvain.be

Yves Deville
UCLouvain
yves.deville@uclouvain.be

ABSTRACT

Motivated by conversations with operators and by possibilities to unlock future Internet-based applications, we study how to enable Internet Service Providers (ISPs) to reliably offer connectivity through disjoint paths as an advanced, value-added service. As ISPs are increasingly deploying Segment Routing (SR), we focus on implementing such service with SR. We introduce the concept of *robustly disjoint paths*, pairs of paths that are constructed to remain disjoint even after an input set of failures, with no external intervention (e.g., configuration change). We extend the routing theory, study the problem complexity, and design efficient algorithms to automatically compute SR-based robustly disjoint paths. Our algorithms enable a fully automated approach to offer the disjoint-path connectivity, based on configuration synthesis. Our evaluation on real topologies shows that such an approach is practical, and scales to large ISP networks.

CCS CONCEPTS

• **Networks** → **Network control algorithms**; **Traffic engineering algorithms**; • **Theory of computation** → *Routing and network design problems*;

ACM Reference Format:

François Aubry, Stefano Vissicchio, Olivier Bonaventure, and Yves Deville. 2018. Robustly Disjoint Paths with Segment Routing. In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281424

1 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 [30]. 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 [46] 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 [5], 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 [36], its sub-optimal usage of resources [31] and its scalability challenges with respect to the routers' state [13, 21].

Looking at other ISPs, our operators were instead considering Segment Routing (SR). SR [14] is a recent architecture that several service providers already deployed [29, 40, 43]. It is a source routing technology enabling to force packets via a sequence of nodes, called *segments*, by adding information about those nodes inside the packet headers. The actual paths followed by packets are then determined by the concatenation of the shortest paths from each segment to the following one. These shortest paths are computed by an underlying IGP, and updated by the IGP itself after failures.

In this paper, we tackle the problem of configuring disjoint paths with Segment Routing. We consider such problem jointly with practical operational requirements indicated by our reference ISP.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT '18, December 4–7, 2018, Heraklion, Greece
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6080-7/18/12...$15.00
https://doi.org/10.1145/3281411.3281424

The first and foremost of those requirements is to *automate* the operation of the disjoint-path service. We investigate an approach based on configuration synthesis, where a compiler computes pairs of robustly disjoint paths for input pairs of source and destination routers connecting a customer. We define *robustly disjoint paths* as pairs of paths that are disjoint in the presence and absence of any link failure [1] in an operator-provided set (e.g., for classes of more frequent failures like all single-link failures [28, 45], or for specific shared risk link groups [17, 41]). Robustly disjoint paths exploit the fact that the same sequence of segments generally maps to different paths (after IGP re-convergence) if the previous ones are disrupted by a failure. They enable to minimize the involvement of human operators and possible centralized network controllers, while also maximizing the likelihood that expected or dangerous failures do not disrupt the disjoint-path service.

A second requirement that we consider is to limit the number of segments added to data-plane packets when implementing robustly disjoint paths. This is both to limit the data-plane overhead due to SR segments in packet headers, and to comply with hardware limitations of commercial routers [38]. We design a compiler based on algorithms that are parametric with respect to the maximum number of segments that can be used for a single path.

Our third requirement is that robustly disjoint paths do not significantly degrade data-plane performance. Since they implement an advanced, likely expensive service, we assume that robustly disjoint paths are used for a relatively low amount of traffic, and hence do not highly contribute to link utilization or significantly interfere with the ISP's traffic engineering. Yet, we would like the installed paths to provide good data-plane performance, at least when there are no failures. Our algorithms therefore calculate robustly disjoint aiming at minimizing the worst-path latency. Low-delay robustly disjoint paths provide support for a range of cutting-edge (and future) applications. For example, they (i) improve performance of multipath transport protocols, for example avoiding MPTCP [32] sub-flows to self-congest; (ii) offer theoretical guarantees against security attacks [11]; and (iii) enable latency-sensitive applications, like telesurgery [3]. We note that this is achieved by the means of an objective function, that can be easily adapted to other goals – e.g., optimizing bandwidth instead of or in addition to delay.

We did not find any prior work aiming at keeping paths disjoint after failures, with or without SR. Our previous work [3] is the closest to this approach. While using SR to implement disjoint paths, it however provides no guarantee in the case of a failure, and did not minimize path delays. Other contributions that do consider post-failure conditions, with and without SR, focus on connectivity preservation (e.g., [2, 15, 27]), congestion avoidance (e.g., [26]) or compliance with BGP policies (e.g., [6]).

To design our solution, we face efficiency and scalability challenges. In particular, computing robustly disjoint paths that minimize delay implies considering (at least implicitly) all the possible pairs of paths and their modification in all the considered failure scenarios. We note that the basic problem of minimizing the worst path delay over disjoint paths is strongly NP-complete even without considering failures [25].

[1] Note that we can simulate node failures by a convenient set of link failures.

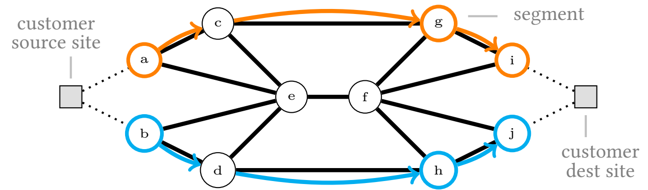


Figure 1: Example of robustly disjoint paths installed by our solution. All IGP link cost are 1. The shown segments ensure that the paths from a to i and from b to j remain disjoint after any single link failure.

We address those challenges with theoretical and algorithmic contributions. We formalize the problem and discover, perhaps surprisingly, that minimizing the worst delay across a pair of robustly disjoint paths becomes computationally tractable when such paths are implemented with SR and the maximum number of segments is fixed (Sec. 3). We also formulate a necessary and sufficient condition for paths to be robustly disjoint upon an input set of failures (e.g., the most common ones, specified by operators) (Sec. 3). We use this condition to design algorithms that efficiently find low-delay robustly disjoint paths by pruning the search space (Sec. 4).

We evaluate our approach by performing extensive simulations (Sec. 5) on 3 large ISPs, the Rocketfuel topologies [35] and the largest topologies in the Internet Topology Zoo [24]. Since we have no way to guess shared-risk link groups in our topologies, we used our algorithms to compute paths that are robustly disjoint for any single-link, random 3-link and any 2-link failure scenarios. Obviously, our approach does not scale indefinitely, and our experiments on all 2-link failures already start to expose some of its limitations both in terms of supported source-destination pairs and computation time. Yet, our algorithms compute low-delay robustly disjoint paths, using a few segments, in (sub-)seconds in most of our experiments. Also, the computed paths tolerate a much larger set of failures than those for which they are guaranteed to remain disjoint. Overall, our evaluation suggests that our approach would enable real ISPs to offer disjoint-path connectivity robustly with respect to the most common failures.

2 USING SEGMENT ROUTING TO KEEP PATHS DISJOINT

Consider the network shown in Fig. 1, where circles represent the routers (or *nodes*), and the lines between them map to their physical interconnections (or *edges*). Suppose that the network is owned by a connectivity provider (e.g. an ISP). Its customers (e.g. large enterprise networks) send traffic between their sites, geographically distributed locations that are connected to some nodes of the provider's network. The squares in Fig. 1 represent two sites of a customer requiring disjoint paths from its left site to its right one.

The provider's operators face two technical problems to offer connectivity via disjoint paths to their customers. First, they have to compute and configure (manually or automatically) two disjoint paths between the customer's sites. Second, they have to repair (manually or automatically) the disjoint paths when failures disrupt one of the configured paths.

To address those problems, the operators cannot exclusively rely on the Interior Gateway Protocols (IGPs) that provide connectivity between the provider's nodes. Those protocols allow the nodes to share a network graph, weighted on the edges, and to forward packets over the shortest paths in this graph. More than one path can be used to forward traffic from a source to a destination, using a feature called Equal Cost Multi-Path, or ECMP. With ECMP, traffic is balanced across all the shortest path from any two nodes, applying a hash function on a per-packet basis. Despite that edge weights can be tweaked to influence the shortest paths, guaranteeing that the paths computed by an IGP are disjoint is hard, or generally impossible for more than one source-destination pair (e.g. multiple customers or customers' sites).

Our solution exploits Segment Routing (SR), a source routing protocol running on top of an IGP. SR is used to automatically compute paths that are disjoint between two input customer's sites and remain so, after the convergence of the IGP, for any failure in a set specified by the operators. We now give an intuition of how SR is used in our solution, using the example in Fig. 1 and supposing that the provider intends to configure two disjoint paths, one from a to i and the other from b to j.

In a nutshell, SR enables to add lists of segments $s_1, \ldots, s_k$ to packets. Each segment s_i represents a node that has to be visited before the packet reaches the destination. Packets are forwarded over the IGP paths from each segment to the next one. For instance, the disjoint paths in Fig. 1 are implemented by instructing a and b to respectively add the segment lists $\langle a \to g \to i \rangle$ and $\langle b \to h \to j \rangle$ to packets for i and j. Because of this SR configuration, packets received by a and destined to i first follow the shortest path from a to g, that is (a, c, g), and then the shortest path from g to i, which is (g, i). The segment list $\langle b \to h \to j \rangle$ analogously results into the path (b, d, h, j). Note that for ease of notation, we always include the initial and final nodes of a path in the segment list.

Our solution exploits SR's ability to **implicitly specify backup paths**. When a link fails, the same list of segments often translates into different paths (around the failure) as a consequence of the re-computation of the IGP shortest paths. Our compiler computes convenient segment lists enforcing the resulting paths to remain disjoint. As an illustration, the segment list used in Fig. 1 ensures that the paths from a and b to respectively i and j remain disjoint after any single link failure. For example, if the link (c, g) fails, the segment list $\langle a, g, i \rangle$ translates into the path (a, e, f, g, i), which is still disjoint from the non-affected path (b, d, h, j).

Our approach is proactive, fast, and requires no external intervention. The computed segment lists are used during normal operation. When one of the input failure case occurs, no changes are applied to any segment list, nor to any node configuration: the IGP automatically re-converges to a new pair of disjoint paths, in sub-second time even in large networks, according to prior studies [16]. SR also allows us to define a segment list for each source-destination pair, and hence avoid the main limitations of an IGP-only approach (i.e. dependency between source-destination pairs).

3 ROBUSTLY DISJOINT PATHS THEORY

We model the network as a doubly weighted directed graph $G = (V, E, igp, lat)$ where $igp : E \to \mathbb{Z}^+$ such that $igp(x, y)$ denotes the IGP weight configured on link (x, y) and $lat : E \to \mathbb{R}^+$ represents the link latencies in milliseconds. As often network topologies have parallel links, we model this by assuming that the network is preprocessed so that each edge is given a unique id identifying it. So that, for instance, two edges from node x to node y are represented by $(x, y, 1)$ and $(x, y, 2)$. This avoids using the concept of multigraph and the messy set operations that come with it.

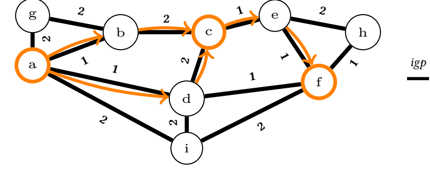


Figure 2: Illustration of the sr-path $\langle \mathrm{a} \to \mathrm{c} \to \mathrm{f} \rangle$.

Throughout this paper, we denote the size of V by n and the size of E by m. We write the in and out neighbours of a node $x \in V(G)$ as $N^-(G, x) = \{y \in V(G) \mid (y, x) \in E(G)\}$ and $N^+(G, x) = \{y \in V(G) \mid (x, y) \in E(G)\}$, respectively.

We define the following set operations on graphs. For any subset of edges $f \subseteq E(G)$, $G \setminus f$ represents the graph that we obtain from removing all edges in f from G, $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$ and $G_1 \cap G_2 = (V(G_1) \cap V(G_2), E(G_1) \cap E(G_2))$. We write $G = \emptyset$ if $E(G) = \emptyset$.

We denote the weight of a path $p = (v_1, \ldots, v_r)$ in G by $igp(p) = \sum_{i=1}^{r-1} igp(v_i, v_{i+1})$. A shortest path between two nodes s and t is a path p from s to t of minimum weight, that is, a path minimizing $igp(p)$. We denote the directed acyclic graph (DAG) formed by all the shortest paths from a given node $x \in V(G)$ by $SP(x)$. The shortest paths between two nodes x and y also form a DAG that we denote by $SP(x, y)$. Given any edge set $f \subseteq E(G)$, we define $SP(x, y, f)$ as the subgraph formed by the shortest paths from x to y on $G \setminus f$.

3.1 Modelling segment routing

We formalize the behaviour of SR by defining a segment routing path, or **sr-path**, of length k from s to t as a sequence $\vec{p} = \langle x_1 \to x_2 \to \ldots \to x_{k-1} \to x_k \rangle$ where each $x_i \in V(G)$, $x_1 = s$ and $x_k = t$. To avoid confusion, we use the vector notation $\vec{p}, \vec{q}, \ldots$ for sr-paths and $p, q, \ldots$ for paths on G.

Segment routing paths determine the paths taken by packets. We call the latter forwarding paths. To introduce the definition of forwarding paths, we use the example in Fig. 2 where the sr-path is $\langle \mathrm{a} \to \mathrm{c} \to \mathrm{f} \rangle$. In this network, packets sent by node a on the sr-path $\langle \mathrm{a} \to \mathrm{c} \to \mathrm{f} \rangle$ can either follow path (a, b, c) or path (a, d, c) to reach node c as there is ECMP between nodes a and c. Then, node c will forward the packets along the path (c, e, f) to reach node f.

We note that when packets are forwarded along a sr-path from node x to y they can use any path in $SP(x, y)$. Routers use hash functions [22] to select a specific path among several equal cost ones. Since network operators cannot configure routers' hash functions, we assume that any given packet can be forwarded on any of the equal cost paths. For a sr-path $\vec{p} = \langle x_1 \to \ldots \to x_k \rangle$ we therefore define the **forwarding graph** of $\vec{p}$ to be the subgraph formed by the union of the shortest paths from x_1 to x_k on G that

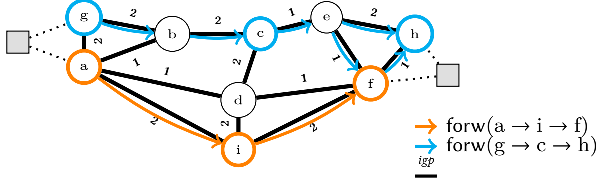


Figure 3: Disjoint sr-paths from a to f and from g to h.

visit nodes $x_1, \ldots, x_k$ in that order. We denote it by forw($\vec{p}$). It is easy to see that forw($x \rightarrow y$) = $SP(x, y)$ and that, more generally, forw($x_1 \rightarrow \ldots \rightarrow x_k$) = $\bigcup_{i=1}^{k-1}$ forw($x_i \rightarrow x_{i+1}$). For simplicity, we omit the brackets ⟨⟩.

We now define **disjoint sr-paths**. Recall from the above discussion on ECMP that packets using a sr-path $\vec{p}_1$ can be forwarded on any edge belonging to the forwarding graph forw($\vec{p}_1$). Hence, we say that two sr-paths $\vec{p}_1$ and $\vec{p}_2$ are *disjoint* if and only if the resulting forwarding graphs do not have edges in common, i.e. forw($\vec{p}_1$) ∩ forw($\vec{p}_2$) = ∅.

For example, suppose that a customer asks for two disjoint sr-paths to be installed in the network in Fig. 2, from a source site directly connected to the nodes a and g to a destination site attached to f and h. Fig. 3 shows two disjoint sr-paths ⟨g, c, h⟩ and ⟨a, i, f⟩ from the customer's source to the customer's destination sites. They are disjoint sr-paths because the corresponding forwarding graphs, respectively identified by blue and orange arrows, do not share any link.

3.2 Robustly disjoint sr-paths

Our final goal is to support paths that remain disjoint for input sets of link failures. In the following, we denote the input **failure set** as $\mathcal{F}$. Each element f of $\mathcal{F}$ is a subset of the set of edges that can fail at the same time. We assume that no failure in f disconnects the graph or otherwise it would even not be possible to provide connectivity at all, let alone disjoint paths.

Robustly disjoint paths for $\mathcal{F}$ are sr-paths that are disjoint in G and in any $G \setminus f$, for any element $f \in \mathcal{F}$. For example, we can set $\mathcal{F} = \{\{e\} \mid e \in E(G)\} = \mathcal{E}$ for paths that are robustly disjoint for any single-link failure. In practice, more or less failure cases can be included in $\mathcal{F}$, depending on the operators' needs. If SRLGs are known, then they can be added to the set. For instance, if three edges e_1, e_2, e_3 share the same physical resources and are likely to fail together, operators can add $\{e_1, e_2, e_3\}$ to $\mathcal{F}$. In the following, our examples focus on single-link failure scenarios, i.e. $\mathcal{F} = \mathcal{E}$; however, our *theory is general* with respect to the elements in $\mathcal{F}$.

To formalize robustly disjoint paths, we need to account for the effect of failures in $\mathcal{F}$. Failures generally force forwarding graphs associated to sr-paths to change. Figure 4 illustrates this point on our running example. If link (c, e) fails then, once the packets reach node c, in order to reach f, they follow the new shortest path (c, d, f). Similarly, if link (e, f) fails, the new shortest path to f becomes path (c, e, h, f). We denote the forwarding graph of a sr-path p after the removal of a given $f \in \mathcal{F}$ by forw($\vec{p}, f$). Formally,

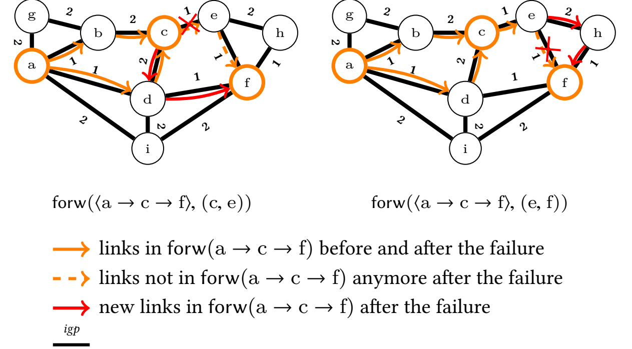


Figure 4: Effects of failures on a forwarding graph.

if $\vec{p} = \langle x_1 \rightarrow \ldots \rightarrow x_k \rangle$ then

$$\text{forw}(\vec{p}, f) = \bigcup_{i=1}^{k-1} \text{forw}(x_i \rightarrow x_{i+1}, f) = \bigcup_{i=1}^{k-1} SP(x_i, x_{i+1}, f).$$

We are now ready to give a formal definition of robustly disjoint sr-paths. Given two sr-paths $\vec{p}_1 = \langle x_1 \rightarrow \ldots \rightarrow x_k \rangle$ and $\vec{p}_2 = \langle y_1 \rightarrow \ldots \rightarrow y_r \rangle$ we say that they are **robustly disjoint** if and only if all the following conditions are met:

(1) $\vec{p}_1$ and $\vec{p}_2$ are disjoint sr-paths:

$$\text{forw}(\vec{p}_1) \cap \text{forw}(\vec{p}_2) = \emptyset$$

(2) $\vec{p}_1$ and $\vec{p}_2$ are disjoint sr-paths in $G \setminus f$ for all $f \in \mathcal{F}$:

$$\forall f \in \mathcal{F}\ \text{forw}(\vec{p}_1, f) \cap \text{forw}(\vec{p}_2, f) = \emptyset$$

We stress that these paths are always well defined as we assume that $\mathcal{F}$ does not contain failures disconnecting the graph.

3.3 The robustly disjoint sr-path problem

One of the operational requirements we consider is to compute paths with good data-plane performance – i.e. low delay. To formalize the actual problem that our algorithms solve, we have to define the **latency of a sr-path**. In general, a sr-path corresponds to a forwarding graph, which can contain many paths. We define the latency of a sr-path $\vec{p}$ as the *worst-case* latency over all possible paths:

$$lat(\vec{p}) = \max\{lat(q) \mid q \text{ is a } s\text{-}t \text{ path on forw}(\vec{p})\}.$$

We can now formalize our problem and prove its computational complexity.

Min-max robustly disjoint sr-path problem (RDP): Given a doubly weighted graph $G = (V, E, igp, lat)$, two sources $s_1, s_2 \in V(G)$, two destinations $t_1, t_2 \in V(G)$, a failure set $\mathcal{F}$ and an integer K, find two robustly disjoint sr-paths $\vec{p}_1 = \langle s_1 = x_1 \rightarrow \ldots \rightarrow x_k = t_1 \rangle$ and $\vec{p}_2 = \langle s_2 = y_1 \rightarrow \ldots \rightarrow y_r = t_2 \rangle$, with $k, r \leq K$, that minimize $\max(lat(\vec{p}_1), lat(\vec{p}_2))$.

Proposition 3.1. *RDP is NP-hard.*

Proof. Given a directed graph G and 4 nodes s_1, s_2, t_1, t_2, deciding whether there exist two edge disjoint paths, one from s_1 to s_2 and the other from t_1 to t_2, is NP-complete [44].

To prove that RDP is NP-hard, we reduce this problem, which we call disjoint path problem, to deciding whether the RDP has one feasible solution. Let $G = (V, E)$ and s_1, s_2, t_1, t_2 be disjoint path instance. We build an RDP instance as follows. We define a graph $H = (V, E', igp, lat)$ where $E' = \{(x, y, 1), (x, y, 2) \mid (x, y) \in E\}$, $igp \equiv 1$ and $lat \equiv 0$. We also set $K = n$ and $\mathcal{F} = E'$, and keep the same nodes s_1, s_2, t_1, t_2 as source-destination tuple. In other words, we duplicate each edge of the original graph. Thus, for each $x, y \in V$ $\mathsf{forw}(x \to y) = \mathsf{fail}(x \to y)$. This means that if we can find two disjoint sr-paths in H, they will also be robustly disjoint. Each solution of the built RDP instance can be efficiently mapped to a solution of the original problem by replacing each segment with any of the corresponding shortest path.

Also, checking that two sr-paths are robustly disjoint can be done in polynomial time, yielding the statement. □

There is one good news from a practical viewpoint, though. RDP is **fixed-parameter tractable** with respect to K, meaning that it is efficiently solvable for any given fixed value of K (the maximum length of an sr-path). This is a consequence of the fact that the number of sr-path pairs of length K is $O(n^{K-2})$, which is polynomial if K is a constant. Note that the search space is still very large for high values of K. Fortunately, we are mostly interested in solutions with *low* values of K, that would limit the data-plane overhead while also being compatible with commercial routers' capabilities. In the evaluation, we also show that robustly disjoint paths tend to exist for many source-destination tuples and practical failure sets in real ISP networks.

4 COMPUTING ROBUSTLY DISJOINT PATHS

To common way of finding disjoint paths in a graph is to compute the minimum cost flow between the sources and destinations using unit edge capacities to ensure disjointness [1] or use Suurballe's [37] specialization of that algorithm for pairs of paths. Both algorithms are very efficient but fail to solve the RDP problem in several ways. First, they cannot match sources with destinations, they can only enforce that each source will be linked to some destination. Second, they do not take into account ECMP so they assume that the exact output path will be enforced on the network. One way to overcome this with SR would be to compute a minimal segmentation of those paths [3]. However, we have no control on the number of segments that will be required and our experiments show that this number can be up to 26 segments, way above most hardware limitations. Third, these algorithms do not minimize the maximum latency of the paths but their sum. Finally, they provide no guarantees on robustness.

As the RDP problem is fixed-parameter tractable, one could wonder whether it is not sufficient to rely on a brute force algorithm that enumerates all possible pairs of sr-paths. Such an algorithm would simply enumerate all pairs of sr-paths and then pick the one with lowest worst-path delay (if any). The number of possible sr-path pairs with K segments is $n^{2(K-2)}$ and the checks to perform (to assess if sr-path pairs are robustly disjoint) are proportional to the number of elements of the input failure set $\mathcal{F}$, each of which requires a shortest path computation. Hence, the algorithm would run in $O(n^{2(K-2)}K|\mathcal{F}|mlog(n))$, where we recall that n and m are the number of nodes and edges in G. This would not be very efficient nor scalable for large ISP networks and a high number of input failures to protect from.

One of the difficulties of designing an algorithm for computing robustly disjoint paths is checking condition (2) of the robustly disjoint path definition, that is, checking whether the forwarding graphs are disjoint for every failure. Checking it either means that it will have to perform $O(|\mathcal{F}|)$ shortest path computations, which is very time-consuming, or pre-compute $\mathsf{forw}(x, y, f)$ for all $x, y \in V$ and $f \in \mathcal{F}$ which consumes a lot of memory, even on medium-sized topologies. To overcome this problem, we define a weaker version of robustly disjoint paths that assumes that each failure affects at most one of the two paths. We then propose an efficient algorithm for finding the best pair of robustly disjoint paths amongst all pairs of sr-paths that satisfy this weaker condition.

4.1 Aggregated failure subgraphs

Let $\vec{p}_1, \vec{p}_2$, be two sr-paths. Saying that each failure does not affect both $\vec{p}_1$ and $\vec{p}_2$ means that for each element $f \in \mathcal{F}$ either $\mathsf{forw}(\vec{p}_1, f) = \mathsf{forw}(\vec{p}_1)$ of $\mathsf{forw}(\vec{p}_2, f) = \mathsf{forw}(\vec{p}_2)$. This means that, to check whether disjointness holds after a failure, we can simply compare $\mathsf{forw}(\vec{p}_1)$ with $\bigcup_{f \in \mathcal{F}} \mathsf{forw}(\vec{p}_2, f)$ and $\mathsf{forw}(\vec{p}_2)$ with $\bigcup_{f \in \mathcal{F}} \mathsf{forw}(\vec{p}_1, f)$. This leads to the following definition.

To shorten the notation, we write $\mathsf{fail}(\vec{p}) = \bigcup_{f \in \mathcal{F}} \mathsf{forw}(\vec{p})$. Two sr-paths $\vec{p}_1$ and $\vec{p}_2$ are said to be **weakly robustly disjoint** if and only if all the following conditions are met:

(1) $\vec{p}_1$ and $\vec{p}_2$ are disjoint sr-paths:

$$\mathsf{forw}(\vec{p}_1) \cap \mathsf{forw}(\vec{p}_2) = \emptyset$$

(2) If a failure affecting $\vec{p}_2$ occurs, then the new links used by $\vec{p}_2$ do not intersect $\vec{p}_1$: $\mathsf{forw}(\vec{p}_1) \cap \mathsf{fail}(\vec{p}_2) = \emptyset$.
(3) If a failure affecting $\vec{p}_1$ occurs, then the new links used by $\vec{p}_1$ do not intersect $\vec{p}_2$: $\mathsf{fail}(\vec{p}_1) \cap \mathsf{forw}(\vec{p}_2) = \emptyset$.

We will write w–robustly disjoint paths for short. We now give a piecewise **characterization of w-robustly disjoint paths** that we use in our algorithms.

PROPOSITION 4.1. *Two sr-paths $\vec{p}_1 = \langle x_1 \to \ldots \to x_k \rangle$ and $\vec{p}_2 = \langle y_1 \to \ldots \to y_r \rangle$ are w-robustly disjoint if and only if for all $i = 1, \ldots, k-1$ and $j = 1, \ldots, r-1$, $forw(x_i \to x_{i+1}) \cap forw(y_j \to y_{j+1}) = forw(x_i \to x_{i+1}) \cap fail(y_j \to y_{j+1}) = fail(x_i \to x_{i+1}) \cap forw(y_j \to y_{j+1}) = \emptyset$.*

PROOF. We first prove that $\mathsf{forw}(\vec{p}_1) \cap \mathsf{forw}(\vec{p}_2) \neq \emptyset$ if and only if $\exists i, j : \mathsf{forw}(x_i \to x_{i+1}) \cap \mathsf{forw}(y_j \to y_{j+1}) \neq \emptyset$.

$$\begin{aligned} \mathsf{forw}(\vec{p}_1) \cap \mathsf{forw}(\vec{p}_2) \neq \emptyset \Leftrightarrow \\ \exists e :\ e \in \mathsf{forw}(\vec{p}_1) \cap \mathsf{forw}(\vec{p}_2) \Leftrightarrow \\ \exists e :\ e \in \mathsf{forw}(\vec{p}_1) \land e \in \mathsf{forw}(\vec{p}_2) \Leftrightarrow \\ \exists e, i, j :\ e \in \mathsf{forw}(x_i \to x_{i+1}) \land e \in \mathsf{forw}(y_j \to y_{j+1}) \Leftrightarrow \\ \exists i, j :\ \mathsf{forw}(x_i \to x_{i+1}) \cap \mathsf{forw}(y_j \to y_{j+1}) \neq \emptyset \end{aligned}$$

It is easy to see that $\mathsf{fail}(\vec{p}_2) = \bigcup_{i=1}^{r-1} \mathsf{fail}(y_i \to y_{i+1})$, so the same reasoning applies if we replace $\mathsf{forw}(\vec{p}_1)$ with $\mathsf{fail}(\vec{p}_1)$, or $\mathsf{forw}(\vec{p}_2)$ with $\mathsf{fail}(\vec{p}_2)$. This yields the statement. □

The advantage of this definition is that pre-computing $\text{fail}(x, y)$ for all $x, y \in V$ does not consumme a lot of memory nor time as we show in Section 4.4. This means that computing w-robustly disjoint path of minimal latency is feasible in practice. The following theorem shows that under some conditions, w-robuslty disjoint path are actualy robuslty disjoint.

PROPOSITION 4.2. *If $\vec{p}_1, \vec{p}_2$ are w-robustly disjoint and for each $f \in \mathcal{F}$ either $f \cap \text{forw}(\vec{p}_1) = \emptyset$ or $f \cap \text{forw}(\vec{p}_2) = \emptyset$ then $\vec{p}_1, \vec{p}_2$ are robustly disjoint.*

PROOF. Suppose that $\vec{p}_1, \vec{p}_2$ are not robustly disjoint. Then there exists f such that $\text{forw}(\vec{p}_1, f) \cap \text{forw}(\vec{p}_2, f) \neq \emptyset$. By definition, either $f \cap \text{forw}(\vec{p}_1) = \emptyset$ or $f \cap \text{forw}(\vec{p}_2) = \emptyset$. Assume, wlog, that $f \cap \text{forw}(\vec{p}_1) = \emptyset$. Therefore, we have $\text{forw}(\vec{p}_1) \cap \text{forw}(\vec{p}_2, f) = \text{forw}(\vec{p}_1, f) \cap \text{forw}(\vec{p}_2, f) \neq \emptyset$. Since $\text{forw}(\vec{p}_2, f) \subseteq \text{fail}(\vec{p}_2)$ this means that $\text{forw}(\vec{p}_1) \cap \text{fail}(\vec{p}_2) \neq \emptyset$, contradicting the fact that $\vec{p}_1, \vec{p}_2$ are w-robustly disjoint. □

Another interesting corollary is that for single link failures, the concepts of w-robustly disjoint paths and robustly disjoint paths coincide.

COROLLARY 4.3. *If $\mathcal{F} = \mathcal{E}$, a pair of sr-paths $\vec{p}_1, \vec{p}_2$ is w-robustly disjoint if and only if it is robustly disjoint.*

PROOF. Let $\vec{p}_1, \vec{p}_2$ be w-robustly disjoint sr-paths. Then, in particular, they are disjoint sr-paths. Hence, no $f \in \mathcal{F} = \mathcal{E}$ can intersect both forwarding paths. Thus, by Proposition 4.2, $\vec{p}_1, \vec{p}_2$ are robustly disjoint. The fact that robustly disjoint paths are w-robustly disjoint is obvious. □

We use these results to design an algorithm for computing the pair of w-wearkly robuslty disjoint paths that is also robuslty disjoint. Of course, we do not assume that the failures only affect one of the paths as the failures are given by the network operator and the paths are computed afterwards. Instead, we will make sure that the algorithm only considers pairs of paths that do not both intersect any of the given failures. Proposition 4.2 will then ensure that these paths are indeed robustly disjoint.

Even though checking the condition for w-robuslty disjoint paths can be done efficiently, we still have a huge search space of size $O(n^{2(K-2)})$ to explore. Our algorithm reduces the search space to at least $O(n^{K-2})$ by using the fact that, given a sr-path $\vec{p}_1$, finding a sr-path $\vec{p}_2$ of minimum latency that is w-robustly disjoint from $\vec{p}_1$ can be done efficiently as we show in the next section. We use this insight to efficiently build $\vec{p}_1$ while keeping the best compatible path $\vec{p}_2$. In our experiments (see Sec. 5), our algorithm runs two orders of magnitude faster than the brute force one.

Our algorithm takes two inputs: the network topology, $G = (V, E, igp, lat)$ and the failure set, $\mathcal{F}$. In practice, the network topology can be easily extracted from the Link State Database of commonly used IGPs like OSPF or IS-IS. The failure set contains the sets of simultaneous edge failures (e.g. single-link failures, SRLGs, ...) for which the operators want to be sure that the computed paths remain disjoint.

4.2 Finding low latency robustly disjoint sr-paths

Given a sr-path $\vec{p}$ from s_1 to t_1, we propose a dynamic programming algorithm to find the minimum latency sr-path from source s_2 to destination t_2 using at most K segments amongst all the paths that are w-robustly disjoint from $\vec{p}$.

We define $minlat(i, x)$ as the minimum latency of a sr-path from s_2 to node x of with at most i segment that is robustly disjoint from $\vec{p}$. Our goal is to find $minlat(K, t_2)$ and the corresponding path. Clearly $minlat(1, s_2) = 0$ and $minlat(1, x) = \infty$ for $x \neq s_2$ as the only sr-path of length 1 from s_2 is $\langle s_2 \rangle$. Given a node x, we define $\mathcal{N}(x, \vec{p})$ to be the set of nodes y such that $\langle y \to x \rangle$ is w-robustly disjoint from $\vec{p}$ and does not share failures with $\vec{p}_1$. In other words, $y \in \mathcal{N}(x, \vec{p})$ if and only if

$$\begin{aligned}\text{forw}(y \to x) \cap \text{forw}(\vec{p}) &= \text{forw}(y \to x) \cap \text{fail}(\vec{p}) \\ &= \text{fail}(y \to x) \cap \text{forw}(\vec{p}) = \emptyset\end{aligned}$$

and

$$\mathcal{F}(\vec{p}_1) \cap \mathcal{F}(\vec{p}_2) = \emptyset.$$

where we use $\mathcal{F}(\vec{p})$ to denote the indexes of the failures $f_i \in \mathcal{F}$ that intersect $\text{forw}(\vec{p})$.

The best way to reach x with a sr-path of length at most i is either the same as with a sr-path of length at most $i-1$ or we reach a node $y \in \mathcal{N}(x, \vec{p})$ in an optimal way with a sr-path of length at most $i - 1$ and append x to it, i.e.:

$$minlat(i, x) = \min \begin{cases} minlat(i-1, x) \\ \min\limits_{y \in \mathcal{N}(x, \vec{p})} minlat(i-1, y) + lat(y, x) \end{cases}$$

This recurrence can easily be computed in $O(n^2 \cdot K + n^2 \cdot m))$ since it computes $O(n^2 \cdot K)$ $minlat$ values and computing $\mathcal{N}(x, \vec{p})$ for all x costs $O(n^2 \cdot m)$. Recovering the sr-path is straightforward by keeping the minimizers of each state. The robustly disjointness is ensured by Proposition 4.2 since, by construction of $\mathcal{N}$, no failure intersects both paths.

4.3 Finding robustly disjoint sr-path pairs

We now develop an algorithm that builds a pair of robustly disjoint sr-paths that minimize $\max(lat(\vec{p}_1), lat(\vec{p}_2))$ amongst all pairs of w-robustly disjoint paths. In order to prune the search, this algorithm requires information on the path latencies. We represent this latency information as $\mathcal{L}$: a $n \times n$ matrix such that $\mathcal{L}(x, y)$ denotes the minimum latency path from x to y in G (note that $\mathcal{L}$ refers to paths, not sr-paths). This matrix can easily be pre-computed using any all-pairs shortest path algorithm on G using lat as edge weights [10].

To find our pair of robustly disjoint sr-paths, we perform a depth-first search. This search incrementally builds the sr-path $\vec{p}_1$ while using Algorithm 1 to maintain a complete sr-path $\vec{p}_2$ that is robustly disjoint from $\vec{p}_1$. Every time we try to extend $\vec{p}_1$ with a new node x, we check whether $\vec{p}_2$ is still robustly disjoint from the concatenation of $\vec{p}_1$ and $\langle x \rangle$. If it is, we continue the search. Otherwise, we compute a new sr-path $\vec{p}_2$ of minimum latency that is robustly disjoint from $\vec{p}_1$. If no such sr-path exists then we know that x is not a valid extension of $\vec{p}_1$ and we stop the search.

Algorithm 1 shortest-rdp $(\vec{p}, s, t, K)$

Input:

- $\vec{p}$: the sr-path that needs to be robustly disjoint.
- s: the source node
- t: the destination node
- K: the maximum length of the returned sr-path

Output: A sr-path from s to t that is robustly disjoint from $\vec{p}$ and has a length of at most K and a minimum latency. If no such path exists it returns $\perp$.

1: $minlat[i][x] \leftarrow +\infty$ **for all** $i = 1, \ldots, K, x \in V(G)$
2: $parent[i][x] \leftarrow \perp$ **for all** $i = 1, \ldots, K, x \in V(G)$
3: $minlat[1][s] \leftarrow 0$
4: **for** $x \in V(G)$ **do**
5: $\quad \mathcal{N}^-(x, \vec{p}) \leftarrow \emptyset$
6: $\quad$ **for** $y \in V(G)$ **do**
7: $\quad\quad$ **if** $\text{forw}(y \to x) \cap \text{forw}(\vec{p}) = \text{forw}(y \to x) \cap \text{fail}(\vec{p}) = \text{fail}(y \to x) \cap \text{forw}(\vec{p}) = \mathcal{F}(\vec{p}) \cap \mathcal{F}(\langle x, y \rangle) = \emptyset$ **then**
8: $\quad\quad\quad \mathcal{N}^-(x, \vec{p}) \leftarrow \mathcal{N}^-(x, \vec{p}) \cup \{y\}$
9: **for** $i = 2$ **to** K **do**
10: $\quad$ **for** $x \in V(G)$ **do**
11: $\quad\quad minlat[i][x] = minlat[i-1][x]$
12: $\quad\quad$ **for** $y \in \mathcal{N}^-(x, \vec{p})$ **do**
13: $\quad\quad\quad$ **if** $minlat[i-1][y] + lat(y, x) < minlat[i][x]$ **then**
14: $\quad\quad\quad\quad minlat[i][x] = minlat[i-1][y] + lat(y, x)$
15: $\quad\quad\quad\quad parent[i][x] = y$
16: **if** $minlat[k][t] = \infty$ **then**
17: $\quad$ **return** $\perp$
18: $\vec{q} \leftarrow \langle\rangle, ci \leftarrow K, cx \leftarrow t$
19: **while** $cx \neq \perp$ **do**
20: $\quad \vec{q} \leftarrow cx + \vec{q}$
21: $\quad cx \leftarrow parent[ci][cx]$
22: $\quad ci \leftarrow ci - 1$
23: **return** $\vec{q}$

When we reach a node of the search tree with a partial sr-path $\vec{p}_1 = \langle x_1 \to \ldots \to x_i \rangle$, before trying to expand it, we check whether we can prune the search based on the latency of the best solution so far $(\vec{p}_1^*, \vec{p}_2^*)$. We know that any extension of $\vec{p}_1$ will have a latency of at least $lat(\vec{p}_1) + \mathcal{L}(x_i, t_1)$. This is so because any extension of the sr-path $\vec{p}_1$ to reach t_1 will have a latency of at least $\mathcal{L}(x_i, t_1)$ since its latency will correspond to the latency of a path in G. In other words, $\mathcal{L}(x, y)$ is always a lower bound for $lat(x \to y)$. Therefore, we can safely stop the search if

$$\max(lat(\vec{p}_1) + \mathcal{L}(x_i, t_1), lat(\vec{p}_2)) \geq \max(lat(\vec{p}_1^*), lat(\vec{p}_2^*))$$

Algorithm 2 details the search initialization. Algorithm 3 shows the actual search procedure. In line 1 we check if the search can be cut-off. If it cannot, the input pair of paths is better than $\vec{p}_1^*, \vec{p}_2^*$. Thus if $\vec{p}_1$ is complete, we update the current best solution. Otherwise, we try all possible extensions of $\vec{p}_1$. Line 8 ensures that: (i) we do not choose a node that has already been selected, and (ii) if we are doing the last extension, we only consider t_1. We then extend $\vec{p}_1$ and continue the search on line 11 if this $\vec{p}_2$ is compatible with this extension. Otherwise, we look for new path $\vec{p}_2$ on line 13 and if we find one, we continue the search (line 15).

We now prove that our depth-first search computes robustly disjoint sr-paths of minimal latency.

Algorithm 2 robustly-disjoint-srpaths (s_1, s_2, t_1, t_2)

Input: The sources and destinations of the two paths s_1, s_2, t_1, t_2.

Output: A pair of robustly disjoint sr-paths from s_1 to t_1 and from s_2 to t_2 minimizing $\max(cost(\vec{p}_1^*), cost(\vec{p}_2^*))$. If not such pair exists we will have $\vec{p}_1^* = \vec{p}_2^* = \perp$.

1: $\vec{p}_1 = \langle\rangle$
2: $\vec{p}_2 = \text{shortest-robustly-disjoit-srpath}\left(\vec{p}_1, s_2, t_2, K\right)$
3: $\vec{p}_1^* \leftarrow \perp$
4: $\vec{p}_2^* \leftarrow \perp$
5: $\text{dfs}(G, \vec{p}_1, \vec{p}_2, \text{forw}, \text{fail}, w, bc, s_1, s_2, t_1, t_2, k, \vec{p}_1^*, \vec{p}_2^*)$
6: **return** $\vec{p}_1^*, \vec{p}_2^*$

Algorithm 3 dfs $\left(\vec{p}_1, \vec{p}_2, \vec{p}_1^*, \vec{p}_2^*\right)$

Input:

- $\vec{p}_1$: the sr-path that we are currently building
- $\vec{p}_2$: the minimum latency sr-path that is robustly disjoint from $\vec{p}_1$
- $\vec{p}_1^*, \vec{p}_2^*$: the best pair of robustly disjoint sr-paths found so far

1: **if** $\vec{p}_1^* \neq \perp$ **and** $\max(lat(\vec{p}_1) + \mathcal{L}(x_i, t_1), lat(\vec{p}_2)) \geq \max(lat(\vec{p}_1^*), lat(\vec{p}_2^*))$ **then**
2: $\quad$ **return**
3: **if** $\vec{p}_1.last = t_1$ **then**
4: $\quad \vec{p}_1^* \leftarrow \vec{p}_1$
5: $\quad \vec{p}_2^* \leftarrow \vec{p}_2$
6: $\quad$ **return**
7: **for** $x \in V(G)$ **do**
8: $\quad$ **if** $x \notin \vec{p}_1$ **and** $(|\vec{p}_1| \neq k - 1$ **or** $x = t_1)$ **then**
9: $\quad\quad \vec{p}_1.add(x)$
10: $\quad\quad$ **if** $\text{forw}(\vec{p}_1) \cap w\vec{p}_2) = \text{fail}(\vec{p}_1) \cap \text{forw}(\vec{p}_2) = \text{forw}(\vec{p}_1) \cap \text{fail}(\vec{p}_2) = \mathcal{F}(\vec{p}_1) \cap \mathcal{F}(\vec{p}_2) = \emptyset$ **then**
11: $\quad\quad\quad \text{dfs}(\vec{p}_1, \vec{p}_2, \vec{p}_1^*, \vec{p}_2^*)$
12: $\quad\quad$ **else**
13: $\quad\quad\quad \vec{p}_2' \leftarrow \text{shortest-rdp}\left(\vec{p}_1, s_2, t_2, K\right)$
14: $\quad\quad\quad$ **if** $\vec{p}_2' \neq \perp$ **then**
15: $\quad\quad\quad\quad \text{dfs}(\vec{p}_1, \vec{p}_2', \vec{p}_1^*, \vec{p}_2^*)$
16: $\quad\quad \vec{p}_1.remove\text{-}last()$

Proposition 4.4. *Algorithm 2 finds the pair of robustly disjoint paths of minimum latency amongst all pairs of w-robustly disjoint paths.*

Proof. Let $(\vec{p}_1^{**}, \vec{p}_2^{**})$ be a pair of w-robustly disjoint paths of minimum latency $\ell^* = \max(lat(\vec{p}_1^{**}), lat(\vec{p}_2^{**}))$. Write $\vec{p}^{**} = (s_1 = x_1, \ldots, x_k = t_1)$. We prove that the depth-first search algorithm, Algorithm 3, either reaches a node where $\vec{p}_1 = \vec{p}_1^{**}$ or finds a solution with cost ℓ^*. Suppose that the search is cut at line 2 with $\vec{p}_1$ being some prefix $(s_1 = x_1, \ldots, x_i)$ of $\vec{p}_1^{**}$. By definition of our cutting rule, one line 1, we have that,

$$lat(\vec{p}_1) + \mathcal{L}(x_i, t_1) = \sum_{j=0}^{i-1} lat(x_j, x_{j+1}) + \mathcal{L}(x_i, t_1)$$

Thus, by definition of *lat* it must hold that

$$\mathcal{L}(x_i, t_1) \leq \sum_{j=i}^{n-1} lat(x_j, x_{j+1})$$

so that

$$lat(\vec{p}_1) + \mathcal{L}(x_i, t_1) \leq \sum_{j=0}^{i-1} lat(x_j, x_{j+1}) + \sum_{j=i}^{n-1} lat(x_j, x_{j+1})$$
$$= \sum_{j=0}^{n-1} lat(x_j, x_{j+1}) = lat(\vec{p}_1^{**})$$

By Proposition 4.1, since p_2^{**} is w-robustly disjoint with $\vec{p}_1^{**}$ it must also be w-robustly disjoint with $\vec{p}_1$ as $\vec{p}_1$ is a prefix of $\vec{p}_1^{**}$. Algorithm 1 outputs the minimum cost path that is w-robustly disjoint with $\vec{p}_1$. Hence $lat(\vec{p}_2) \leq lat(\vec{p}_2^{**})$. Since the search is cut, it holds that $\max(lat(\vec{p}_1)+\mathcal{L}(x_i, t_1), lat(\vec{p}_2)) \geq \max(lat(\vec{p}_1^{*}), lat(\vec{p}_2^{*}))$. Therefore

$$\max(lat(\vec{p}_1^{**}), lat(\vec{p}_2^{**})) \geq \max(lat(\vec{p}_1) + \mathcal{L}(x_i, t_1), lat(\vec{p}_2))$$
$$\geq \max(lat(\vec{p}_1^{*}), lat(\vec{p}_2^{*}))$$

showing that the algorithm found a w-robustly disjoint pair of paths. Since in line 10 we ensure that they do not share failures, Proposition 4.2 shows that they are robustly disjoint.

On the other hand, if the search is never cut, it will reach a node with $\vec{p}_1 = \vec{p}_1^{**}$ and $\vec{p}_2$ equal to the minimum-cost robustly disjoint path from $\vec{p}_1$, which is also an optimal solution. □

4.4 Precomputing *forw* and *fail*

Since the subgraphs *forw* and *fail* are key in the above procedures, we also develop efficient algorithms to pre-compute these subgraphs for all pairs of nodes in a given network G and for an input failure set $\mathcal{F}$.

Computing any subgraph $forw(x \to y)$ maps to computing the shortest path directed acyclic graph (DAG) from x to y. We do this by first computing the shortest path DAG rooted at each node x, $SP(x)$. This is achieved by running the Dijkstra algorithm [10] at each node $x \in G$. Then, for a fixed x we need to extract for each y the subgraph of $SP(x)$ that contains all the shortest paths ending in y. To do this efficiently, we observe that $SP(x)$ is a DAG, and thus it admits a topological order. If we compute such an order $x_1, \ldots, x_n$, the shortest paths to a node x_i depend only on the shortest paths to nodes x_j for $j < i$. More concretely, $SP(x, x_1) = \emptyset$ for $x_1 = x$, and for any other node x_i, one shortest path from x to x_i must also be a shortest path from x to an in-neighbor y of x_i to which we add edge (y, x_i). Formally, for any $i > 1$ we have that

$$SP(x, x_i) = \bigcup_{y \in N^-(SP(x), x_i)} (SP(x, y) \cup \{(y, x_i)\}).$$

This formulation has the advantage of using set operations which can be performed quite efficiently by using a bitset representation of subgraphs. In our implementation, we index every edge of the input graph from 0 to $m - 1$. Whenever a subgraph is computed we preserve the indexes on the edges. Therefore, a subgraph can be represented by a simple bitset of size m where bits are set to 1 only on the edges that belong to the subgraph. This implementation makes all the set operations roughly 64 times faster.

To compute the failure subgraphs we use the same ideas. For each node x and set $f \in \mathcal{F}$ we compute $SP(x, f)$, which is also a DAG, and use the same process as above to compute $SP(x, y, f)$ for all y. We add each of those subgraphs to $fail(G, x, y)$. This is also where we check if fail is well-defined by verifying that each $SP(x, y, f)$ is not empty.

Real	Nodes	Edges
ISP 1	≈ 150	≈ 700
ISP 2	≈ 220	≈ 800
ISP 3	≈ 170	≈ 440

Rocketfuel	Nodes	Edges
RF 1221	108	306
RF 1239	315	1944
RF 1755	87	322
RF 3257	161	656
RF 3967	79	294
RF 6461	141	748

Largest in Topology Zoo	Nodes	Edges
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

Table 1: Topologies used in our experiments.

5 EVALUATION

We implement our algorithms in ≈ 6,000 lines of Java code. We use our implementation to experiment with three private ISP topologies, all Rocketfuel ones [35] and the largest networks in the Internet Topology Zoo [24]. Table 1 summarizes our dataset.

Our experiments span three types of failure sets:

- All single-link failures: $\mathcal{F} = \mathcal{E} = \{\{e\} \mid e \in E(G)\}$;
- Random 3-link failures: $\mathcal{F}$ contains m sets 3 element sets, $\{e_1, e_2, e_3\}$, where e_1, e_2, e_3 are randomly selected edges and m is the number of links in the topology;
- All 2-link failures: $\mathcal{F} = \{\{e_1, e_2\} \mid e_1, e_2 \in E(G)\}$.

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 the provider cannot offer a robust connectivity service between routers that are poorly connected. We repeat each experiment allowing between 1 and 3 detours. We stop at 3, because trends are already evident and real routers support a limited stack of SR labels [38].

We run all these experiments on a Dell Latitude E5450 with 8GB of Ram.

5.1 All single-link failures

We first compute paths that are robustly disjoint for all single-link failures. We envision that this may be a common use case for our approach, as several studies [28, 45] report that single-link failures are the most common failure cases in real networks.

Table 2 summarizes our experimental results. It also compares the performance of our algorithms against a baseline: the Suurballe's algorithm [37], designed to quickly compute disjoint paths that are *not* robustly disjoint. We remind that algorithms like Suurballe's one can only be used within a reactive approach. In contrast, our solution is proactive, and upon failures (at least, those specified

LEGEND: RDPs stands for Robustly Disjoint Paths, and det for detour(s). Src-dst tuples are two source-destination pairs, selected as described at the beginning of Sec. 5. Latency ratio is the average ratio between the max latency of the robustly disjoint paths computed by our algorithm and the max latency of the IGP shortest paths between the same source-destination tuples. The reported times are averages across all the src-dst tuples. Suurballe's algorithm refers to the algorithm in [37], that computes *simply* disjoint paths.

topology	Our algorithms											Suurballe's algorithm		
	src-dst tuples with RDPs				latency ratio wrt IGP			RDPs computation time			pre-	avg	max	latency
	0 det	1 det	2 det	3 det	1 det	2 det	3 det	1 det	2 det	3 det	computation	# det	# det	ratio
Real ISP 1	83%	100%	100%	100%	0.97	0.97	0.97	0.8 s	11.3 s	6.98 m	4 s	2	5	0.90
Real ISP 2	89%	100%	100%	100%	0.98	0.98	0.98	2.5 s	1.6 m	70.5 m	9 s	1.58	4	0.88
Real ISP 3	73%	100%	100%	100%	0.97	0.96	0.96	0.2 s	3.5 s	2.9 m	4 s	4.9	10	0.84
RF 1221	82%	98%	100%	100%	0.99	0.99	0.99	29 ms	0.4 s	9.2 s	1 s	1.04	4	0.89
RF 1239	90%	100%	100%	100%	0.97	0.97	0.97	5.8 s	43.9 s	4.2 m	44 s	2.12	4	0.87
RF 1755	52%	98%	100%	100%	0.90	0.89	0.88	40 ms	0.36 s	1.8 s	1 s	2.34	5	0.80
RF 3257	76%	100%	100%	100%	0.91	0.89	0.88	0.3 s	3.4 s	37 s	4 s	3.06	8	0.72
RF 3967	71%	99%	100%	100%	0.97	0.97	0.97	36 ms	0.5 s	14.8 s	1 s	1.91	5	0.79
RF 6461	75%	100%	100%	100%	0.97	0.97	0.97	0.3 s	3.7 s	3.59 m	3 s	2.65	6	0.82
ITZ Cogentco	78%	97%	100%	100%	0.85	0.84	0.84	4.5 s	5.7 s	18 s	4 s	2.01	6	0.70
ITZ Colt	58%	71%	73%	73%	0.88	0.87	0.86	3.7 s	4.9 s	20 s	2 s	2.31	6	0.73
ITZ Deltacom	74%	99%	99%	100%	0.91	0.90	0.90	4 s	4.7 s	4.58 s	1 s	4.08	9	0.81
ITZ Dia	54%	77%	79%	79%	0.96	0.98	0.98	4.59 s	4.4 s	10 s	2 s	0.85	3	0.85
ITZ GtsCe	78%	98%	100%	100%	0.78	0.77	0.75	7.24 s	5 s	4.54 s	1 s	2.8	8	0.60
ITZ Interoute	81%	99%	100%	100%	0.93	0.91	0.90	5.4 s	5.79 s	3.46 s	1 s	4.01	10	0.81
ITZ Ion	64%	100%	100%	100%	0.95	0.94	0.94	3.29 s	3.68 s	2.56 s	1 s	1.43	5	0.86
ITZ Tata	86%	100%	100%	100%	0.90	0.89	0.89	4 s	6 s	4.32 s	2 s	3.19	11	0.75
ITZ UsCarrier	72%	83%	85%	85%	0.92	0.92	0.92	4.6 s	4.6 s	11 s	2 s	1.11	4	0.86

Table 2: Summary of the results for paths that are robustly disjoint for any single-link failure. Our algorithms are very effective to find robustly disjoint paths using few SR segments, for all source-destination tuples that theoretically admit such paths.

in input), it relies on the automated IGP convergence only, without any reaction to failures from operators or centralized controller.

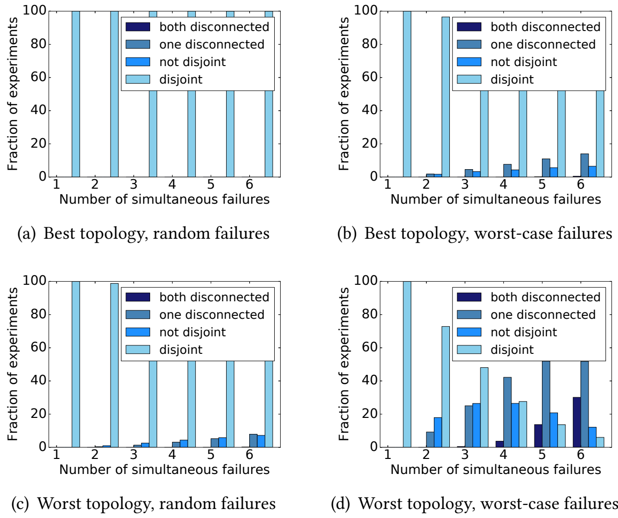
Real networks admit robustly disjoint paths. The leftmost part of Table 2 reports the percentage of source-destination tuples (among those we selected as described before) for which our algorithms can compute robustly disjoint paths for any single-link failure. Sometimes, only selecting the right IGP 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 71%-100% across all the topologies, and to 97% or more for all topologies but two. Allowing more detours provides only slightly more flexibility in our experiments.

As a comparison, Suurballe's algorithm can compute paths for all source-destination tuples we experiment with – which is expected as disjoint paths must exist for those tuples, by their definition. The algorithm runs in few milliseconds. These paths also provide better average latency ratio. However, implementing Suurballe's paths would require an average number of segments comparable and often higher than the one needed for robustly disjoint paths. Also, in the worst case, Suurballe's paths require up to 9-11 segments for several topologies, which would create a large per-packet overhead and goes beyond the capabilities of existing routers [38].

The computed robustly disjoint paths decrease worst-path latency. Our algorithms are designed to find sr-paths that are both robustly disjoint and have minimal worst-path delay. As the central part of Table 2 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.

Robustly disjoint paths can sustain more failures than the input ones. Robustly disjoint paths are computed with respect to the failures in $\mathcal{F}$ specified as input by the operator. However, unexpected failures can and will eventually occur.

We check how paths robustly disjoint for any single-link failure react to multiple-link failures. For $s = 1, \ldots, 6$, we generate 100 sets of s simultaneous link failures. We simulate the effect of each failure set on the two paths computed by our algorithm. After each failure, we record if the paths remain disjoint, if they are both working but not disjoint anymore, if one source-destination pair is disconnected, or if both source-destination pairs have no path anymore.



(a) Best topology, random failures

(b) Best topology, worst-case failures

(c) Worst topology, random failures

(d) Worst topology, worst-case failures

Figure 5: Paths robustly disjoint for *single* link failures likely sustain *multiple* link failures. Worst-case failures refer to our experiments where all the simulated failures affected links used in the computed robustly disjoint paths.

We perform two types of experiments using the above methodology. In the *random failures* experiment, we generate the sets of failures by successively extracting edges randomly, with uniform probability, among all network edges. In the *worst-case failures* experiment, all the failed edges are extracted randomly from the used paths (i.e., the N-th failed edge is extracted from the paths resulting from the previous N-1 failures).

Fig. 5 shows the results of our experiments on ISP 1 and ITZ Dia: they are 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 ISP 1, 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 ITZ Dia, 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.

Our algorithms are fast with respect to the envisioned use cases. We now evaluate how fast our algorithms are, and if they can enable ISPs to quickly setup disjoint path services for new customers, and to re-compute paths for expected (e.g., maintenance operations) and possibly unexpected failures.

Table 2 reports the average computation times taken by our algorithms, when run on the topologies in our dataset. We break down the computation time in two components.

The first component is the time to output robustly disjoint paths with pre-computed fail and forw subgraphs. This is, for example, the time that would be needed for an ISP to set up the disjoint-path service for a new customer. Our algorithms compute robustly disjoint paths in less than 10 seconds on average in most of our experiments. Only for Real ISP 2 with 3 detours the average time is considerably larger. Overall, we consider these times reasonable for realistic use cases. As comparison, setting up a new connectivity service typically takes days for current ISPs.

The second component is the time needed to compute the forw and fail subgraphs. This computation only depends on the topology and the failure set; it takes only a few seconds on all our topologies for the all single-link failure scenario. This time has to be added to the first time component discussed above whenever robustly disjoint paths have to be recomputed after a topology change. We argue that the sum of the two time components shown in Table 2 is still reasonably low for planned addition or removal of links, e.g., during a maintenance operations. With respect to unexpected failures, times to compute robustly disjoint paths with 1 detour seem still reasonable, especially considering that many robustly disjoint paths tend to remain disjoint or at least preserve connectivity after more than the expected link failures provided in input (see Fig. 5).

We note that time efficiency is a property of our algorithms. As discussed in Sec. 3, the RDP problem is can be solved in polynomial time for any fixed number of detours, but the space of its solutions is very large. In fact, a brute force algorithm scanning all the sr-path pairs without pruning the search space takes two orders of magnitude more time than our algorithms, across all topologies.

We also remark that the computation time does not necessarily increase with the number of segments. This is because allowing more segments increases the number of robustly disjoint paths, which is a favorable condition for our approach. Indeed, our algorithms are fast to find robustly disjoint paths when they exist, and effective in using the computed paths to prune the search space.

5.2 Multiple link failures

We repeat our experiments on multiple-link failure sets.

Our algorithms are still efficient to compute the (more rare) paths which are robustly disjoint for random 3-link failures. As already discussed, multiple links may fail at the same time in practice (e.g. SRLGs). We have no information of possible SRLGs in our topologies. As a rough approximation, we experiment with failure sets of 3 random links.

Our algorithms are still very efficient to compute robustly disjoint paths on the random 3-link failure cases: their average computation time is always below 20 seconds across all our topologies.

Despite being efficient, they could not compute robustly disjoint paths for as many source-destination tuples as for the single-failure experiments. This is mainly because the paths to be computed must be robust to a higher number of failures. For example, the percentage of source-destination tuples with a robustly disjoint path becomes 41%-43% (depending on the number of detours) for ISP 1, 36%-45% for ISP 2, and only 9% for ISP 3. Overall, the best topologies are RF 1239 (90% of source-destination tuples), RF 6461 (89%-100%) and RF 3257 (66%-80%), while it is very rare to find any source-destination tuple supporting robustly disjoint paths in Internet Topology Zoo's networks.

These results mainly depend on the density of the individual topologies. In fact, it is well known that Internet Topology Zoo's topologies are generally not extremely well connected (see, e.g., [19]).

Further analyzing our topologies, we find a direct correlation between the presence of robustly disjoint paths for 3-link failures and the minimum cut for pairs of nodes in the corresponding network. The *minimum cut for two nodes* is defined as the minimum number of edges that we would need to remove for disconnecting the two nodes; its size is therefore equal to the maximum number of disjoint paths between the two nodes. Node pairs have a minimum cut higher than 2 for a very low number of Internet Topology Zoo's networks. In contrast, in the topologies, the minimum cut for more node pairs is higher than 2; and the ones with many node pairs with higher minimum cut values are also the ones on which robustly disjoint paths for random 3-link failures can be computed for most tuples. For example, the minimum cut is higher than 2 for about 50% of node pairs, and even higher than 30 for some pairs, in RF 1239. The discrepancy between the Internet Topology Zoo and the other (router-level) topologies in our dataset is perhaps due to the fact that many networks in [24] are PoP-level (i.e. each node represents a Point-of-Presence) and inferred from public Web sites.

Our algorithms show limitations for *all possible* 2-link failures. Obviously, our solution do not scale indefinitely. We start to observe limitations when trying to compute paths robustly disjoint for all 2-link failures, a requirement that is ambitious even for simpler failure-tolerant approaches (e.g. only preserving connectivity).

In addition to exacerbate the reduction of source-destination tuples admitting robustly disjoint paths, this failure set also significantly slows down our algorithms. The biggest challenge for our algorithm in this case is to pre-compute forw and fail since the failure set $\mathcal{F}$ is huge in this case. For instance, on RF 1239 $\mathcal{F}$ has almost four million elements and it takes about 15 hours just to pre-compute these graphs. Intuitively, the efficiency reduction of our algorithms for finding robustly disjoint path is likely due to the presence of less robustly disjoint paths. In fact, our algorithms tend to quickly find solutions when they exist, but they take much more time to prove that no solution exists. This is expected since our algorithms are not able to prune the search space when there is no solution.

5.3 Simulations on a real failure trace

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 $\mathcal{F} = \mathcal{E}$ (all single-link failures). We then replay 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.

6 RELATED WORK

We consider tighter requirements than connectivity. Prior contributions have focused on how to quickly restore connectivity upon failures. Basic (deployed) techniques [2] quickly restore connectivity after single link failures. More advanced ones [9, 12, 27, 39] can address multiple link failures, even providing strong robustness guarantees like the ability to deliver packets after failure of any $k-1$ links, with $k \leq 5$, in k connected networks – which is more than what our solution can ensure. Some of those techniques can also be implemented with SR [15]. However, *none of the above contributions* maintains paths disjoint after failures (in a set specified by operators), which is the goal of our work.

We explore a new class of disjoint paths. Many research efforts considered problems related to path disjointness in the past [8, 18, 23, 25]. They study the computational complexity and propose efficient algorithms to calculate specific disjoint paths (e.g. minimizing the maximum delay or including one shortest path). Other works focus on how to install disjoint paths in MPLS networks [34, 42] or with Segment Routing [3]. However, *none of them* provides guarantees about disjointness after failures.

We provided the background to reason about paths that remain disjoint for any failure in an input set (§3), and use it to design efficient algorithms to compute such paths (§4).

We complement recent work on Segment Routing and configuration synthesis. Efficiently encoding robustly disjoint paths represents a new application scenario for a protocol that is classically associated to traffic engineering [7, 21, 33] or monitoring [4]. Also, our work confirms the effectiveness of configuration synthesis to (pro-actively) deal with failures, i.e., to keep paths disjoint in addition to maintaining compliance with routing policies [6] and achieving congestion-free failure recovery [20].

7 CONCLUSIONS

We started this paper from a conversation with a specific ISP, willing to configure disjoint paths with Segment Routing in order to offer a robust connectivity service to its customers. We investigated theory and algorithms to provide such a service in an automated and reliable way. We focused on SR by explicit request of our reference ISP, who wanted to avoid the high costs of physically replicating the network as well as the operational issues of MPLS.

The obvious follow-up question is whether SR is a good routing technology to implement the disjoint-path service, and what are the pros and cons of relying on it.

We analyzed the main alternative approaches, dividing them into classes that cover existing mechanisms, prior work, and non-SR implementations of robustly disjoint paths (e.g. through pre-provisioned MPLS tunnels selectively activated upon specific failures). We compared these classes with our approach over four dimensions: ability to guarantee disjoint paths, additional network

Approach	Path Disjointness	Network State	Expressiveness	Failure Reaction
shortest path routing (e.g., IGP)	*not always possible*	minimal	connectivity	IGP convergence
connectivity-targeted fast rerouting (e.g., LFA [2] or with OpenFlow [9])	*not considered*	additional state on routers	connectivity	data-plane update
failure-tolerant configurations (e.g., FFC [26] or synthesized [6, 20])	*not considered*	light routers' state, protocol-specific overhead	forwarding policies (e.g., no congestion)	data-plane update
centrally configured disjoint paths (e.g., with RSVP-TE [42] or SR [3])	if no failure, or via reconfiguration	*heavy routers' state* (stores backup paths)	all possible disjoint paths	configuration changes (possibly many)
robustly disjoint paths with RSVP-TE or OpenFlow	for any failure in an input set	*heavy routers' state* (stores backup paths)	all possible disjoint paths	data-plane update
robustly disjoint paths with SR (this paper)	***for any failure in an input set***	***minimal on routers, longer packet headers***	***SR-implementable disjoint paths***	***IGP convergence***

Table 3: Our approach trades some expressiveness (not all paths are implementable in SR) and data-plane overhead (longer packet headers) for the ability to keep paths disjoint for an input set of failures, without requiring state on the routers and any configuration change upon failures.

state required (on routers and packets), expressiveness, and failure-triggered reaction.

Table 3 summarizes the results of our analysis. It highlights two main observations. On one hand, SR enables to configure paths that remain disjoint after an input set of failure cases, without overloading routers with an exponential number of backup paths or requiring any configuration change upon failures – and ours is the first work studying and exploiting this opportunity, to the best of our knowledge. On the other hand, the price to pay for such an approach is to rely on IGP convergence for failure recovery, restrict to paths implementable with SR, and encode information in the packet header. These limitations do not seem to be very constraining in realistic deployments. The IGP convergence has been shown to be fast [16], even if slower than data-plane updates. Also, our evaluation suggests that adding 1 or 2 conveniently computed SR segments to packets enables to implement robustly disjoint paths for many reasonably well-connected source-destination pairs in ISP networks and frequent failures (e.g., all the single-link ones).

One additional point is worth noting. It is thanks to SR that we have been able to design an efficient algorithm for an otherwise computationally intractable problem. In fact, finding delay-minimizing robustly disjoint paths is fixed parameter tractable in SR (see Sec. 3), while computationally hard in general [44]. This is a core element for the practicality of the SR configuration synthesis we studied.

SOFTWARE ARTIFACTS

A Java implementation of our algorithms, the public topologies used for the experiments and the experimental results are publicly available on the following repository https://bitbucket.org/franaubry/robustlydisjointpathscode/.

ACKNOWLEDGMENTS

We are grateful to the CoNEXT anonymous reviewers and our shepherd, Desislava Dimitrova, for the insightful comments that helped improving this paper. This work is partially supported by the ARC grant 13/18-054 (ARC-SDN) from Communauté française de Belgique.

REFERENCES

[1] 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.
[2] A. Atlas and A. D. Zinin. Basic Specification for IP Fast Reroute: Loop-Free Alternates. RFC 5286, 2008.
[3] F. Aubry, D. Lebrun, Y. Deville, and O. Bonaventure. Traffic duplication through segmentable disjoint paths. In *IFIP Networking*, 2015.
[4] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure. SCMon: Leveraging Segment Routing to Improve Network Monitoring. In *INFOCOM*, 2016.
[5] 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.
[6] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *SIGCOMM*, 2016.
[7] R. Bhatia, F. Hao, M. Kodialam, and T. Lakshman. Optimized Network Traffic Engineering using Segment Routing. In *INFOCOM*, 2015.
[8] A. Björklund and T. Husfeldt. Shortest two disjoint paths in polynomial time. In *ICALP*, 2014.
[9] M. Chiesa, I. Nikolaevskiy, S. Mitrovic, A. Panda, A. Gurtov, A. Maidry, M. Schapira, and S. Shenker. The quest for resilient (static) forwarding tables. In *INFOCOM*, 2016.
[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.
[11] D. Dolev, C. Dwork, O. Waarts, and M. Yung. Perfectly Secure Message Transmission. *J. ACM*, 40(1):17–47, Jan. 1993.
[12] T. Elhourani, A. Gopalan, and S. Ramasubramanian. Ip fast rerouting for multi-link failures. In *INFOCOM*, 2014.
[13] A. Farrel, O. Komolafe, and S. Yasukawa. An analysis of scaling issues in mpls-te core networks. IETF RFC5439, 2009.
[14] 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.
[15] K.-T. Foerster, M. Parham, M. Chiesa, and S. Schmid. Ti-mfa: Keep calm and reroute segments fast. In *Proc. IEEE Global Internet Symposium (GI)*, 2018.

[16] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving sub-second igp convergence in large ip networks. *ACM SIGCOMM Computer Communication Review*, 35(3):35–44, 2005.

[17] M. Ghobadi and R. Mahajan. Optical Layer Failures in a Large Backbone. In *IMC*, 2016.

[18] Y. Guo, F. Kuipers, and P. Van Mieghem. Link-disjoint paths for reliable qos routing. *Int. Jour. of Comm. Sys.*, 16(9):779–798, 2003.

[19] N. Gvozdiev, S. Vissicchio, B. Karp, and M. Handley. Low-Latency Routing on Mesh-Like Backbones. In *HotNets*, 2017.

[20] F. Hao, M. Kodialam, and T. Lakshman. Optimizing Restoration with Segment Routing. In *INFOCOM*, 2016.

[21] 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. In *SIGCOMM*, 2015.

[22] C. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, 2000.

[23] R. M. Karp. Reducibility among combinatorial problems. In *Symposium on the Complexity of Computer Computations*, 1972.

[24] 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.

[25] C. Li et al. The complexity of finding two disjoint paths with min-max objective function. *Discrete Appl. Math.*, 26(1):105 – 115, 1990.

[26] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic Engineering with Forward Fault Correction. In *SIGCOMM*, 2014.

[27] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *NSDI*, 2013.

[28] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM transactions on networking*, 16(4):749–762, 2008.

[29] P. Mattes. Segment Routing for Datacenter Interconnect at Scale. MPLS World Congress, 2017.

[30] NANOG mailing list. Bell outage. https://mailman.nanog.org/pipermail/nanog/2017-August/091828.html, 2017.

[31] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz. Latency Inflation with MPLS-based Traffic Engineering. In *IMC*, 2011.

[32] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277. ACM, 2011.

[33] 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 *IFIP Networking*, 2017.

[34] V. Sharma and F. Hellstrand. Framework for Multi-Protocol Label Switching (MPLS)-based Recovery. RFC 3469, 2003.

[35] N. Spring et al. Measuring ISP Topologies with Rocketfuel. *IEEE/ACM ToN*, 12(1):2–16, 2004.

[36] R. Steenbergen. MPLS Autobandwidth. RIPE 64 presentation, 2012.

[37] J. W. Suurballe and R. E. Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984.

[38] 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.

[39] O. Tilmans and S. Vissicchio. IGP-as-a-Backup for Robust SDN Networks. In *CNSM*, 2014.

[40] F. Trate. Segment Routing is crossing the chasm with real live deployments. Cisco Blog, 2016.

[41] 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.

[42] J.-P. Vasseur, M. Pickavet, and P. Demeester. *Network recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS.* Elsevier, 2004.

[43] D. Voyer. CO/DC Network Transformation. MPLS World Congress, 2017.

[44] J. Vygen. Np-completeness of some edge-disjoint paths problems. *Discrete Applied Mathematics*, 61(1):83 – 90, 1995.

[45] D. Watson, F. Jahanian, and C. Labovitz. Experiences with monitoring ospf on a regional service provider network. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 204–213. IEEE, 2003.

[46] R. White and D. Donohue. *Art of Network Architecture, The: Business-Driven Design.* Cisco Press, 2014.