FAST AND SCALABLE OPTIMIZATION FOR SEGMENT ROUTING

RENAUD HARTERT

Thesis submitted in fulfillment of the requirements for the degree of Doctor of Applied Science in Engineering

September 2018

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Louvain School of Engineering
Louvain-la-Neuve
Belgium

"I can't, it's too big !"

— Luke Skywalker

ABSTRACT

Nowadays, the Internet delivers more than 1 zettabyte (or 1,000,000,000,000,000,000,000 bytes) of data to more than 3.2 billions of users every year. As a consequence, network operators face difficult challenges due to the impressive (and continuously growing) quantity of data to handle. Upgrading a network infrastructure with better hardware, to accommodate such a growth, has a significant cost (in terms of money and time) that network operators are trying to reduce by improving the efficiency (and thus the lifespan) of their network infrastructures.

Traffic Engineering is a domain of networking that specifically aims at providing network operators with tools to optimize their networks. This thesis focuses on the problem of increasing the efficiency of a network by controlling the paths on which the traffic is sent using a new network framework called Segment Routing. In particular, we present various algorithms based on Mixed Integer Programming, Local Search, and Constraint Programming techniques to solve this traffic engineering problem on large networks like the ones of major Internet service providers.

ACKNOWLEDGEMENT

Dressing the list of all the people who contributed to the achievement that this thesis represents is a challening task as it would undoubtedly require its own additional chapter. Though, I could not possibly present this document without mentioning a few persons who had a significant impact on this thesis and on myself.

First and foremost, I'd like to express my gratitude to my advisors Pierre Schaus and Olivier Bonaventure. The path that led to this thesis was disseminated with pitfalls and it is thanks to their guidance that I managed to avoid all of them — almost.

Many thanks go to my office mates: Cyrille Dejemeppe, Sascha Van Cauwelaert, and Steven Gay. Climbing the infamous "rue des wallons" every morning to reach the office was a real pleasure thanks to you. The work environment we managed to create is the best balance I ever experienced in terms of productivity, friendship, and fun. To that regard, I think that our white board deserves its own round of applause for all the discussions, debates, and *situations* — may it remain silent — it had to endure.

When I think about this thesis, two persons immediately come to my mind. The first is Steven Gay who has been my research mate for the past 4 years at UCL but also at Google. Steven is the person who ultimately made this thesis a reality as he kept pushing me to finish this document. Working on this thesis after long office hours was difficult and exhausting. I would never have completed this journey if it wasn't for Steven's encouragement. The second person is our department secretary, Vanessa Maon, who was basically the *god mother* of everyone in the office. I would never have managed to complete this document in time without her help filling the administrative paper work while I was working abroad.

Last but not least, I want to thank my parents who never stopped to support me despite the few *detours* I chose to take during my studies. This thesis is entirely dedicated to their unswerving trust.

CONTENTS

1 INTRODUCTION

In a few decades, the Internet has quickly evolved from ARPAnet, a research network sponsored by the DARPA in the late 1960s [20], to a world spanning infrastructure responsible for delivering a large variety of services (such as social networks, video streaming, and cloud computing) to more than 3.2 billions people [51]. As of today, the global traffic delivered on IP networks (read "the Internet") has exceeded the symbolic quantity of one zettabytes (i.e. 10^{21} bytes). Cisco Systems, one major vendor of network solutions, estimates that this global annual traffic will triple in 2021 reaching 3.34 zettabytes (see Figure 1.1) [104]. To put things in perspective, that corresponds to 22,829 DVDs[1] worth of data to be sent over the world *every second*.

If the Internet has considerably changed in terms of traffic, it has also significantly evolved in shape. Indeed, the Internet is not a single network anymore but a collection of interconnected smaller networks best known as *autonomous systems*. An Autonomous System (AS) is a complete network on its own which is managed by a single authority such as an Internet Service Provider (ISP), a country, or a University campus [58]. Each AS is responsible for defining its own intra-domain routing which can be seen as the process of transmitting Internet traffic through the AS's network. It is the job of the AS's network operators to configure intra domain routing in order to increase the operational efficiency of the network and make sure that it answers the needs of its customers.

1.1 TRAFFIC ENGINEERING

Traffic Engineering (TE) is a field of networking that aims at providing network operators with tools to configure their network. Network congestion, a phenomenon that occurs when network components have to carry more data than what they can handle, is one of the most important problems TE has

1 Assuming a capacity of 4.7 GB.

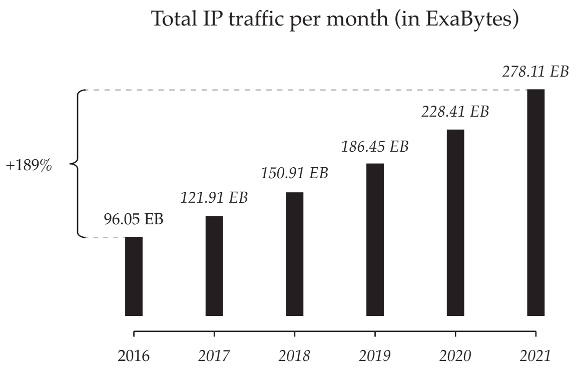


Figure 1.1: Cisco Systems Forecast (in italic) of global IP traffic [61].

to deal with because it almost always results in a degradation of the user experience [9].

In this thesis, we focus on a particular aspect of TE that consists in increasing the operational capacity of the network — and thus reduce its exposition to congestion — by controlling the paths on which packets are forwarded. The typical input of this problem, that we refer to as the *general traffic engineering problem*, are (i) the AS's network topology, (ii) a set of demands that represent the different flows of packets that are forwarded on the network, and (iii) a metric to minimize. We formalize them as follows.

Definition 1: Network. A network is a *strongly connected directed graph* $T(\mathbf{N}, \mathbf{E})$ made of a set of nodes $\mathbf{N}$ and of a set of directed edges $\mathbf{E}$. Each edge $(u, v) \in \mathbf{E}$ is associated with a capacity $c_{u,v} \in \mathbb{N}$. We assume that each edge $(u, v) \in \mathbf{E}$ has a sibling $(v, u) \in \mathbf{E}$ but both edges can be associated to different capacities.

Nodes and edges respectively represent the routers and the links of that network. The capacity of an edge is its bandwidth, i.e., the maximum quantity of bits the link can deliver per second.

Definition 2: Demand. A demand d is a triplet (s, t, k) that represents a continuous flow of k packets, called the bandwidth of the demand, that traverses a network $T(\mathbf{N}, \mathbf{E})$ from a *source* node $s \in \mathbf{N}$ to a *destination* node $t \in \mathbf{N}$.

A demand can spread itself over different paths in the network as long as they all connect its source to its destination. Demands to be forwarded on the network are contained in a set $\mathbf{D}$. Note that several demands having the same source and destination may exist thus allowing $\mathbf{D}$ to be much larger than $|\mathbf{N}|^2$.

Definition 3: Link Utilisation. Given a routing policy, we can associate each link (u, v) to a load $load_{u,v}$ that corresponds to the total amount of packets delivered through that link. The utilisation of (u, v) is $load_{u,v}/c_{u,v}$ and a link is said to be congested if its utilisation exceeds 100%.

Definition 4: Network Utilisation. The *network utilisation*, denoted ϕ, is the value of the maximum link utilisation $\phi = \max_{(u,v)\in\mathbf{E}} \frac{load_{u,v}}{c_{u,v}}$.

Configuring the forwarding of packets so that the network utilisation is minimized is a popular way to minimize the network exposition to congestion in both online and offline contexts. Indeed, it tends to minimize the consumption of network resources (offline) and thus to maximize the effective capacity of the networkwhich is more likely to accept new demands (online). Unfortunately, minimizing the network utilisation might result in a poor use of the network resources if the network has bottlenecks, i.e. links whose utilisation cannot be reduced no matter the routing policy (see Section A.2). The problem with bottlenecks is that they are likely to block the minimization of the remaining link utilisations. Researchers and network operators have thus proposed several alternatives that do not suffer from this weakness. In [10], Balon et al. compared how the most famous of those alternative objective functions perform in realistic contexts.

While Balon et al. analysis highlighted that the objective functions proposed by Degrande et al. [30], Elwalid et al. [36], and Fortz et al. [40] provide good results in many situations, it also showed that minimizing the network utilisation performs well in most cases. Therefore, we chose to focus on minimizing the network utilisation because it still remains a commonly used objective function used by default by most researchers in the traffic engineering community.

1.2 SOFTWARE DEFINED NETWORK

Aside from the continuous growth of Internet traffic [104], dealing with congestion has become a challenge of increasing difficulty for network operators. Indeed, recent years have seen an important rise of sudden traffic surges (due to newly popular content, social networks, or related flash crowds [112]) that tend to significantly stress the network devices. The most common — and arguably the most reliable — solution network operators have found to tackle those unexpected events is to provision their networks to operate at 30-40% of their capacity. This has the advantage of virtually masking all non-critical perturbation, but comes at the prohibitive cost of a two to three fold over-provisioned network infrastructure. For those reasons, dynamically controlling the paths followed by the traffic has become an increasingly critical challenge for network operators.

Software Defined Network (SDN) [69] is a new paradigm specifically designed to solve the problems encountered by modern network applications. In a nutshell, SDN exposes the configuration of the network's devices through high-level programmable interfaces (see Open-flow [74]) thus enabling network management by a centralized software entity called *SDN controller*. Using the SDN controller, network operators are now able to monitor and adapt the behavior and functionalities of their network in reliable and secure way, without having to manually configure the network devices anymore. It is therefore not surprising that researchers and network operators have started to rely on SDN controllers to automatically perform TE tasks by reconfiguring the network devices when needed.

Unfortunately, prior works on SDN do not cover carrier-grade networks, i.e., geographically distributed networks with hundreds of nodes like the ones of ISPs. Those networks have special needs. Beyond manageability and flexibility, ISPs operators also have to guarantee high scalability and to preserve network performance upon link and node failure (e.g. to comply with Service Level Agreements). Moreover, the large scale and geographic distribution exacerbates SDN challenges like controller reactivity and controller-to-devices communication. Consequently, SDN solutions targeting campuses [74], enterprises [22], and data-centers [4] cannot be easily ported to carrier-grade networks. Even approaches designed for large area and inter data-centers networks [60, 62, 64] do not fit. Indeed, they assume that (i) the scale of the network is small, (ii) scalability and robustness play a more limited role (e.g. because of the small number of demands [62]), and (iii) that the SDN controller may control the traffic distribution itself [64].

1.3 DECLARATIVE AND EXPRESSIVE FORWARDING OPTIMIZER

DEFO (Declarative and Expressive Forwarding Optimizer) is an SDN-like architecture — that we have designed with researchers from ICTEAM and Cisco Systems — to solve the limitations of current SDN architecture when applied to large carrier-grade networks.

DEFO follows the recent SDN trend of separating TE and connectivity tasks [60, 62, 107] but goes a step further by explicitly separating the responsibility of forwarding packets on three layers (see Figure 1.2):

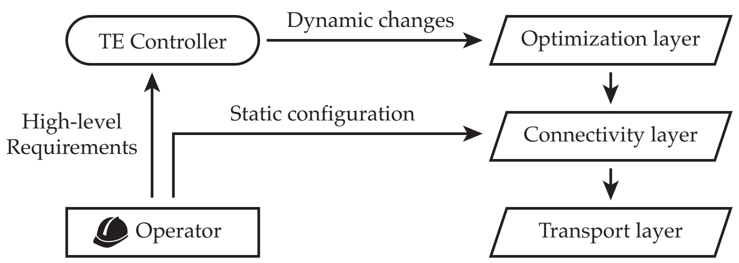


Figure 1.2: DEFO's architecture.

- The *transport layer* corresponds to the physical transfer of the data and is ensured by the network's hardware.

- The *connectivity layer* ensures network-wide reachability, robustness and fast failure recovery by generating and updating connectivity paths for each pair of nodes in the network. In DEFO, we assume that the connectivity layer is maintained by Interior Gateway Protocols (IGPs) which are statically configured by the network operators (the necessary background on IGPs is presented in Chapter 2).

- The *optimization layer* defines optimized TE paths that overwrite the default connectivity paths for specific flows. In DEFO, we rely on the *segment routing* architecture to define such optimized paths (segment routing is presented in details in Chapter 3). It is the responsibility of the DEFO controller to dynamically compute optimized segment routing paths to dynamically maintain network effectiveness.

The DEFO controller is the brain of the whole architecture. To support expressiveness, DEFO exposes a high-level interface that allows network operators to state their network requirement declaratively and to rely on the DEFO controller to translate these requirements into high quality solutions with minimal congestion. Of course, computing paths that realize DEFO goals in a carrier-grade network is far from being easy and requires the use of sophisticated optimization techniques and algorithms.

1.4 CONTENT OF THE THESIS

This thesis focuses on the design of optimization techniques and algorithms to be embedded in the DEFO controller in order to compute high quality network configurations that minimize the network's congestion while respecting the network's requirements.

We start this long journey by presenting the necessary background about Interior Gateway Protocols (i.e. the protocols used in the connectivity layer) and show why IGP alone is not enough to answer the network requirements considered in this thesis.

Chapter 3 presents the segment routing architecture, formalize the Segment Routing Traffic Engineering Problem (SRTEP) and provides the reader with a proof that the SRTEP belongs to the family of NP-Hard problems and is thus computationally difficult to solve.

Chapter 4 focuses on the use of Mixed Integer Linear Programming (MILP) techniques that are commonly used to solve traffic engineering problems optimally. We present two different models to solve the SRTEP. The first one is a direct extension of a model proposed by researcher from the Bell labs. We show that this model is efficient but suffers from important scalability issue when one aims to define segment routing paths with more than one segment. The second model that we suggest aims specifically at avoiding the scalability issues of the first model. We finally present a third model that is the result of mixing both previous models to get the best of both world. Those three models are then used to evaluate segment routing as a technology for TE and its ability to reach optimal routing configurations under different assumptions such as constraints on the connectivity layer.

Chapter 5 investigates the use of Local Search (LS) techniques to solve large instances of the SRTEP in a very short time. We formalize the SRTEP as a LS problem and define two connected neighborhoods with different properties in terms of scalability. Finally we propose two highly scalable LS algorithms and empirically analyze their robustness on a large sample of instances (see Appendix A).

Chapter 6 considers the use of Constraint Programming (CP) and Large Neighborhood Search (LNS) to design a fast and modular LS algorithm that keeps the good scaling behavior of the LS algorithms presented in Chapter 5 but benefits from the declarative aspects that are typical of exact techniques like MILP and CP. To achieve this, we define a new domain representation as well as new propagators which, together, form a new language that extends CP solvers to model the SRTEP in a natural way. In particular, we focus on the design of lightweight structures that allow our algorithm to scale well when applied on large instances of the SRTEP. We finally compare the efficiency of this approach against the best algorithms and models presented in Chapters 4 and 5.

Chapter 7 concludes the thesis by providing general advices to network operators who might want to use the techniques presented in this thesis to implement their own SDN controller. This chapter also discusses different ideas and tracks for further work.

1.5 CONTRIBUTIONS

We briefly summarize the list of the original contributions of this thesis:

- Formalization of the Segment Routing Traffic Engineering Problem (SRTEP) and first proof that it is NP-Hard (Chapter 3).
- Conjecture that IGP weights that respect a strict triangle inequality tend to perform better with segment routing (Chapter 3).
- A new MILP model that does not suffer from the exponential growth of prior models (Chapter 4).
- Encoding of network requirements in the proposed model as well as in existing models (Chapter 4).
- Empirical evaluation of segment routing as a technology that highlights the importance of the connectivity layer and the maximum number of segments (Chapter 4).
- Formalization of the SRTEP as an LS problem (Chapter 5).
- Two highly scalable and robust LS algorithms based on two different neighborhoods (Chapter 5).
- Formalization of the SRTEP as a CP problem (Chapter 6).
- Design of a new domain representation for unbounded string variables (Chapter 6).
- Design of several TE propagators for the previously proposed domain representation (Chapter 6).
- Island removal algorithm to reduce the size of poorly connected SRTEP instances (Appendix A).

1.5.1 *Related publications*

Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication – SIGCOMM 2015*, pages 15-28. ACM, 2015.

Renaud Hartert, Pierre Schaus, Stefano Vissicchio, and Olivier Bonaventure. Solving Segment Routing Problems with Hybrid Constraint Programming Techniques. In *Principles and Practice of Constraint Programming – CP 2015*, pages 149-157. Springer, 2015.

Steven Gay, **Renaud Hartert**, and Stefano Vissicchio. Expect the Unexpected: Sub-second Optimization for Segment Routing. In *INFOCOM 2017 - The 36th Annual IEEE International Conference on Computer Communications*. IEEE, 2017.

1.5.2 *Other publications*

Pierre Schaus, and **Renaud Hartert**. Multi-objective large neighborhood search. In *International Conference on Principles and Practice of Constraint Programming*, pages 611-627. Springer, 2013.

Renaud Hartert and Schaus Pierre. A Support-Based Algorithm for the Bi-Objective Pareto Constraint. In AAAI, pages 2674-2679. 2014.

Steven Gay, **Renaud Hartert**, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 149-157. Springer, 2015.

Steven Gay, **Renaud Hartert**, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 140-148. Springer, 2015.

Steven Gay, **Renaud Hartert**, and Pierre Schaus. Time-table disjunctive reasoning for the cumulative constraint. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 157-172. Springer, 2015.

Jordan Demeulenaere, **Renaud Hartert**, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207-223. Springer, 2016.

Renaud Hartert. Kiwi–A Minimalist CP Solver. In *arXiv*, 2017.

2

BACKGROUND ON IGP TRAFFIC ENGINEERING

Nowadays, most autonomous systems rely on Interior Gateway Protocols (IGPs) to perform such a task. IGPs, like OSPF (Open Shortest Path First) [77] and IS-IS (Intermediate System to Intermediate System) [80], are based on *shortest path routing*. The main idea behind shortest path routing is to associate a weight $w_{u,v} \in \mathbb{Z}^+$ to each link $(u, v) \in \mathbf{E}$ and then to use these weights to compute the shortest paths on which packets are sent.[1]

Example 1. Let us consider that packets have to be forwarded from router s to router t in the network depicted in Figure 2.1. Dashed links represent the links that are not part of the shortest path. The shortest path from s to t has a total cost of 6 and visits routers a, d, b, and t in that order.

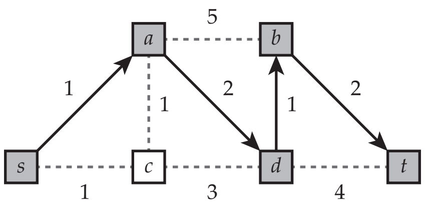
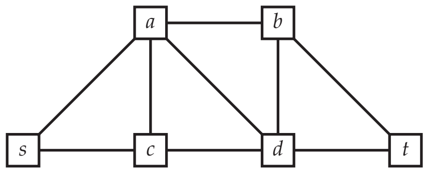


Figure 2.1: Shortest path from router s to router t. Numbers represent the link weights.

In IGPs, routers rely on a decentralized version of the Dijsktra's algorithm to dynamically compute the shortest paths [15]. The particularity of this distributed computation is that no router is aware of the whole network topology. Instead, each router contains a *routing table* that lists the IP address of the next router — the next hop — on the path towards any reachable destination (see Example 2). Routers periodically update their routing table

1 In practice, OSPF allows link weights to be any 16–bits unsigned integer while IS-IS is restricted to 8–bits unsigned integers (though recent versions of IS-IS extend the range of number to 24–bits signed integers [101]).



Destination	Next hop	Weight
a	a	1
b	a	2
c	c	1
d	a	2
t	c	3

Figure 2.2: A possible routing table for router s (with unary link weights).

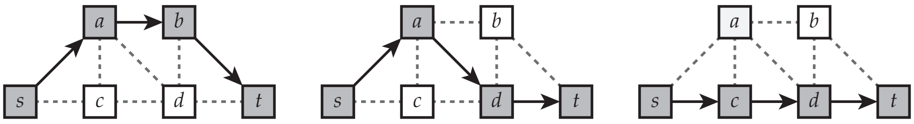
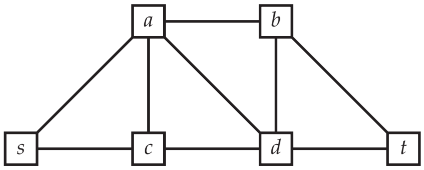


Figure 2.3: Shortest paths from router s to router t.

by sharing information with their direct neighbors. This continuous process enables the automatic detection of changes in the topology, such as a link failure or the shutdown of a router.

Example 2. Let us consider the network and the routing table of router s depicted in Figure 2.2. The shortest path from router s to router b first visits router a which is the next hop from s to b. Likewise, a possible shortest path from router s to router t first visits router c and then router d thus making c the next hop from s to t.

The reader might have observed that there are three different shortest paths of cost 3 between routers s and t in Example 2 (see Figure 2.3). As a matter of fact, multiple shortest paths are quite common in well connected network and, according to Filsfils and al. [39], it is not rare to see as much as 128 different shortest paths between a source and a destination within a single AS's network [39]. Equal Cost MultiPath (ECMP) is a feature that improves load balancing in IGPs by spreading demands over those multiple shortest paths [106]. Precisely, it extends the routing table to contain a list of all the next hops towards a given destination instead of a single one (see Figure 2.4). With those ECMP routing tables, routers now forward packets equally on all the next hops. We formalize this forwarding mechanism as follows.



Destination	Next hop	Weight
a	a	1
b	a	2
c	c	1
d	a, c	2
t	a, c	3

Figure 2.4: Routing table of router s when ECMP is enabled.

Definition 5: ECMP forwarding. Let $d = (s, t, k)$ be a demand from router s to router t and let $w^{path}_{s,t}$ be the weight of the shortest path between those routers. By the transitivity of shortest path, we know that each link (s, u) such that $w_{s,u} + w^{path}_{u,t} = w^{path}_{s,t}$ is part of a shortest path from s to t. Let $next_{s,t}$ be the set of all those routers u:

$$next_{s,t} = \{u \in \mathbf{N} \mid w_{s,u} + w^{path}_{u,t} = w^{path}_{s,t}\}.$$

ECMP forwarding sends exactly $k/|next_{s,t}|$ packets on each edge (s, u) where $u \in next_{s,t}$.

Definition 6: Forwarding Graph. A forwarding graph is a function $FG_{s,t}$: $\mathbf{E} \to [0, 1]$ that represents the flow obtained by sending packets between routers s and t with ECMP:

$$FG_{s,t}(u,v) = \begin{cases} 0 & v \notin next_{u,t} \\ \frac{1}{|next_{s,t}|} & v \in next_{u,t} \wedge u = s \\ \frac{\sum_{m \in \mathbf{N}} FG_{s,t}(m,u)}{|next_{u,t}|} & \text{otherwise.} \end{cases}$$

There is only one forwarding graph for each pair of routers $s, t \in \mathbf{N}$ with $s \neq t$, and the set of all link (u, v) such that $FG_{s,t}(u, v) > 0$ forms a directed acyclic graph.

Forwarding graphs and ECMP forwarding are both illustrated in Figure 2.5. To compute forwarding graphs, we first need to compute the $next_{s,t}$ sets for all pairs of routers s and t. This can easily be achieved by computing the shortest path DAGs using a slightly extended version of the Dijkstra's algorithm. Forwarding graphs are then computed in $\mathcal{O}(|\mathbf{E}|)$ by iterating on the routers of the corresponding DAG in topological order. This process is described in Pseudocode 2.1 in which we assume that the $next_{s,t}$ sets are given.

```
function BUILDFG$(s, t)$:

  for $e \in \mathbf{E}$:
    $FG[e] = 0$

  for $n \in \mathbf{N}$:
    $inflow[n] = 0$
    $deg[n] = 0$

  for $n \in \mathbf{N}$:
    for $v \in next_{n,t}$:
      $deg[v] = deg[v] + 1$

  $Q = \{s\}$
  $inflow[s] = 1$

  while $Q \neq \emptyset$:
    remove a node $u$ from $Q$
    $flow = inflow[u] / |next_{u,t}|$

    for $v \in next_{s,t}$:
      $FG[(u,v)] = flow$
      $inflow[v] = inflow[v] + flow$

      $deg[v] = deg[v] - 1$
      if $deg[v] = 0$:
        $Q = Q \cup \{v\}$

  return $FG$
```

Pseudocode 2.1: Algorithm to compute $FG_{s,t}$ from its shortest path DAG.

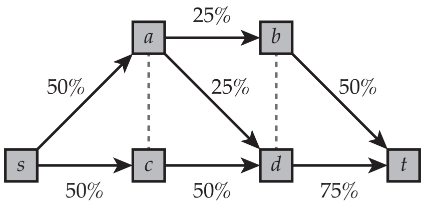


Figure 2.5: ECMP sends packets on all the shortest paths between routers *s* and *t*. Links are labeled with the percentage of packets they carry.

2.1 WEIGHT OPTIMIZATION

The way packets are forwarded in IGP networks is determined by the shortest paths and thus by the link weights. While unary link weights might already be enough to make the network works seamlessly, carefully chosen link weights can substantially improve the network efficiency. For instance, an ISP serving clients with delay critical applications, such as voice calls, is likely to assign small weights to the fastest links. To the contrary, an AS serving clients with high volume of information, such as video streaming, might favor links with large bandwidth. As a rule of thumb, router vendors commonly recommend to set the weight of a link to the inverse of its bandwidth [80].

Choosing link weights that optimize the requirements of network operators turns out to be a singularly complex and difficult task. One of the main reason is that a marginal change of a single link weight can divert many shortest paths and result in an entirely different routing configuration. Consequently, network operators often make mistakes that can drastically impact the performance of their network. In fact, a recent report published by Juniper Systems [78] shows that network downtimes are more likely to be caused by human than by hardware failures — not considering hardware failures that are directly caused by humans, e.g. drunken hunters [28].

IGPs being ubiquitous in IP networks, it is not surprising that the problem of optimizing link weights, best known as IGP Weights Optimization (IGP-WO), has stimulated a lot of interest in the TE community. Unfortunately, IGP-WO proves to be NP-Hard even for simple TE objectives such as minimizing the network utilization [40, 85, 86]. Nonetheless, researchers have devoted much effort in tackling IGP-WO in order to optimize a large range of TE objectives as efficiently as possible. The exhaustive review of those contributions being out of the scope of this thesis, we only briefly mention what we believe to be two of the most successful.

EXACT APPROACH. In [85, 86], Pièro et al. proposed to formulate IGP-WO as a Mixed Integer Linear Programming (MILP) problem. Using this formulation, they were able to compute optimal link weights that result in a minimal network utilization. Though, as it is often the case with MILP problems, this approach suffers from important scalability issues. Indeed, current commercial MILP solvers such as Gurobi 7.0 [52] usually cannot solve IGP-WO optimally on network made of more than a few tens of nodes in reasonable time.

HEURISTIC. A fundamentally different approach based on tabu search [48] has been proposed by Fortz et al. in [40, 41]. To the contrary of the above MILP approach, tabu search does not strive to prove the optimality of its solutions but rather tries to find very good solutions as fast as possible. As a result, tabu search tends to behave very well on large scale problems. The tabu search of Fortz et al. is often considered as the state-of-the-art technique to perform IGP-WO and has been implemented in commercial solvers such as Cisco MATE [24].

2.2 LIMITATIONS OF SHORTEST PATH ROUTING

Despite all their strengths, IGPs suffer from important weaknesses that come from the shortest paths themselves. A first limitation of shortest path routing is that it might not be able to prevent congestion while the network capacity could allow it (see Example 3). A second weakness of shortest path routing is that it cannot differentiate two demands having the same source and destination. This is a particularly important issue if both demands are subject to different service level agreements. For instance, one of the demand might come from a client who is paying to access high-priority links that are not available to his competitors. Last but not least, a third weakness of shortest path routing is that changing the link weights, e.g. to answer a sudden change of traffic, comes with an operational cost and the risk of potential disruptions due to inter-domain routing policies [105, 109]. Therefore, network operators might actually be reluctant to the idea of optimizing link weights — at least dynamically.

Example 3. Let us consider the network depicted in Figure 2.6. There are two demands $d_1 = (s, t, 8)$ and $d_2 = (s, v, 4)$ to forward on that network. All the possible ways to route those demands using shortest path routing with and without ECMP are presented in Figure 2.7. We observe that both demands have to be forwarded on the shortest path from s to c before being routed

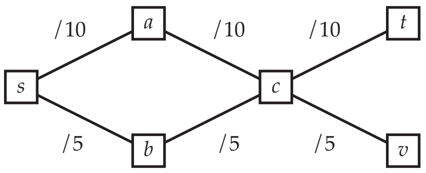


Figure 2.6: A network where the number next to both links (u, v) and (v, u) corresponds to their capacity $c_{u,v}$ and $c_{v,u}$.

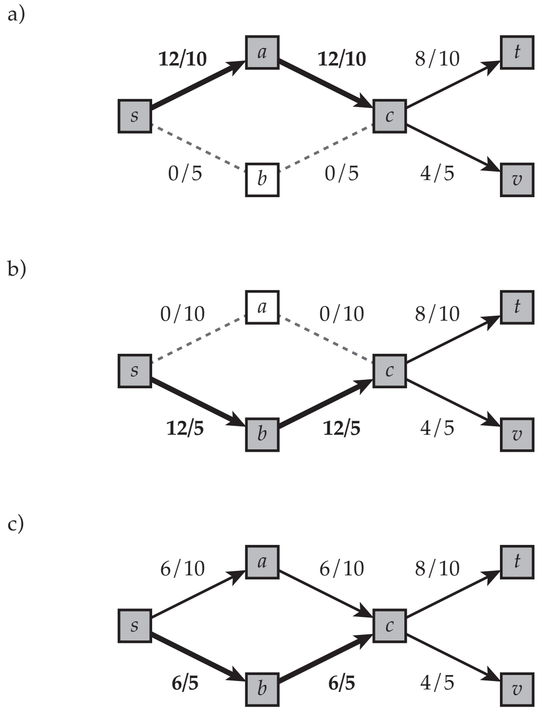


Figure 2.7: There is no way to prevent congestion with shortest path routing.

to their respective destination — thus making c an intermediate destination of both demands. Our first routing configuration (see Figure 2.7 a) sends all the packets from router s to router c on path s, a, c and results in congestion on links (s,a) and (a,c). The second solution (see Figure 2.7 b) sends all the packets on path s, b, c which results in an even worst congestion on links (s,b) and (b,c). Finally the last solution (see Figure 2.7 c) makes use of ECMP to spread the traffic equally on both previous paths. Sadly, even this solution results in congestion on links (s,b) and (b,c) despite the fact that enough resource is available on the network to accommodate both flows. A simple solution to this problem is to send d_1 on path s, a, c, t and to send d_2 on path s, b, c, v.

2.3 CONCLUSION

In this chapter, we have reviewed the core concepts behind IGPs and, more precisely, shortest paths routing as well as its ECMP extension. We have briefly presented the IGP weight optimization traffic engineering problem and reviewed what we believe to be the most important contributions to the field.[2] We finally showed that, and despite the additional flexibility offered by ECMP, IGPs remain constrained by the shortest path routing model and might be unable to avoid congestion when enough resource is available elsewhere in the network.

2 Readers who are interested to learn more about IGP weight optimization may refer to [108] for a detailed overview of this traffic engineering problem.

3

SEGMENT ROUTING

Segment Routing (SR) is an emerging technology, introduced in mid 2013 [39] and developed within the Internet Engineering Task Force (IETF), that is specifically designed for SDN and scalable traffic engineering. In SR, a path is encoded as a sequence of detours, called *segments*, that is stored directly inside the packet headers. Each pair of successive detours in an SR path are connected using the shortest paths defined by the underlying routing protocol. Concretely, when a packet is sent, it first follows one of the shortest path to the first detour, and then follows one of the shortest paths to the second detour and so on until it reaches its destination. The whole definition of an SR path being contained in the header of its packet (instead of storing it in the network infrastructure as done by other traffic engineering protocols such as RSVP-TE [8]), only the ingress router (i.e. the router at which the packet enters the network) is responsible for maintaining the path description. This simple and elegant mechanism is the key that allows SR to avoid the well-known scalability issue and operational challenges of RSVP-TE [76, 114] while keeping an expressive path definition.

Two types of segments, namely *adjacency segments* and *node segments*, can be used to define an SR path. An adjacency segment references a particular interface of a router. When that router is reached, it removes the segment from the top of the header and then forwards the packet on its corresponding interface. The practical interest of adjacency segments is that they can be used to bypass the routing table of a router by forcing it to send packets on any link (see Example 4).

Example 4. We illustrate the use of adjacency segments to forward a flow of packets from router s to router t on the network depicted in Figure 3.1. First, router s removes the top segment of the header and forwards the packets on its link to router c. Note that the link from s to c does not belong to the shortest path from s to c — which visits s, a, and c. Router c then also removes

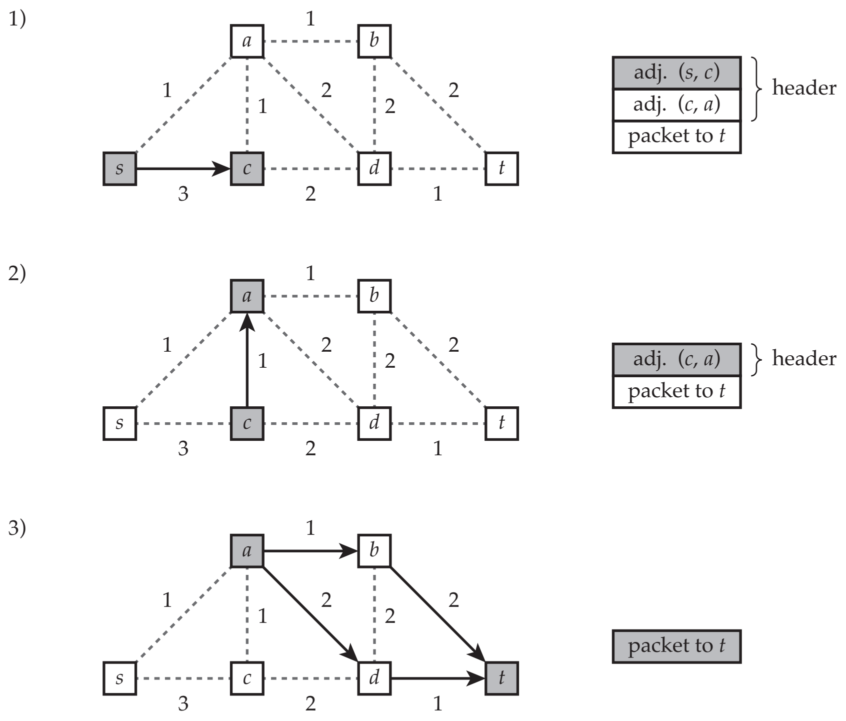


Figure 3.1: Use of adjacency segments to bypass the routing tables. Each link label corresponds to the IGP cost of its link.

the top adjacency segment and forwards the packets on its corresponding interface to router *a*. Finally, router *a* routes the packets on the shortest paths to their destination, i.e., router *t*.

A node segment identifies a specific router in the network that temporarily replace the actual destination of the packet. Once this temporary destination reached, it is removed (or "popped") from the packet's header and the next node segment (if any) becomes the new temporary destination. This process is repeated until no node segment remains, meaning that the packet can finally be sent to its actual destination (see Example 5).

Example 5. Figure 3.2 illustrates the use of node segments to forward a flow of packets from router *s* to router *t* in a way that is impossible to achieve with shortest path routing alone. The first node segment — the one on top of the header — is a reference to router *d* which acts as a temporary destination.

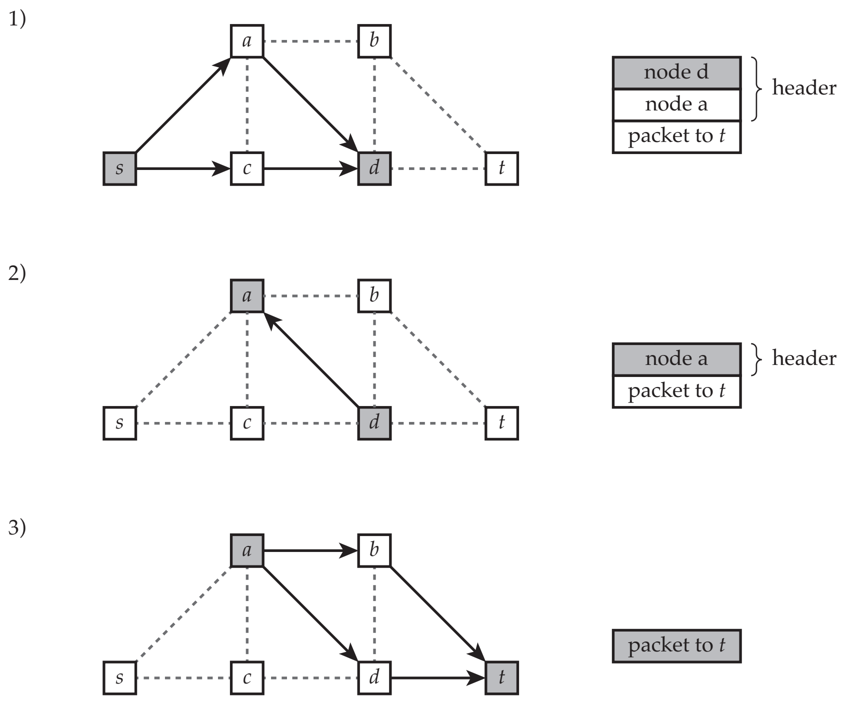
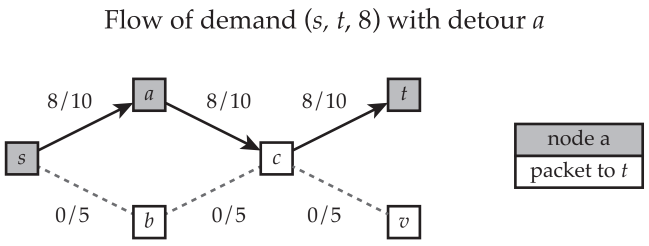


Figure 3.2: Use of node segments to route packets on a sequence of shortest paths. All link weights are set to 1.

Router s thus forwards the packets on the shortest paths to router d. Router d then removes the segment on top of the header and forwards the packets to the next destination, i.e., router a. Finally, router a removes the last segment and forwards the packet to its destination.

Example 6. Example 3 illustrated the limitation of shortest paths routing. SR allows us to prevent congestion (and makes the network operate at 80% of its capacity) by defining distinct paths for both flows (see Figure 3.3).

In the general case, the set of paths that can be constructed using either node or adjacency segments are different. The paths built with node segments essentially depend on the underlying shortest paths. On the contrary, adjacency segments can be used to route a demand on any simple path no matter the shortest paths. Nevertheless, we focus on SR paths only made of



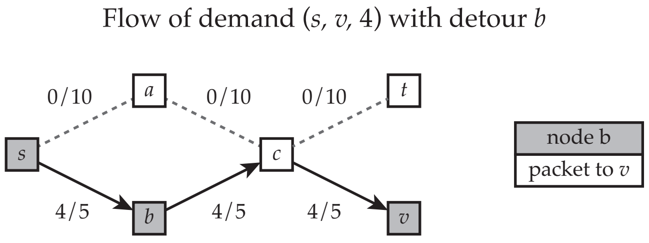


Figure 3.3: Shortest path from router *s* to router *t* where numbers correspond to link weights.

node segments for the following reasons.[1] First, it is also possible to build any simple path with node segments if the shortest path from node u to node v is link (u, v) itself (if it exists). This happens when the link weights respect a *strict triangle inequality* — e.g. unary link weights have that property. Second, node segments are more reliable than adjacency segments in case of failure. If a failure occurs, the routing protocol will automatically detect it and recompute the shortest paths, thus ensuring that an SR path with node segments still connects its source to its destination. Last but not least, node segments benefit from ECMP and thus favor load balancing. We formalize SR paths as follows.

Definition 7: Segment Routing Path. An SR path $p_{s,t}$ from node s to node t is represented by a sequence of nodes

$$s = m_0, m_1, \dots, m_{i-1}, m_i, m_{i+1}, \dots, m_k, m_{k+1} = t$$

in which the subsequence $m_1, \dots, m_k$ is the sequence of node segments, called *midpoints*, towards the path destination t. Each pair of successive nodes (m_i, m_{i+1}) in an SR path is connected by a forwarding graph $FG_{m_i,v_{m+1}}$ such that the SR path can also be represented as a sequence of forwarding graphs

$$FG_{s,m_1}, \dots, FG_{m_{i-1},m_i}, FG_{m_i,m_{i+1}}, \dots, FG_{m_k,t}.$$

We subsequently denote $FG(p_{s,t})$ the sequence of forwarding graphs that are part of $p_{s,t}$. The link between $p_{s,t}$ and $FG(p_{s,t})$ is illustrated in Figure 3.4. By abuse of notation, we often use $FG(p_{s,t})$ as a set of forwarding graphs.

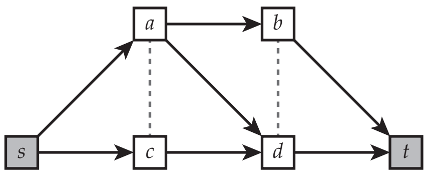
3.1 THE SEGMENT ROUTING TRAFFIC ENGINEERING PROBLEM

The Segment Routing Traffic Engineering Problem (SRTEP) is the natural specialization of the general traffic engineering problem to the SR flavor. The problem is thus the one of finding an SR path for each demand such that the network operates below a given maximum network utilization. It is formalized as follows.

The inputs of the problem are a network $T(\mathbf{N}, \mathbf{E})$, a set of demands $\mathbf{D}$, a maximum network utilization $\phi \in \mathbb{R}^+$, and a forwarding graph $FG_{s,t}$ for each

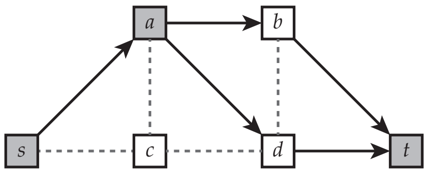
1 Note that all the techniques developed in the subsequent chapters can be extended to handle adjacency segments as well by adding fake nodes, that will simulate adjacency segments, to the network's topology.

$$p_{s,t} = s, t$$



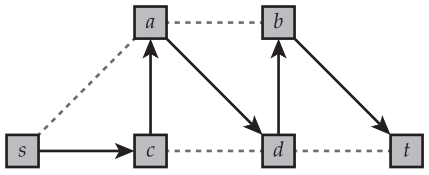
$$FG(p_{s,t}) = FG_{s,t}$$

$$p_{s,t} = s, a, t$$



$$FG(p_{s,t}) = FG_{s,a}, FG_{a,t}$$

$$p_{s,t} = s, c, a, d, b, t$$



$$FG(p_{s,t}) = FG_{s,c}, FG_{c,a}, FG_{a,d}, FG_{d,b}, FG_{b,t}$$

Figure 3.4: Midpoint and forwarding graph representations of three SR paths. The forwarding graphs correspond to the ECMP computed with unary link weights.

pair of nodes $s, t \in \mathbf{N}$ with $s \neq t$.[2] The load of link $(u, v) \in \mathbf{E}$, denoted $load_{u,v}$, is the sum of all the flows that traverse that link:

$$load_{u,v} = \sum_{d=(s,t,k)\in\mathbf{D}} \; \sum_{FG_{i,j}\in p_d} FG_{i,j}(u,v)k.$$

To solve the SRTEP, one must find an SR path for each demand $d \in \mathbf{D}$ such that:

$$\forall(u,v) \in \mathbf{E} \, : \, load_{u,v} \leq \phi c_{u,v}.$$

In its optimization form, the goal of the SRTEP is to find a solution with the minimal value of ϕ.

Theorem 1. *The SRTEP is NP-Hard.*

Proof. We show that the SRTEP is NP-Hard by a simple reduction of the well known *partition problem* [25]. The partition problem is an NP-complete problem that consists of n numbers $c_1, \ldots, c_n \in \mathbb{N}$. The question is whether there is a set $A \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i\in A} c_i = \sum_{j\in\overline{A}} c_j$$

where $\overline{A}$ is the set of elements not contained in A. This problem can easily be reduced to the instance of the segment routing problem depicted in Figure 3.5 with all link capacities fixed to $\sum_{i=1}^{n} c_i/2$. First, consider n demands $(s, t, c_i), \ldots, (s, t, c_n)$. Then, consider that segments are defined such that there are only two possible SR paths from node s to node t (see the left and right parts of Figure 3.5). Finally, let us constrain the maximum load of each link to not exceed the link capacity. Finding a valid SR path for each demand in this context amounts to find a solution to the Partition problem, i.e., demands having s, A, t as SR path are part of the set A while the remaining demands are part of the set $\overline{A}$. $\square$

3.2 ADDITIONAL CONSTRAINTS

Minimizing the network utilization is not the only goal of network operators. Internet service providers deliver different kind of services that are subject to different levels of agreement. Video and audio calls, for instance, must

2 As mentioned in Definition 6, we assume that forwarding graphs match the ECMP computed accordingly to the network link weights.

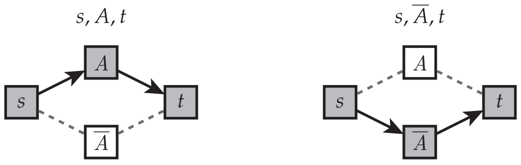


Figure 3.5: The partition problem encoded as an instance of the SRTEP. Forwarding graphs are such that there are only two possible SR paths from node s to node t.

be carried as fast as possible to ensure a good user experience. Demands containing suspicious packets (e.g. coming from a DoS attack) should be forwarded through firewalls spread over the networks while high-priority demands should be forwarded through load-balancers [90]. Network operators might also agree to forward privacy sensitive demands on link disjoint SR paths thus reducing the risk of being spied on by sending the message and its encryption key on two different SR paths. Beside, all those requirements must be fulfilled with a minimum number of SR tunnels to enable fast reconfiguration when dynamic events, such as link failures or substantial traffic changes, will occur. For all these reasons, we propose to use the following additional constraints to extend the SRTEP to meet a large set of network operators requirements.

3.2.1 *Max Cost*

Let C^{FG} be a set of costs such that $C^{FG}_{s,t} \in \mathbb{R}^+$ is the cost of forwarding graph $FG_{s,t}$ ($\forall s, t \in \mathbf{N}$, $s \neq t$). The `maxCost`(p, C^{FG}, k) constraint ensures that the cost of SR path p does not exceeds the maximum cost k:

$$\sum_{FG_{s,t} \in FG(p)} C^{FG}_{s,t} \leq k. \tag{1}$$

Latency is an obvious cost that network operators might want to associate to an SR path but it is not the only one. The number of segments that can be encoded into the packet header is likely to be limited too due to hardware restrictions. In this context, the cost of each forwarding graph is simply set to one and $k + 1$ is the maximum number of detours.

3.2.2 *Service Chaining*

We represent a service $S \subset \mathbf{N}$ as a subset of nodes that provide a particular service (e.g. a firewall) to any demand making a detour to a node $m \in S$. The $\texttt{serviceChaining}(p, S_1, \dots, S_k)$ constraint ensures that SR path p visits an ordered list of services on its way from its source s to its destination t such that

$$p = s, \dots, m_1, \dots, m_2, \dots, m_k, \dots, t$$

where $m_i \in S_i$ for all $i \in \{1, \dots, k\}$.

3.2.3 *Disjoint Paths*

The $\texttt{disjoint}(p_1, p_2)$ constraint ensures that SR paths p_1 and p_2 do not visit the same network links:

$$\forall (u,v) \in \mathbf{E} : \sum_{FG_{s,t} \in FG(p_1)} FG_{s,t}(u,v) = 0 \vee \sum_{FG_{s,t} \in FG(p_2)} FG_{s,t}(u,v) = 0.$$

Note that links (u, v) and (v, u) are considered to be different links.

3.3 CONCLUSION

This chapter focuses on Segment Routing (SR), an emerging architecture specifically designed for SDN and scalable traffic engineering. We briefly introduced the core concepts of SR such as adjacency and node segments. Then, we showed that node segments typically offer more advantages that adjacency segment because they tend to improve load balancing, have a better behavior in case of a link failure, and can be used to define any simple path in the network as long as the underlying IGP weights respect a strict triangle inequality. Next, we formalized the Segment Routing Traffic Engineering Problem (SRTEP) as an optimization problem and showed that this problem is NP-Hard (and thus computationally difficult to solve). We concluded by presenting different constraints that could be used to customize the SRTEP in order to handle specific goals and requirements from network operators.

4 INTEGER PROGRAMMING AND RELAXATIONS

Linear Programming (LP) is a fundamental tool to solve optimisation problems. In LP, an optimisation problem is expressed by a set of n decisions variables, a set of m linear inequalities that represent the constraints of the problem, and by a linear objective function to minimise. The *linear formulation* of a problem, also called *linear program* or *linear model*, is formulated as follows:

$$\begin{array}{lll}
\texttt{minimise} & c_1\mathbf{x}_1 + \cdots + c_n\mathbf{x}_n & \\
\texttt{subject to} & a_{11}\mathbf{x}_1 + \cdots + a_{1n}\mathbf{x}_n & \leq b_1 \\
 & \vdots & \\
 & a_{m1}\mathbf{x}_1 + \cdots + a_{mn}\mathbf{x}_n & \leq b_m \\
\texttt{and} & \mathbf{x}_i \in \mathbb{R} & \forall i \in \{1, \ldots, n\}
\end{array}$$

where the $\mathbf{x}_i$ are the decision variables and the $a_{j,i}$, b_j, and c_i correspond to constants in $\mathbb{R}$.

What makes LP so interesting in practice is that most linear programs can be solved efficiently using algorithms such as the simplex algorithm [26] or the interior point algorithm [2]. LP is however not able to solve problems that require some of their decisions variables to be assigned to integer values only. Mixed Integer Linear Programming (MILP) is an extension of LP that considers such integer variables as well as linear variables. Let's assume that $\mathbf{I} \subseteq \{1, \ldots, n\}$ is the set of the integer variables and that $\mathbf{L} = \{1, \ldots, n\} \setminus \mathbf{I}$ is the set of the linear variables. The general form of a MILP linear program is the following:

$$
\begin{array}{lll}
\texttt{minimise} & c_1\mathbf{x}_1 + \cdots + c_n\mathbf{x}_n & \\
\texttt{subject to} & a_{11}\mathbf{x}_1 + \cdots + a_{1n}\mathbf{x}_n & \leq b_1 \\
 & \vdots & \\
 & a_{m1}\mathbf{x}_1 + \cdots + a_{mn}\mathbf{x}_n & \leq b_m \\
\texttt{and} & \mathbf{x}_i \in \mathbb{R} & \forall i \in \mathbf{L} \\
 & \mathbf{x}_i \in \mathbb{Z} & \forall i \in \mathbf{I}
\end{array}
$$

Unfortunately, the simple addition of integer decisions variables makes the MILP problem NP-Hard. Nevertheless, many techniques and algorithms (such as branch-and-bound, branch-and-cut, or branch-and-price) have been developed during the last decades to tackle MILP problems as efficiently as possible [113]. All these approaches aim at solving problems optimally or at least to provide one with a lower bound on the value of the optimal solution.

Flow problems can be encoded particularly easily with LP and MILP [3]. It is therefore not surprising that they both have a long history of being used to solve traffic engineering problems. This chapter is dedicated to LP and MILP approaches to solve the Segment Routing Traffic Engineering Problem (SRTEP). We first review two approaches to solve the general traffic engineering problem — better known as the multi-commodity flow problem — and explain how it is related to the SRTEP. We then present two MILP models (one being adapted from related work [17]) to solve the SRTEP. Finally, we use both MILP models to analyse the capability of segment routing as a traffic engineering technology to minimize the network utilisation with a minimal number of tunnels.

4.1 BACKGROUND ON MULTI-COMMODITY FLOW

The Multi-Commodity Flow Problem (MCFP) is one of the most studied problem of traffic engineering and is for the matter often referred to as the *general traffic engineering problem* (see Chapter 1). Basically, the MCFP is the problem of finding an optimal way to forward packets on the network without being restricted by the expressivity of routing protocols — such as being forced to follow the shortest paths. Solving the MCFP thus provides operators with an optimistic view, i.e. a lower bound, on how much network bandwidth can be saved with perfect traffic engineering.

4.1.1 *Per Demand Formulation*

We formulate the MCFP as a LP program using a similar notation to the one introduced in Chapter 3. The network is a strongly connected directed graph made of a set of nodes $\mathbf{N}$ and of a set of directed edges $\mathbf{E}$ that respectively represent the routers and the links of that network. Each edge $(u, v) \in \mathbf{E}$ is associated with a capacity denoted $c_{u,v}$. The demands to be forwarded on the network are contained in a traffic matrix $\mathbf{D} = \{(s, t) \mid s, t \in \mathbf{N} \wedge s \neq t\}$ and the bandwidth requirement of each demand $(s, t) \in \mathbf{D}$ is denoted $\texttt{bw}_{s,t} \in \mathbb{R}^+$. The linear variable $\mathbf{f}^{s,t}_{u,v} \in \mathbb{R}^+$ denotes the amount of traffic from demand $(s, t) \in \mathbf{D}$ that is routed through edge $(u, v) \in \mathbf{E}$. The network utilisation to minimize is denoted $\phi \in \mathbb{R}^+$ and constrains the total amount of data traversing each edge $(u, v) \in \mathbf{E}$ to not exceed $\phi c_{u,v}$. Finding an optimal solution to the MCFP consists in finding the lowest possible value for ϕ such that all the following constraints are respected:

$$\sum_{u \in \mathbf{N}} \mathbf{f}^{s,t}_{s,u} = \sum_{u \in \mathbf{N}} \mathbf{f}^{s,t}_{u,t} = \texttt{bw}_{s,t} \qquad \forall (s,t) \in \mathbf{D} \qquad (2)$$

$$\sum_{u \in \mathbf{N}} \mathbf{f}^{s,t}_{u,m} = \sum_{u \in \mathbf{N}} \mathbf{f}^{s,t}_{m,u} \qquad \forall (s,t) \in \mathbf{D}, \forall m \in \mathbf{N}, m \neq s,t \qquad (3)$$

$$\sum_{(s,t) \in \mathbf{D}} \mathbf{f}^{s,t}_{u,v} \leq \phi c_{u,v} \qquad \forall (u,v) \in \mathbf{E}. \qquad (4)$$

Linear equations (2) and (3) are called *flow conservation* constraints.[1] They guarantee that no packet is lost or created along the paths from their source to their destination. In particular, linear equations (2) ensure that the amount of packets that leave the source of a demand reach its destination while linear equations (3) ensure that all packets reaching any intermediate node m leave this node. Equations (4) are called *capacity* constraints and guarantee that the utilization of link (u, v) does not exceed the maximum network utilization ϕ.

4.1.2 *Per Destination Formulation*

It is possible to significantly improve the solving process of the MCFP if one does only care about the minimal value of ϕ and not how the packets

1 The flow conservation constraints are exactly the same as the ones we used in the definition of forwarding graph in Chapter 3.

are actually forwarded on the network [65]. The idea is to aggregate the flow variables by destination such that variable $\mathbf{f}^t_{u,v} \in \mathbb{R}^+$ corresponds to the amount of traffic destined to node t that is routed through edge (u, v). The linear constraints are the following:

$$\sum_{u \in \mathbf{N}} \mathbf{f}^t_{m,u} = \sum_{v \in \mathbf{N}} \mathbf{f}^t_{v,m} + \mathsf{bw}_{m,t} \qquad \forall m, t \in \mathbf{N}, m \neq t \qquad (5)$$

$$\sum_{t \in \mathbf{N}} \mathbf{f}^t_{u,t} \leq \phi c_{u,t} \qquad \forall (u, t) \in \mathbf{E}. \qquad (6)$$

The first set of linear constrains (5) corresponds to the flow conservation constraints. The second set of linear constraints (6) is similar to (4) and ensures that link (u, v) operates below the network utilization.

The attentive reader might have been wondering why no inequality constrains the value of the variables $\mathbf{f}^t_{t,v}$ $(\forall v \in \mathbf{N})$. The reason for this absence is simply that there is no demand from a node to itself. The only way to assign one of these variables to a positive value is to introduce a cycle in the network. While this may sound bad at first, it is easy to transform any solution with a cycle into a solution with no cycle. Also, the LP solving process will automatically remove all the cycles that impact the objective value, thus returning the optimal network utilization regardless of the presence of a cycle.

Both linear programs, *per demand* and *per destination*, have been extensively analysed and showed to be equivalent in [63]. However, as mentioned before, the *per destination* formulation is usually much faster to solve due to the smaller number of variables and of linear equations required to formulate the MCFP.[2] Table 1 shows the difference between both models. We assume that the network is sparse such that the number of edges is proportional to the number of nodes, i.e., $\mathcal{O}(|\mathbf{E}|) \simeq \mathcal{O}(|\mathbf{N}|)$. Also, we assume that we have a demand between every pair of distinct nodes in the network such that $|\mathbf{D}| = |\mathbf{N}|^2 - |\mathbf{N}| = \mathcal{O}(|\mathbf{N}|^2)$.

4.2 THE PATH MODEL

Solving the MCFP provides us with an optimistic view of how much network resource can be spared in optimal traffic engineering conditions. Of course,

2 Note that Jones et al. showed in [63] that the *per demand* formulation can be faster to solve with techniques based on Dantzig-Wolfe decomposition [11].

Model	#variables	#inequalities
Per demand	$\mathcal{O}(\lvert\mathbf{D}\rvert\lvert\mathbf{E}\rvert) \simeq \mathcal{O}(\lvert\mathbf{N}\rvert^3)$	$\mathcal{O}(\lvert\mathbf{D}\rvert\lvert\mathbf{N}\rvert) \simeq \mathcal{O}(\lvert\mathbf{N}\rvert^3)$
Per destination	$\mathcal{O}(\lvert\mathbf{N}\rvert\lvert\mathbf{E}\rvert) \simeq \mathcal{O}(\lvert\mathbf{N}\rvert^2)$	$\mathcal{O}(\lvert\mathbf{N}\rvert^2)$

Table 1: Number of variables and constraints involved in the *per demand* and *per destination* models.

those perfect conditions are often far from reality which has to deal with software and hardware limitations. In the context of segment routing, those limitations mainly come from the underlying routing protocol which defines the shortest paths used to build the SR paths. The Segment Routing Traffic Engineering Problem (SRTEP) can thus be seen as a more constrained version of the MCFP.

In [17], Bhatia et al. proposed a MILP model to solve the 2-SRTEP, i.e. a more constrained version of the SRTEP in which SR paths are made of at most two forwarding graphs (or a single midpoint). This model, that we subsequently refer to as the *path model*, shines by its simplicity. It comes down to precomputing all the possible SR paths a demand can be forwarded on and to assigning each demand to exactly one of those paths such that the network utilization is minimized. SR paths being restricted to at most one detour, each demand can be routed on exactly $|N| - 1$ SR paths, i.e. one with no midpoint and $|N| - 2$ with a single midpoint. We present a generalized version of the *path model* in which a demand can be routed on any SR path no matter the number of midpoints. This extension is rather trivial since it basically consists in precomputing more SR paths. Naturally, the number of possible SR paths grows exponentially with the maximum number of midpoints but let us assume that this is not an issue for the moment.

Let P_d be the set of all SR paths demand d can be routed on, and let $\texttt{flow}_p : \mathbf{E} \rightarrow \mathbb{R}^+$ be a function that returns the quantity of flow that traverses any edge $(u, v) \in \mathbf{E}$ when demand d is forwarded on SR path p:

$$\forall d \in \mathbf{D}, \forall p \in P_d : \quad \texttt{flow}_p(u,v) = \texttt{bw}_d \sum_{FG_{s,t} \in FG(p)} FG_{s,t}(u,v).$$

Note that $\texttt{flow}_p(u, v)$ might be higher than $\texttt{bw}_d$ if SR path p sends demand d several times on (u, v), e.g. if there's a cycle in p.

We associate a binary variable $\mathbf{x}_p^d \in \{0,1\}$ to each demand d and path $p \in P_d$ that is assigned to 1 if demand d is forwarded on SR path p or set to 0 otherwise. The SRTEP is modeled as follows:

$$\sum_{p \in P_d} \mathbf{x}_p^d = 1 \qquad \forall d \in \mathbf{D} \tag{7}$$

$$\sum_{d \in \mathbf{D}} \sum_{p \in P_d} \mathbf{x}_p^d \, \texttt{flow}_p(u,v) \leq \phi c_{u,v} \qquad \forall (u,v) \in \mathbf{E} \tag{8}$$

Equation (7) enforces each demand to be forwarded on a single SR path. Equation (8) computes the load of each edge (u,v) as the sum of all the data that is forwarded through (u,v) and constrains its link utilization to not exceed the network utilization ϕ.

The problem of the *path model* appears when we start remembering that the number of SR paths grows exponentially with the number of midpoints. This is probably fine if SR paths are limited to one or two midpoints and simple constraints but it is likely to be the source of important scalability issues otherwise. Let us consider the k-SRTEP in which k is the maximum number of forwarding graphs that an SR path can contain.[3] The number of possible SR paths for a demand is $\sum_{i=0}^{k-1}(|\mathbf{N}|-2)^i$ and has an asymptotical growth of $\mathcal{O}(|\mathbf{N}|^{k-1})$. The *path model* thus requires $\mathcal{O}(|\mathbf{D}||\mathbf{N}|^{k-1})$ binary variables, $\mathcal{O}(|\mathbf{E}|)$ linear variables, and $\mathcal{O}(|\mathbf{D}|)$ linear equations.[4]

4.3 THE SEGMENT MODEL

We propose a slightly more complex model, called *segment model*, that does not suffer from the scalability issue of the *path model* and can handle any SR path that does not traverse the same forwarding graph more than once.[5] The *segment model* is based on the fact that finding an SR path can be seen as finding a path in a clique of midpoints, called *segment graph*, in which each edge corresponds to a forwarding graph (see Figure 4.1).

Let $\mathbf{x}_{u,v}^d$ be a binary variable that is assigned to 1 if the SR path of demand d traverses forwarding graph $FG_{u,v}$ or assigned to 0 otherwise. We model the SRTEP as follows:

$$\sum_{u \in \mathbf{N}} \mathbf{x}_{s,u}^d = \sum_{u \in \mathbf{N}} \mathbf{x}_{u,t}^d = 1 \qquad \forall d = (s,t) \in \mathbf{D} \tag{9}$$

3 The 1-SRTEP being the "problem" of routing the demands on the shortest paths only.
4 Such simple models with an exponential number of variables are usually well suited for Dantzig-Wolfe decomposition techniques such as column generation [11].
5 Getting rid of this limitation is relatively trivial and left as an exercice for the reader.

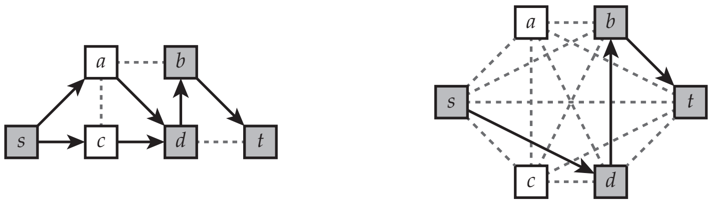


Figure 4.1: An SR path (left) can be seen as a path in the segment graph (right). Edges in the segment graph correspond to forwarding graphs.

$$\sum_{v \in \mathbf{N} \setminus \{u\}} \mathbf{x}^d_{v,u} = \sum_{v \in \mathbf{N} \setminus \{u\}} \mathbf{x}^d_{u,v} \qquad \forall d = (s,t) \in \mathbf{D}, \forall u \in \mathbf{N} \setminus \{s,t\} \tag{10}$$

$$\sum_{d \in \mathbf{D}} \sum_{s,t \in \mathbf{N}, s \neq t} \mathbf{x}^d_{s,t}\, FG_{s,t}(u,v)\, \mathtt{bw}_d \leq \phi c_{u,v} \qquad \forall (u,v) \in \mathbf{E} \tag{11}$$

Constraints (9) and (10) are basically a binary version of the flow conservation constraints from (2) and (3). Precisely, constraint (9) specifies that a single forwarding graph could leave (resp. enter) the source (resp. destination) of a demand, while constraint (10) ensures that if a forwarding graphs enters a node then it must leave that node. In other words, constraints (9) and (10) model SR paths as sequences of forwarding graphs, i.e. as simple paths in the segment graph. Constraint (11) ensures that the link utilization of each edge (u,v) does not exceed the network utilization.

Similarly to the MCFP, every solution of the *segment model* that contains an SR path with a cycle, i.e. its path in the segment graph is not simple, can be turned into an as good solution (in terms of network load) that contains no cycle. The *segment model* thus implicitly models the k-SRTEP where SR paths are limited to $k = |\mathbf{N}| - 1$ forwarding graphs. We can easily restrict the model to any value of $k < |\mathbf{N}| - 1$ by adding the following constraint:

$$\sum_{u,v \in \mathbf{N}, u \neq v} \mathbf{x}^d_{u,v} \leq k \qquad \forall d \in \mathbf{D} \tag{12}$$

Contrarily to the *path model*, this additional constraint allows us to handle multiple SR path length without changing the spacial complexity of the model. Indeed, the *segment model* requires $\mathcal{O}(|\mathbf{D}||\mathbf{N}|^2)$ binary variables, $\mathcal{O}(|\mathbf{E}|)$ linear variables, and $\mathcal{O}(|\mathbf{D}||\mathbf{N}|)$ inequalities no matter the value of k.

Despite its exponential behavior, the *path model* still remains smaller than the *segment model* when applied to the 2-SRTEP. In fact, the *path model* of the 2-SRTEP can be seen as a simplified *segment model* in which useless variables,

Model	#binary variables	#linear variables	#inequalities
Path	$\mathcal{O}(\|\mathbf{D}\|\|\mathbf{N}\|^{k-1})$	$\mathcal{O}(\|\mathbf{E}\|)$	$\mathcal{O}(\|\mathbf{D}\|)$
Segment	$\mathcal{O}(\|\mathbf{D}\|\|\mathbf{N}\|^{2})$	$\mathcal{O}(\|\mathbf{E}\|)$	$\mathcal{O}(\|\mathbf{D}\|\|\mathbf{N}\|)$

Table 2: Number of variables and inequalities in the *path* and *segment* models.

i.e. variables that are assigned in any case, have been removed. By definition, any SR path in the 2-SRTEP is made of at most one single midpoint. Therefore, all variables that connect two midpoints, i.e. every variable $\mathbf{x}^d_{u,v}$ where u and v are different from the source and the destination of d, are guaranteed to be set to 0 and can thus be removed from the model. This removal drastically reduces the size of constraint (10) which becomes the following:

$$\mathbf{x}^d_{s,m} = \mathbf{x}^d_{m,t} \qquad \forall d = (s,t) \in \mathbf{D}, \forall m \in \mathbf{N} \setminus \{s,t\} \tag{13}$$

Every pair of variables $\mathbf{x}^d_{s,m}$ and $\mathbf{x}^d_{m,t}$ can thus be substituted by a single variable $\mathbf{x}^d_m$ such that there is exactly one variable per SR path per demand — as with the *path* model.

Table 2 compares the size of the *path* and *segment* models. As mentioned above, we see that the *path model* is the smallest one for the 2-SRTEP (i.e. when $k = 2$) and should be preferred over the *segment model* when solving this problem. Both models are similar when applied to the 3-SRTEP but the *path model* gets exponentially larger than the *segment model* when we consider problems with more than two midpoints, i.e., when solving the k-SRTEP with $k \geq 4$.

4.4 LINEAR RELAXATIONS

A lower bound of a problem is an optimistic value that is guaranteed to be lower or equal to the optimal objective value of that problem. Lower bounds are essential to minimization problems because they allow one to evaluate the quality of a solution by measuring the distance between its value and the lower bounds. Various types of lower bounds are likely to exist for a given problem and the best of them, the ones with the highest value, are usually the most expensive to compute.

The linear relaxation of a MILP model is an LP model obtained by substituting all the integer variables by linear ones. The interest of linear relaxation is that their optimal solution is a lower bound of the original MILP model and that they can be computed quite efficiently (see the introduction of this chapter). Since MILP solvers strongly rely on linear relaxation, models that have a strong linear relaxation, i.e. a relaxation that provides a lower bound that is close to the optimal objective value, are usually easier to solve.

We have seen in the previous section that the *path* and *segment* models of the 3-SRTEP have similar sizes. In this section, we compare the efficiency of both models by comparing the optimal value of their *linear relaxation* for the 3-SRTEP.

4.4.1 *Path and Segment Linear Relaxations*

Basically, the linear relaxation of the *path* model allows each demand to be split arbitrarily on several SR paths, each one being made of at most k forwarding graphs. The linear relaxation of the *segment* model goes one step further and also relax the length constraint (12) thus allowing SR paths with more than k forwarding graphs. The linear relaxation of the *path* model thus provides better lower bounds than the one of the *segment* model. We verify this statement empirically on all the instances of our dataset that have less than 40 nodes and no bottleneck link (see Appendix A).[6] Let lb_{path} and $lb_{segment}$ be the value of the linear relaxation of the *path* and *segment* models respectively. The comparison of lb_{path} and $lb_{segment}$ on the 3-SRTEP is presented in Figure 4.2. As expected, we see that $lb_{segment}$ is never better than lb_{path}. However, lb_{path} barely higher than $lb_{segment}$ on most of our test cases. Indeed, both linear relaxations return the same lower bound for 77–78% of the instances. Also, lb_{path} is less than 0.5% higher for 99% of the cases and is only 9% better in the most extreme case.

Ultimately, both linear relaxations are similar in practice despite a marginal advantage for the *path* model.[7] Therefore, we subsequently rely on the *path* model to solve the k-SRTEP when $k \leq 3$ and rely on the *segment* model when $k \geq 4$. We denote this linear relaxation lb_{SRTEP}:

$$lb_{SRTEP} = \begin{cases} lb_{path} & k \leq 3 \\ lb_{segment} & \text{otherwise.} \end{cases}$$

6 We focus on instances with less than 40 nodes for the sake of scalability.

7 However, — and this is quite counterintuitive — we observed that Gurobi 7.0 [52], the MILP solver we used to compute all the results in this chapter, is usually faster on the *segment* model.

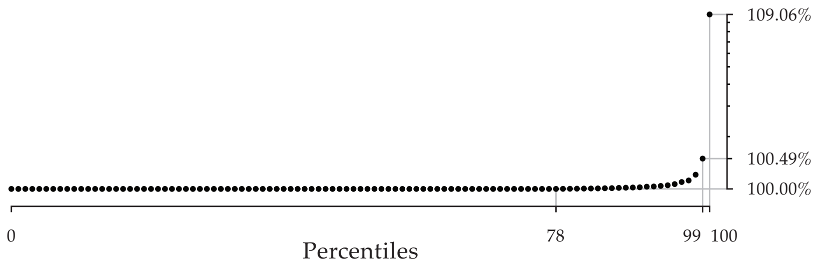


Figure 4.2: Comparison of lb_{path} and $lb_{segment}$ on the 3-SRTEP. The vertical axis is in log scale and corresponds to the value of $lb_{path}/lb_{segment}$ for each percentile. The 78th percentile is the first to exceed 100%.

4.4.2 *MCFP vs Path and Segment Linear Relaxations*

As mentioned at the beginning of this chapter, solving the MCFP already provides us with a lower bound of the SRTEP. Let lb_{MCFP} denote the optimal value of the MCFP. Obviously, lb_{MCFP} is worse than lb_{SRTEP} because it isn't constrained to follow the shortest paths at all (see Section 4.1). But is lb_{MCFP} really weaker in practice? We compare lb_{MCFP} with lb_{SRTEP} on the k-SRTEP with $k = 2$ and $k = |\mathbf{N}| - 1$. Note that all the demands of our dataset have been rescaled such that the optimal value of the MCFP is 100% (see Appendix A). Consequently, the ratio lb_{SRTEP}/lb_{MCFP} always corresponds to the value of lb_{SRTEP}.

The comparison of lb_{SRTEP} and lb_{MCFP} on the 2-SRTEP is presented in the left part of Figure 4.3. We see that lb_{SRTEP} is higher than lb_{MCFP} for 25% of all the tested instances. Particularly, lb_{SRTEP} becomes substantially higher for 5% of the instances and reaches 606.84% in the most extreme case. The right part of Figure 4.3 presents the comparison between lb_{SRTEP} and lb_{MCFP} on the k-SRTEP. The results are difficult to differentiate from the one presented in the left part of Figure 4.3, but lb_{SRTEP} returns slightly lower values between its 75th and 90th percentiles. As before, lb_{SRTEP} is substantially higher for 5% of the instances and reaches 606.84% of lb_{MCFP} in the best case.

In conclusion, lb_{SRTEP} is a stronger lower bound, both in practice and in theory, than the lb_{MCFP}. Nevertheless, lb_{SRTEP} is much more expensive to compute than lb_{MCFP} due to the additional number of variables and inequalities involved in its LP model (see Tables 1 and 2). This is especially true when solving the k-SRTEP with $k \geq 3$. The MCFP thus remains an excellent tool

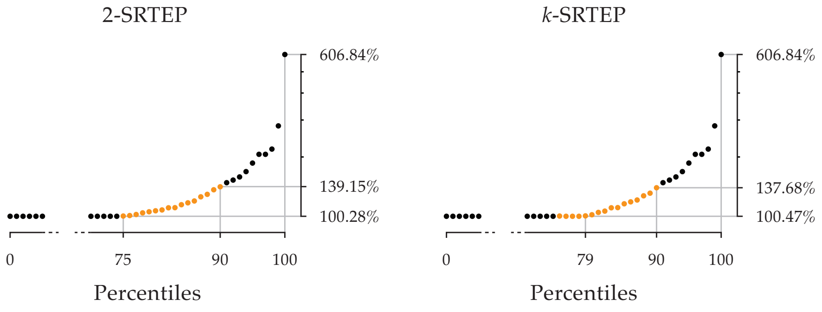


Figure 4.3: Comparison of lb_{SRTEP} and lb_{MCFP} on the 2-SRTEP (left) and the k-SRTEP where $k = |\mathbf{N}| - 1$ (right). The vertical axis is in log scale and corresponds to the value of lb_{SRTEP}/lb_{MCFP} for each percentile. The 75th (resp. 79th) percentile is the first to exceed 100% on the left (resp. right) part of the figure. The difference between both curves is highlighted in orange.

to evaluate the minimum utilization of networks that operate with segment routing.

4.5 MIDPOINTS AND LINK WEIGHTS

In [17], Bhatia et al. observed that the optimal value of lb_{path} on the 2-SRTEP was "almost as good as" the one of the MCFP and thus suggested that one single midpoint is enough to reach good traffic engineering solutions. This statement is somehow refuted by the results presented in the left part of Figure 4.3 which show that the optimal value of the 2-SRTEP can be much higher than the one of lb_{MCFP} for a significant number of instances. Both parts of Figure 4.3 actually suggest that "2-SRTEP is almost as good as k-SRTEP" would be a better statement. First, the similarities between both parts of Figure 4.3 hint that the number of midpoints only plays a minor role on the value returned by lb_{SRTEP}. Second, the high network utilization of the last percentiles seem to indicate that the efficiency of SR strongly depends on the underlying forwarding graphs. This is precisely what happens on the *NTT* instance (see Figure 4.4) on which lb_{SRTEP} returns a lower bound that is 606.84% higher than the actual network capacity. This impressive difference is

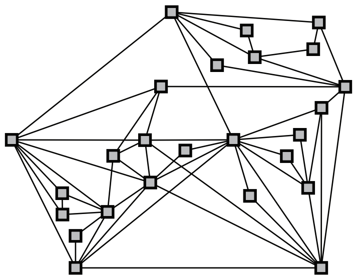


Figure 4.4: The NTT topology — node coordinates have been handpicked to make the topology "as planar as possible".

the result of a bad set of link weights that does not exploit the connectivity of the topology.[8]

Still, we only have compared lower bounds until now and we know from experience that actual optimal solutions can be even farther than the value of the MCFP. Luckily, the *path* and *segment* models provide us with the necessary tools to compute the optimal solutions of the k-SRTEP for any value of $k \in [2, |\mathbf{N}| - 1]$. In particular, we have computed the optimal solution of where k is equal to 2, 3, and $|\mathbf{N}| - 1$. Also, we broke down our our analysis on different set of link weights, i.e. unary, inverse, and random, to observe the actual impact of midpoints and shortest paths on the quality of the returned solutions.

As before, we performed our analysis on a subset of instances that have less than 40 nodes and no bottleneck. Our results are presented as set of box plots in Figure 4.5. They not only show that 2-SRTEP solutions can fall far from the value of *MCFP* but also that k-SRTEP with $k \geq 3$ often significantly outperforms 2-SRTEP no matter the set of link weights. Speaking of link weights, our results highlight that they are of particular importance and can drastically hamper the ability of segment routing. The best tested combination is unary link weights with no restriction on the number of midpoints. Though, 3-SRTEP seems to already provide a lot of improvement compared to 2-SRTEP — especially if the link weights are bad.

8 The instance was solved with inverse-capacity link weights.

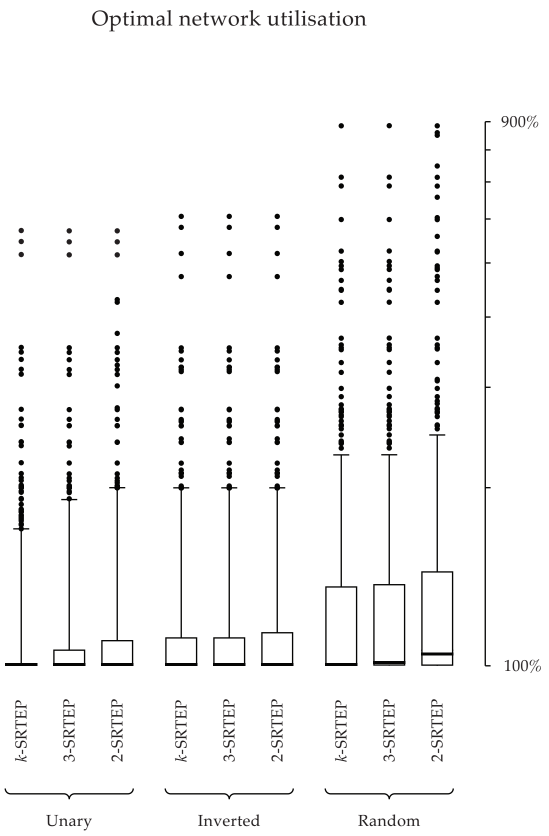


Figure 4.5: Distributions of optimal network utilizations for the 2-SRTEP, the 3-SRTEP, and the k-SRTEP (with $k = |\mathbf{N}| - 1$) on unary, inverse, and random link weights. Unary link weights with no restriction on the number of midpoints achieves the best result. The box contains all data points between the first and third quartiles (the thick line being the median) while data points explicitly represented correspond to outliers that exceed the 95th percentile (end of the whisker).

4.6 ADDITIONAL CONSTRAINTS

The constraints presented in Section 3.2 can be divided into two categories: *per demand* and *global*. A *per demand* constraint restricts the set of SR paths a demand can be forwarded on, no matter the value assigned to the other variables of the problem — e.g. the load of an edge or the SR path of another demand. The `maxCost` and `serviceChaining` constraints are examples of per demand constraints. On the contrary, a *global* constraint considers several variables of the problem such as the SR path of several demands (e.g. the `disjoint` constraint).

One of the nicest property of the *path* model is that we can extend it with any per demand constraint without having to change the model at all. Indeed, the set of all the possible SR paths of each demand being precomputed, we only have to filter out forbidden SR paths beforehand to ensure that any solution respects all the per demand constraints. This not only allows us to encode per demand constraints with no additional inequalities or variables, it also reduces the size of the problem by removing variables associated to forbidden SR paths.

Encoding global constraints usually requires additional inequalities and/or variables. For instance, a naive way to encode the `disjoint` constraint is to add an inequality to forbid all the pairs of incompatible SR paths of two demands:

$$\mathbf{x}_{p_1}^{d_1} + \mathbf{x}_{p_2}^{d_2} \leq 1 \qquad \forall p_1 \in P_{d_1}\ \forall p_2 \in P_{d_2}, \neg\texttt{disjoint}(p_1, p_2) \quad (14)$$

Unfortunately, this encoding suffers from the same weakness than the *path* model because the number of additional inequalities might grow exponentially with the number of possible SR paths.

An alternative, more scalable, way to encode the `disjoint` constraint consists in focusing on the edges instead of the SR paths. Let us introduce a new function $\texttt{visit}_p : \mathbf{E} \to \{0, 1\}$ that returns 1 if and only if SR path p visits a given link (u, v):

$$\texttt{visit}_p(u, v) = \begin{cases} 1 & \texttt{flow}_p(u, v) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Using this new function, we adapt (8) to guarantee that each edge is only visited by a single disjoint demand:

$$\sum_{p \in P_{d_1}} \mathbf{x}_p^{d_1} \texttt{visit}_p(u, v) + \sum_{p \in P_{d_2}} \mathbf{x}_p^{d_2} \texttt{visit}_p(u, v) \leq 1 \qquad \forall (u, v) \in \mathbf{E} \quad (15)$$

The gain of scalability is twofold. First, we only need $\mathcal{O}(|\mathbf{E}|)$ additional inequalities to encode the constraint on a pair of demands (d_1, d_2). Second, this encoding can actually be used to impose the disjoint constraint on a set of disjoint demands D_{dis} instead of a pair:

$$\sum_{d \in D_{dis}} \sum_{p \in P_d} \mathbf{x}_p^d \mathtt{visit}_p(u,v) \leq 1 \qquad \forall (u,v) \in \mathbf{E} \tag{16}$$

In the context of the *segment* model, the disjoint constraint is encoded in a similar way as in (16) but with a visit function $\mathtt{visit}_{s,t} : \mathbf{E} \to \{0,1\}$ for each forwarding graph $FG_{s,t}$ such that

$$\mathtt{visit}_{s,t}(u,v) = \begin{cases} 1 & FG_{s,t}(u,v) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Adding per demand constraints to the *segment* model requires a little more of elbow grease because they must be explicitly encoded into the model. Nevertheless, this is not a big problem for the `maxCost` constraint which is basically a weighted version of constraint (12). Encoding the `serviceChaining` constraint with a single service S (or set of services with no specific order between them) can also be done quite easily as follows:

$$\sum_{s \in S} \sum_{(u,s) \in \mathbf{E}} \mathbf{x}_{u,s}^d \geq 1 \tag{17}$$

$$\sum_{s \in S} \sum_{(u,s) \in \mathbf{E}} \mathbf{x}_{s,v}^d \geq 1 \tag{18}$$

where d is the demand on which the constraint is enforced.

Encoding the general form of the constraint, though, is much more challenging since it requires to extend the model with both additional variables and inequalities. Therefore, we recommend to rely on the path model when dealing with large service chaining constraint — especially if the services only have a small number of nodes.

4.7 HYBRID MODEL

The path and segment models have different pros and cons. While the path model naturally supports — and might actually benefit from — per demand constraints, only the segment model is able to handle SR paths with large number of midpoints within practical space requirements. Sometimes, the path model might be very well suited for a subset of demands (e.g. demands with strong per demand constraints) while the segment model is better suited for another. When operators have to deal with both sets of demands at the same time, they can rely on a hybrid model that combines both previous models into a single one. Let $\mathbf{D}_{path}$ and $\mathbf{D}_{seg}$ be the set of demands to be encoded using the path and segment models respectively such that $\mathbf{D}_{path} \cup \mathbf{D}_{seg} = \mathbf{D}$ and $\mathbf{D}_{path} \cap \mathbf{D}_{seg} = \emptyset$. The hybrid model is encoded as follows:

$$\sum_{p \in P_d} \mathbf{x}_p^d = 1 \qquad \forall d \in \mathbf{D}_{path} \quad (19)$$

$$\sum_{u \in \mathbf{N}} \mathbf{x}_{s,u}^d = \sum_{u \in \mathbf{N}} \mathbf{x}_{u,t}^d = 1 \qquad \forall d = (s,t) \in \mathbf{D}_{seg} \quad (20)$$

$$\sum_{v \in \mathbf{N} \setminus \{u\}} \mathbf{x}_{v,u}^d = \sum_{v \in \mathbf{N} \setminus \{u\}} \mathbf{x}_{u,v}^d \qquad \forall d = (s,t) \in \mathbf{D}_{seg}, \forall u \in \mathbf{N} \setminus \{s,t\} \quad (21)$$

$$\mathbf{load}_{u,v}^{path} = \sum_{d \in \mathbf{D}_{path}} \sum_{p \in P_d} \mathbf{x}_p^d \, \texttt{flow}_p(u,v) \qquad \forall (u,v) \in \mathbf{E} \quad (22)$$

$$\mathbf{load}_{u,v}^{seg} = \sum_{d \in \mathbf{D}_{seg}} \sum_{s,t \in \mathbf{N}, s \neq t} \mathbf{x}_{s,t}^d \, FG_{s,t}(u,v) \, \texttt{bw}_d \qquad \forall (u,v) \in \mathbf{E} \quad (23)$$

$$\mathbf{load}_{u,v}^{path} + \mathbf{load}_{u,v}^{seg} \leq \phi c_{u,v} \qquad \forall (u,v) \in \mathbf{E} \quad (24)$$

Constraints (19), (20), and (21) respectively encode SR paths accordingly the path and segment models. Inequalities (22), (23) and (24) compute the load of each link as the sum of demands in $\mathbf{D}_{path}$ and $\mathbf{D}_{seg}$ that traverse that link. Particularly, (24) makes sure that no link utilization exceeds the network utilization ϕ.

4.8 CONCLUSION

In this chapter, we have analysed the ability of SR to solve traffic engineering problems. For that purpose, we introduced the segment model to solve the

SRTEP optimally without limiting ourselves to an arbitrary maximum number of midpoints. Particularly, the segment model allowed us to show that the assumption, popularized by [17], that "one midpoint is enough" is misleading. Indeed, our observation showed that allowing SR paths with two midpoints often substantially improve over being restricted to a single one.

Curiously, the impact of the shortest paths on the SR paths has only been barely discussed in the literature — at least to the best of our knowledge. We showed that bad shortest paths might hamper the ability of SR path to prevent congestion and highlighted that allowing more midpoints usually does not make up for those bad shortest paths. Of course, finding an optimal set of link weights that minimize the network utilization is an NP-hard problem [41] — and a difficult one in practice. Looking for optimal SR paths and link weights at the same time thus raises many challenges. We however think that the benefits of being able to solve such a global problem are limited (we briefly discussed the cost and risks of changing shortest paths in Section 2.2). Indeed, we believe that a reasonable operational strategy consists in finding a set of link weights such that the network is able to accommodate a large variety of situations without relying on SR path (see the oblivious shortest path routing approaches proposed by Altın et al. [5, 6]). SR paths are then used as a last resort to deal with a critical situation or to enforce particular service level agreements such as minimum delays and disjoint paths.

While MILP is an extremely powerful tool for traffic engineering, it suffers from speed and scalability weaknesses that are inherent to solving NP-Hard problems optimally. Those weaknesses are the main reason why we have limited or analysis to instances with less than 40 nodes, i.e., instances that could be solved by Gurobi 7.0 [52] in reasonable time. Despite being primordial in analysing SR as a technology, optimality is a much less desired property in practice. Indeed, network operators are mostly interested by approaches that are able to provide near optimal solution consistently and as fast as possible. The next chapters of this thesis are dedicated to such techniques that trade the ability to prove optimality for speed and scalability.

5 FAST RECOVERY WITH LOCAL SEARCH

For networks, the capability to promptly answer to unexpected dynamics, such as link failures and traffic changes, in a very short time is more critical than ever. This is especially true for large networks such as inter-datacenter wide area networks and the ones of ISPs. On the first hand, new network architectures powered by SDN enable network to almost fully use their link capacity and let them carry more traffic without having to heavily over-provision their infrastructures (see Chapter 4 and [60, 62]). On the second hand, sudden traffic surges due to newly popular content, flash crowds [112] or even DoS attacks[1] tend to create more frequent unexpected perturbations. Those events can significantly change traffic distribution, and sum to the previously-considered factors (like link failures) that can affect traffic optimization. Those two factors increase the likelihood of experiencing congestion at any moment and led recent works to focus on online traffic-engineering solutions [35, 92, 53].

In Chapter 4, we have seen that Mixed Integer Linear Programming (MILP) is a very powerful tool to solve TE problems — the SRTEP not being an exception. However, MILP approaches usually struggle to provide network operators with solution in short delays. This scalability issue are particularly stressed on large network — as the ones of ISPs. Nevertheless, pragmatic workarounds have been developed to avoid, or at least diminish, the poor scalability of MILP solvers.

periodic re-optimization. This workaround simply consists in re-optimizing the network periodically — typically every few (e.g. 5) minutes — instead of computing a new configuration each time a significant change occurs in the network [60, 62]. The obvious interest of this approach is that it can handle several changes by solving a single problem using more time.

1 DoS (Denial-of-Service) attacks usually consist in flooding network devices with superfluous requests in order to make the network unavailable to its users.

However, by running periodically, it can only remove congestion after the fixed period. The worst case being the one in which a critical event, such as a link failure, occurs just after the previous re-optimization and thus leaves the network in a congested state until the next optimization. The second issue with this approach is that it suffers from its own scalability issue. Indeed, the bigger the network, the longer the MILP solver will have to run to reconfigure the SR paths. Another difficulty comes from the fact that the successive configuration must stay relatively homogenous to minimize the cost of reconfiguring several network devices — which can lead to additional instability on the network.

PRECOMPUTED EVENTS. An alternative approach — that is complementary to the previous one — is to precompute a congestion-free configuration for each of the most expected, or the most critical, events [17, 35, 53, 92]. When one of those event occurs, the controller just has to push the corresponding precomputed configuration on the network devices. Events that are likely to be considered typically include link and node failures but can also include common traffic patterns. For instance, the daily and nightly traffic distributions of an ISPs might be very different from each other but stable on a long period. Of course, precomputing a network configuration for each combination of possible events is not scalable both in time and memory. Network operators thus have to focus on a set of expected events thus leaving the network vulnerable to the unexpected ones by definition.

In this chapter, we explore the feasibility of a different approach where SR paths are re-optimized as soon as the utilization of a link increases too much, e.g. in consequence of a significant traffic change or a link or node failures. In particular, we focus on providing operators with high-quality re-optimized network configurations in very short delays. To tackle this challenge, we rely on an optimization techniques called *local search*. To the contrary of systematic approaches like MILP, Local Search (LS) does not strive to prove the optimality of its solutions but rather tries to find high-quality solutions as fast as possible. Because of that, LS tends to behave very well on large scale problems. Also, LS is inherently an anytime optimization techniques that will return a solution no matter when the algorithms is stopped. Therefore, LS is particularly well suited for time-constrained scenarios like the one described above.

5.1 BACKGROUND ON LOCAL SEARCH

This section briefly formalizes and presents the concepts and notations used throughout this chapter.[2] To do so, we assume that the combinatorial optimization problem $\mathcal{P}$ to be solved has the form

$$\begin{array}{ll} \texttt{minimize} & f(\mathbf{x}) \\ \texttt{subject to} & c_1(\mathbf{x}) \\ & \vdots \\ & c_m(\mathbf{x}) \end{array}$$

where $\mathbf{x} \in \mathbb{Z}^n$ is a vector of n discrete decision variables, f is an objective function $\mathbb{Z}^n \to \mathbb{R}$ that evaluates the quality of a variable assignment, and $c_1, \ldots, c_m$ are constraints defining the search space.

Definition 8. A *solution* of $\mathcal{P}$ is an assignment of values to the variables in $\mathbf{x}$ that satisfies the constraint $c(\mathbf{x})_1 \wedge \cdots \wedge c(\mathbf{x})_m$.

Definition 9. The *search space* of $\mathcal{P}$ is the set of all its solutions and is denoted by $\mathcal{S}_\mathcal{P}$.

Definition 10. The set of *optimal solutions* of $\mathcal{P}$ is denoted by $\mathcal{S}^*_\mathcal{P}$ and defined as

$$\mathcal{S}^*_\mathcal{P} = \{s \in \mathcal{S}_\mathcal{P} \mid f(s) = \min_{s' \in \mathcal{S}_\mathcal{P}} f(s')\}.$$

Typically, an LS algorithm starts from a solution $s \in \mathcal{S}_\mathcal{P}$ and moves from solutions to solutions in order to improve the value of its objective function f. At each iteration, the LS algorithm evaluates the neighboring solutions $N(s)$ of solution s and decides whether to move to the most promising one, or to stay at solution s. We now define the concepts of neighborhood, transition graph, and local optimality which are central to this iterative process.

Definition 11. A *neighborhood* is a function $N : \mathcal{S}_\mathcal{P} \to P(\mathcal{S}_\mathcal{P})$ that defines the set of adjacent solutions $N(s) \subseteq \mathcal{S}_\mathcal{P}$ of each solution s.

Definition 12. The *transition graph* $G(\mathcal{S}_\mathcal{P}, N)$, associated to a search space $\mathcal{S}_\mathcal{P}$ and a neighborhood N, is the graph whose nodes are solutions in $\mathcal{S}_\mathcal{P}$ and where an arc $s \to s'$ exists if $s' \in N(s)$.

2 Note that the definitions presented in this section are directly adapted from [59].

Definition 13. A neighborhood $N : \mathcal{S}_{\mathcal{P}} \to P(\mathcal{S}_{\mathcal{P}})$ is connected if and only if, for each pair of solutions s_1, s_2, there exists a path from s_1 to s_2 in $G(\mathcal{S}_{\mathcal{P}}, N)$.

Definition 14. A solution $s \in \mathcal{S}_{\mathcal{P}}$ is locally optimal with respect to its neighborhood N if

$$f(s) \leq \min_{i \in N(s)} f(i).$$

Definition 15. The set of locally optimal solutions is denoted by $\mathcal{S}^{+}_{\mathcal{P}}$ and contains all the optimal solutions of $\mathcal{P}$ so that $\mathcal{S}^{+}_{\mathcal{P}} \subseteq \mathcal{S}^{*}_{\mathcal{P}}$.

Neighborhoods play a critical part in the effectiveness of local search algorithms. Large neighborhoods generally take more time to explore but simultaneously reduce the number of local optima as well as the diameter of the transition graph, which, in turn, increase the likelihood of finding high quality solution quickly. Finding the right trade-off between the diameter of the transition graph and the time spent exploring neighborhoods is particularly import design choice that is often problem dependent — if not instance dependent.

Connectivity is another important property of neighborhoods. Basically, a connected neighborhood guarantees that there exists a path between any solution $s \in \mathcal{S}_{\mathcal{P}}$ and any solution $s* \in \mathcal{S}^{*}_{\mathcal{P}}$. In other words, a connected neighborhood guarantees that a well-designed local search algorithm can reach an optimal solution. However, the heuristic used to evaluate the solutions in the neighborhood might prevent the local search algorithm from doing so.

Definition 16. A *local search* algorithm for $\mathcal{P}$ computes a path

$$s_0 \to s_1 \to \cdots \to s_k$$

in the transition graph $G(\mathcal{S}_{\mathcal{P}}, N)$ such that

$$s_{i+1} \in N(s_i) \quad (1 \leq i \leq k).$$

One of the simplest local search algorithm one can imagine is the following

```
function LocalSearch($N, s_{init}$):
   $s \leftarrow s_{init}$
   while $\neg$StopCondition():
      for $s' \in N(s)$:
         if $c_1(s') \wedge \cdots \wedge c_n(s')$ and $f(s') \leq f(s)$:
            $s \leftarrow s'$
   return $s$
```

It starts from s_{init} and, at each iteration, selects the best solution in the neighborhood of the current one. The algorithm stops when a stop condition, like a time limit, is met. The final solution returned by such a local search algorithm usually belongs to $\mathcal{S}^+_\mathcal{P}$ (for a given neighborhood N).

While heuristics aim at reaching good local optima quickly by choosing the next solution in a neighborhood, metaheuristics, on the contrary, aim at escaping those local optima and to direct the local search in the most promising parts of the search space. We briefly describe the ones that we consider to be the most famous.

TABU SEARCH. Tabu search [48] is a popular and effective metaheuristic that is the parent of a large set of metaheuristics and techniques. Tabu search aims at preventing an embedded local search heuristic from returning to recently visited areas of the search space. To achieve this, tabu search maintains a short term memory of the specific changes between the recently explored solutions to forbid the undoing of those changes in the near future. Tabu search has been successfully used in traffic engineering to optimize link weights [40].

SIMULATED ANNEALING. Simulated annealing [67] is based on the Metropolis heuristic [75] which accepts a degrading move with probability

$$exp(\frac{f(s) - f(s')}{t})$$

where t is a parameter of the heuristic called *temperature*. The key idea underlying simulated annealing is to iterate the Metropolis heuristic with a sequence of decreasing temperatures. The effect is that simulated annealing accepts many moves at the beginning of the search, thus sampling the search space widely, and moves progressively toward small values of t thus converging toward random improvements and hopefully a high quality local optima.

GUIDED SEARCH. Guided local search [111] is based on the recognition that a local optima $s \in \mathcal{S}^+_\mathcal{P}$ for objective function f may not be locally optimal with respect to another function f'. Therefore, using f' instead of f could drive the search away from s. As a consequence, the key idea behind guided local search is to use a sequence of objective functions $f_0, \ldots, f_i$ to escape local optima and to explore the search space widely.

5.2 A NEIGHBORHOOD FOR SEGMENT ROUTING

The SRTEP can easily be instantiated as a local search problem $\mathcal{P}$ in which each demand is associated with a decision variable that represents the SR path on which that demand is forwarded. We subsequently refer to those variables as *path variables*. The objective function to be minimized computes the network utilization of a complete assignment of the path variables

$$f(\mathbf{x}) = \max_{(u,v)\in\mathbf{E}} \frac{\sum_{i\in\mathbf{D}} \texttt{flow}_{\mathbf{x}_i}(u,v)}{c_{u,v}}.$$

In this chapter, we assume that there is no additional constraint to impose on the path variables. Therefore, any assignment is a valid solution. We have seen in Chapter 4 that the number of SR paths grows exponentially with the number of midpoints. The size of the search space $\mathcal{S}_\mathcal{P}$ thus grows exponentially with both the maximum number of midpoints k and the number of demands in $\mathbf{D}$ such that

$$\mathcal{O}(|\mathcal{S}_\mathcal{P}|) = \mathcal{O}(|\mathbf{N}|^{k|\mathbf{D}|}).$$

We propose to build a simple neighborhood function N_{path} so that the neighborhood of a solution s is made of every solution s' that is equal to s except for one SR path

$$N_{path}(s) = \{s' \in \mathcal{S}_\mathcal{P} \mid \exists! i \in \mathbf{D} : s'_i \neq s_i\}$$

where $\exists!$ denotes the uniqueness quantification. Without any surprise, the size of $N_{path}(s)$ grows exponentially with the number of midpoints k and is

$$\mathcal{O}(|\mathbf{D}||\mathbf{N}|^{k-1})$$

which could quickly become prohibitive when solving the k-SRTEP with large value of k. The transition graph induced by N is strongly connected and has a diameter of length $|\mathbf{D}|$. Theoretically, this guarantees that a local search algorithm could reach an optimal solution, from any other one, in at most $|\mathbf{D}|$ iterations.

We adapted the basic local search algorithm presented in the previous section to use the neighborhood function N_{path}. The algorithm, that we subsequently refer to as *basic local search*, is the following

```
function BasicLocalSearch($s_{init}$):
    $s \leftarrow s_{init}$
```

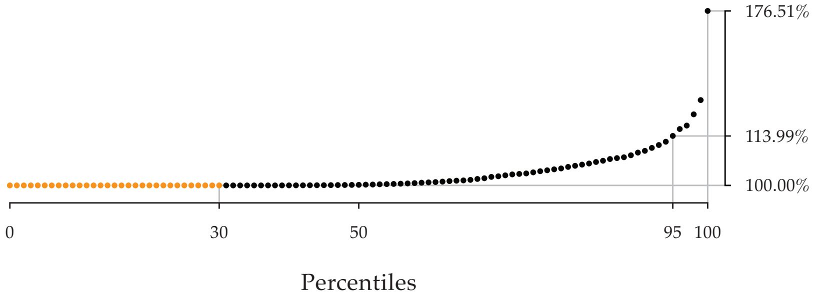


Figure 5.1: Comparison of the basic local search with the optimal instance of 2-SRTEP. The algorithm reaches optimality on 30% of the instances and is 76.51% higher than the optimal solution in the worst case.

```
3   while $\{s' \in N_{path}(s) \mid f(s') \leq f(s)\} \neq \emptyset$:
4       $s \leftarrow$ SelectRandom($\{s' \in N_{path}(s) \mid f(s') \leq f(s)\}$)
5   return $s$
```

It starts from solution s_{init} which we assume to be the shortest paths routing, i.e. the solution in which every SR path directly follows the shortest paths towards its destination without making any detour. At each iteration, the algorithm explores the entire neighborhood $N_{path}(s)$ and moves to the solution that leads to the largest improvement of the objective function f. If there are several such solutions, one of them is selected randomly. The algorithm stops when $N_{path}(s)$ contains no solution with a better objective value than s — s is a local optima.

We measured the efficiency of this algorithm by solving the 2-SRTEP on all the instances solved optimally in Chapter 4. The results are presented in Figure 5.1 where the vertical axis is the ratio of the best objective value found divided by the optimal objective value. Quite surprisingly, this simple algorithm already performs well: it reaches optimality on 30% of the considered instances, returns solutions that are not more than 14% higher than the optimal one for 95% of them, and is only 76.51% larger than the optimal objective value in the most extreme case.

Improving the efficiency of the basic local search algorithm goes hand-in-hand with escaping local optima. A first natural way to achieve this is to use larger, or at least partially disjoint, neighborhoods. However, N_{path} is already very likely to face scalability issues due to its exponential asymptotic growth.

A second solution to escape local optima is to rely on metaheuristics — which happens to be the topic of the next section.

5.3 LINK GUIDED LOCAL SEARCH

Understanding the properties of local optima is a solid basis to design a procedure to escape them. By definition, a solution s is a local optima with respect to f, if no solution $s' \in N_{path}(s)$ has a smaller objective value than s. In particular, a solution s is a local optima if it is impossible to reduce the network utilization by changing the SR path of a single demand in s. Ultimately, minimizing the network utilization comes down to minimizing the utilization of the most loaded links. Let $f_{u,v} : \mathcal{S}_{\mathcal{P}} \to \mathbb{R}$ be the function that returns the utilization of link (u,v) given an assignment of the path variables

$$f_{u,v}(\mathbf{x}) = \frac{\sum_{i \in \mathbf{D}} \texttt{flow}_{\mathbf{x}_i}(u,v)}{c_{u,v}} \qquad (\forall (u,v) \in \mathbf{E})$$

so that $M(\mathbf{x})$ denotes the set of all the most utilized links

$$M(\mathbf{x}) = \{(u,v) \in \mathbf{E} \mid f_{u,v}(\mathbf{x}) = f(\mathbf{x})\}.$$

Therefore, to improve over s, a solution $s' \in N_{path}(s)$ needs to reduce the utilization of all links in $M(s)$,

$$\forall (u,v) \in M(s) : f_{u,v}(s') < f_{u,v}(s), \tag{25}$$

while ensuring that the utilization of all links in $\mathbf{E} \setminus M(s)$ remain strictly lower than $f(s)$. To respect condition (25), solution s needs to contain an SR path that forwards its demand on all the links in $M(s)$. However, the probability of having such an SR path grows finer when $M(s)$ grows larger. Especially, it is very unlikely that such an SR path exists if $M(s)$ is made of two symmetrical links

$$M(s) = \{(u,v),(v,u)\}$$

because the SR path would have to contain a cycle.

Similarly to guided local search (see Section 5.1), we tackle this issue by dynamically changing the objective function to escape local optima. In particular, each iteration of our local search algorithm focuses on minimizing the utilization of a single link instead of focusing on the overall network utilization. The general pseudo code of this local search algorithm, that we refer to as *link guided local search*, is the following:

```
1  function LinkGuided($N, s_{init}$):
2    $s \leftarrow s_{init}$
3    while $\neg$StopCondition():
4      $(u,v) \leftarrow$ SelectLink($s$)
5      for $s' \in N(s)$:
6        if $f_{u,v}(s') < f_{u,v}(s)$ and $f(s') \leq f(s)$:
7          $s \leftarrow s'$
8    return $s$
```

Note that the second condition at line 6 ensures that the algorithm never degrades the quality of the solution.

Selecting the next link to drive the local search (line 4) is the most critical design choice of this algorithm. There are two opposite natural heuristics to do that. The first is to select the next link among the most utilized:

```
function SelectLinkMax($s$):
  return $(u,v) \in M(s)$ with probability $1/|M(s)|$
```

The second is to select the next link randomly among **E** in order to drive the search in various parts of the search space:

```
function SelectLinkUniform($s$):
  return $(u,v) \in E$ with probability $1/|E|$
```

We propose to rely on a third heuristic which stands halfway between SelectLinkMax and SelectLinkUniform. The key idea is to select the next link randomly but where the probability of selecting a link increases with the link utilization. Formally, the probability $P_{u,v}(\mathbf{x})$ of selecting link (u,v) is determined by its utilization and by an *intensification coefficient* denoted by α:

$$P_{u,v}(\mathbf{x}) = \frac{f_{u,v}(\mathbf{x})^{\alpha}}{\sum_{(u,v)\in\mathbf{E}} f_{u,v}(\mathbf{x})^{\alpha}}.$$

The pseudocode of our heuristic is thus the following:

```
function SelectLinkWeighted($s$):
  return $(u,v) \in \mathbf{E}$ with probability $P_{u,v}(s)$
```

High values of α increase the chance of selecting the most utilized edges. The larger α is, the more likely SelectLinkWeighted is to return the same solution has SelectLinkMax. To the contrary, low values of α flatten the probability distribution. Particularly, setting α to 0 results in the same behavior as SelectLinkUniform.

Figure 5.2 shows the impact of α on the efficiency of link guided local search. As before, the algorithm was evaluated on all the optimally solved instances from Chapter 4. For each instance, link guided local search was given a limit of 10,000 iterations. The right most box plot presents the same results as in Figure 5.1 to ease the comparison with the basic local search. We see that setting α to 0, which corresponds to SelectLinkUniform, leads to the worst results. The problem with this configuration is that it focuses too much on exploring the search space. Therefore, it is not able to take advantage of the limited iterations to drive the search toward a high quality local optima. Increasing the value of α substantially improves the effectiveness of link guided local search. However, and this is quite interesting, too high values of α, making the selection behave like SelectLinkMax, reduce the effectiveness of the algorithm. The best results were obtained with $\alpha = 8$.

5.4 SCALABLE NEIGHBORHOOD

As the *path model* before it (see Chapter 4), neighborhood N_{path} grows exponentially with the maximum number of midpoints

$$\mathcal{O}(|N_{path}(s)|) = \mathcal{O}(|\mathbf{D}||\mathbf{N}|^{k-1}) \qquad (\forall s \in \mathcal{S}_{\mathcal{P}}).$$

Exploring this neighborhood thus becomes a problem when $k \geq 3$. Fortunately, there are many solutions to overcome this issue. One could, for instance, design a selection heuristic that samples the neighborhood instead of systematically exploring it (see random walks [59] and the Metropolis heuristic [75]). Another approach, the one we chose to follow, is to reduce the size of the neighborhood or to explore it partially. In this section, we present a second neighborhood that is strongly connected, and scales well with k.

The *Levenshtein distance* [72], commonly referred to as an *edit distance*, is a way of counting the dissimilarities between two sequences of *symbols*, e.g. two strings. Basically, the edit distance between two sequences is the minimum number of primitive operations that must be applied to transform one sequence into the other. Those operations usually are:

- *Insert* a single symbol;
- *Remove* a single symbol;
- *Substite* a single symbol by another one.

Since an SR path can be represented as a sequence of midpoints, we can apply the edit distance to count the dissimilarities between two SR paths (see Example 7).

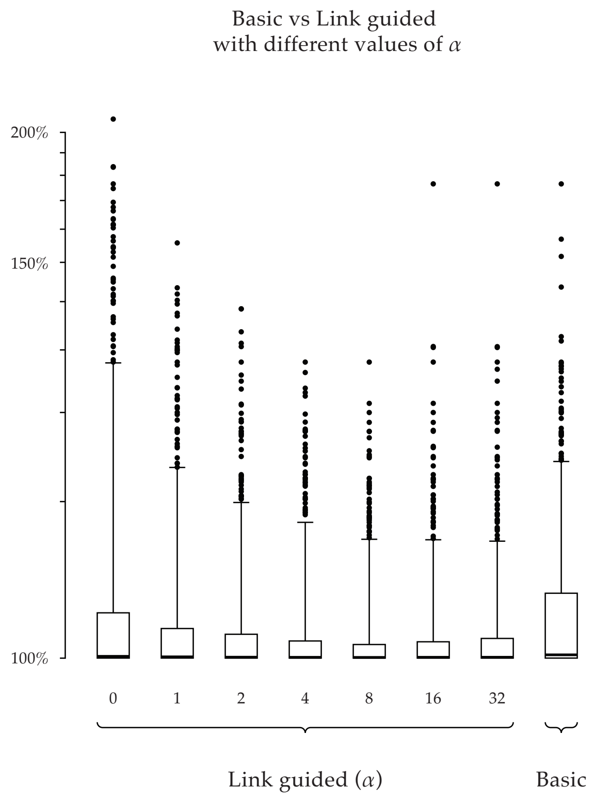


Figure 5.2: Impact of α on the effectiveness of link guided local search. The best configuration is $\alpha = 8$.

Example 7. The edit distance between SR paths $p_1 = s, a, b, c, t$ and $p_2 = s, a, d, t$ is 2:

1. $s, a, b, \mathbf{c}, t \rightarrow s, a, b, t$ (deletion of c),
2. $s, a, \mathbf{b}, t \rightarrow s, a, \mathbf{d}, t$ (substitution of b by d).

An alternative is to delete b and then substitute c by d.

Let *edit* be a function $\mathbf{N}^* \times \mathbf{N}^* \rightarrow \mathbb{N}$ that returns the edit distance between two SR paths using the three previous operations plus an additional one:

- *Insert (I)* a new midpoint;
- *Remove (R)* a midpoint;
- *Substite (S)* a midpoint by another one;
- *Clear (C)* an SR path by removing all its midpoints.

We propose a new neighborhood function N_{edit} so that the neighborhood of a solution s is made of every solution s' that is equal to s expect for one SR path which differ by a single edit operation

$$N_{edit}(s) = \{s' \in \mathcal{S_P} \mid \exists! i \in \mathbf{D} : s'_i \neq s_i \ \wedge \ edit(s'_i, s_i) = 1\}.$$

To the contrary of N_{path}, the size of N_{edit} grows linearly with the maximum number of midpoints k and is

$$\mathcal{O}(|N_{edit}(s)|) = \mathcal{O}(k\,|\mathbf{N}||\mathbf{D}|) \qquad (\forall s \in \mathcal{S_P}).$$

Furthermore, the transition graph produced by N_{edit} remains strongly connected and has a diameter of length $k\,|\mathbf{D}|$. Indeed, it is possible to transform any SR path into another one by applying at most k edit operations. Note that insertion and removal are the only required operations to ensure that the transition graph of N_{edit} is strongly connected.[3] Nevertheless, the substitute and clear operations increase the connectivity of the transition graph, and thus potentially reduce the number of local optima, without changing the asymptotical growth of $|N_{edit}|$.

Property 1. Neighborhoods N_{path} and N_{edit} are equal when $k = 2$.

3 The edit distance using only insertion and removal is called *Longest Common Subsequence* (LCS) distance. It is an upper bound of the Levenshtein distance.

Proof. Trivially, two SR paths made of at most one midpoint cannot differ by more than a single midpoint. □

We compared the efficiency of N_{edit}, with different set of edit-operations, to N_{path} by solving the 3-SRTEP on our usual dataset. The results are presented in Figure 5.3. As before, each algorithm was configured with a limit of 10,000 iterations to solve each instance. The first — striking — observation is that N_{edit} is not only competitive with N_{path} but actually computes better solutions. We explain those counter intuitive results by the fact that N_{edit} might perform less attractive moves than N_{path} (in terms of improvement of the objective function) but that lead to local optima that are easier to escape. Indeed, N_{edit}, with our four edit operations (i.e. IRSC), performs much better than the other tested neighborhoods and computes solutions that are only 12% higher than their optimal objective value in the worst case. Finally, as a general observation, we see that Link Guided local search finds better solutions when solving the 3-SRTEP than 2-SRTEP, no matter if it is used with N_{edit} or N_{path}.

5.5 INTENSIFICATION VS DIVERSIFICATION

Even a small neighborhood like N_{edit} might be to large to explore entirely in critical situations in which promptly finding a good-enough solution is of paramount importance. In those contexts, it is often interesting to (temporarily) forget about exploring *diversified* parts of the search space to *intensify* the search in a potentially sub-optimal but promising direction.

We showed in the beginning of Section 5.3 that the exploration of the neighborhood could be sped up by focusing on the SR paths visiting the selected link (because changing other SR path cannot lead to an improvement of the objective function). We could go a step further by reducing even more the set of considered SR paths. Quite intuitively, changing the SR path of the largest demand that visits the selected link is likely to result in the biggest improvement of that link utilization. Therefore, we propose to adapt the exploration of the neighborhood so that solutions that change the path of large demand are visited first.

Let $d_1, \dots, d_n$ be the n demands with the largest contribution to the load of link (u, v), sorted in non-increasing order

$$\texttt{flow}_{d_1}(u, v) \geq \cdots \geq \texttt{flow}_{d_n}(u, v).$$

Heuristic OrderedImprovement relies on those sorted demands to drive the exploration of neighborhood N

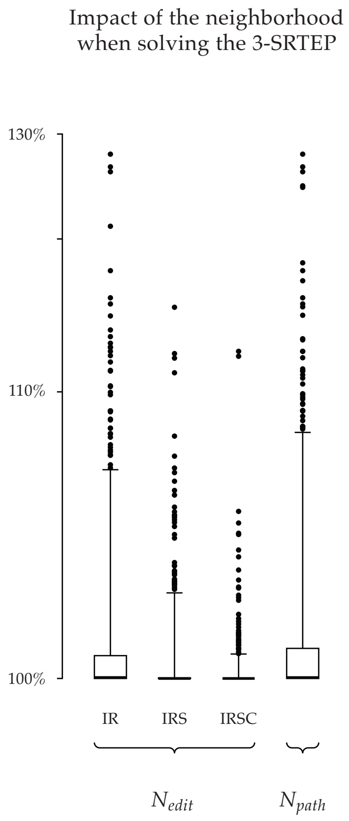


Figure 5.3: Comparison of N_{edit}, with different set of edit operations, to N_{path} on the 3-SRTEP. Letters I, R, S, and C describe the set of edit-operations used to define N_{edit}.

```
1  function OrderedImprovement($N, s, d_1, \dots, d_n$):
2    for $i \in 1, \dots, n$:
3      $s^+ \leftarrow s$
4      for $s' \in \{s' \in N(s) \mid s'_{d_i} \neq s_{d_i}\}$:
5        if $f_{u,v}(s') < f_{u,v}(s^+)$ and $f(s') \leq f(s)$:
6          $s^+ \leftarrow s'$
7      if $s^+ \neq s$:
8        return $s^+$
9    return $s$
```

At each iteration i, it explores the solutions in $N(s)$ that changes the path of demand d_i with regard to solution s (line 4). The heuristic returns the best improving solution, or iterates to the next demand d_{i+1} if no such solution exists for d_i (lines 5 to 8). The heuristic returns solution s if it wasn't able to find an improving solution.

We propose a new local search algorithm called Link Guided Variable Search (LGVS) that extends the previous link guided search to rely on OrderedImprovement. The pseudocode is the following

```
1  function LinkGuidedVariableSearch($N, maxSize, s_{init}$):
2    $s \leftarrow s_{init}$
3    $n \leftarrow 1$
4    while $\neg$StopCondition() and $n < maxSize$:
5      $(u, v) \leftarrow$ SelectLink($s$)
6      $d_1, \dots, d_n \leftarrow$ SortedDemands($u, v$)
7      $s' \leftarrow$ OrderedImprovement($N, s, d_1, \dots, d_n$)
8      if $s \neq s'$:
9        $s \leftarrow s'$
10       $n \leftarrow 1$
11     else:
12       $n \leftarrow n + 1$
13   return $s$
```

At each iteration, the algorithm selects one of the most loaded links (line 5), and selects and sorts the n demands with the largest contribution to the load of that particular link (line 6). OrderedImprovement is then used on the selected link and demands to drive the exploration of the neighborhood (line 7). If an improving solution was found, the number of demands to select at the next iteration of the algorithm is reset to 1; it is incremented otherwise. The algorithm stops when a stop condition is met, or when n exceeds the maximum number of demands to select.

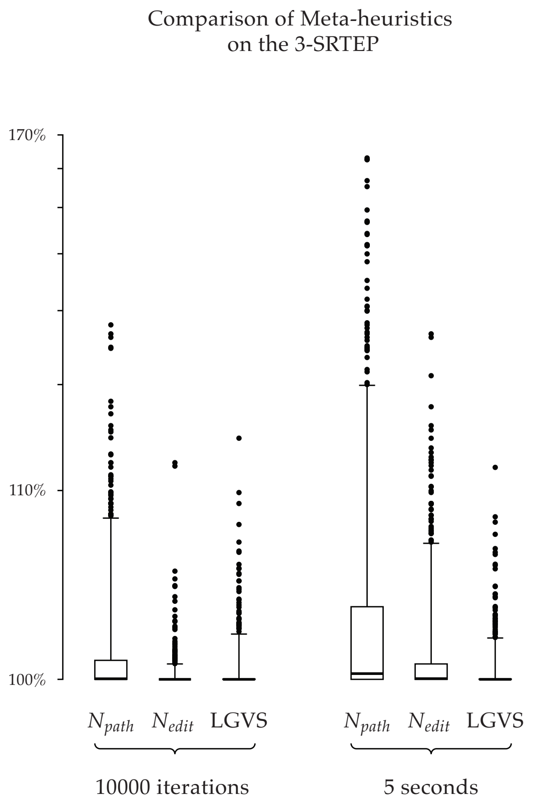


Figure 5.4: Comparison of LGVS to link guided local search with N_{edit} and N_{path}. LGVS achieves the best results in time constrained scenarios.

We compared the efficiency of LGVS on the 3-SRTEP against link guided search with N_{edit} and N_{path}. Precisely, the three local search algorithms were tested in two different scenario: (i) with a limit of 10,000 iterations, and (ii) with a pragmatic time limit of 5 seconds. The results are presented in Figure 5.4. On the left part, we see that fast link guided local search performs quite well, getting solution that are less than 14% higher than their optimal solution, but not as well as link guided search with N_{edit}. However, on the right part, we see that fast link guided local search is by far superior than the other approaches when the actual computing time is used as a stop condition.

5.6 CONCLUSION

In this chapter, we focused on the design of a local search algorithm to find high-quality solutions to the SRTEP in a scalable and robust way. To achieve this, we designed a scalable neighborhood, N_{edit} to explore the search space of the SRTEP. We also proposed Linked Guided Variable Neighborhood Search (LGVS), a local search algorithm that rely and focuses on minimizing the utilization of the most utilized edges, and adapts its exploration of N_{edit} to control the trade-off between sampling the search space and improvement. We showed that this algorithm performs well: it usually finds solutions with an objective value that is less than 1% higher than the optimal one for more than 75% of the considered instances, and remains at a reasonable distance (15%) of the optimal objective value in the worst cases.

Escaping local optima can be approached in a multitude of way. This is witnessed by the large variety of heuristics and metaheuristics developed during the past decades. It is therefore unlikely for N_{edit} and LGVS to be the best possible combination one can come up with. Nevertheless, we believe that the approaches we designed could help network operators and serve as a baseline for researchers in the field.

While all the local search algorithms we proposed could easily be extended to handle additional constraints, by filtering invalid solution when exploring the neighborhood, we still have to face the problem of finding an initial solution that actually respects those constraints in minimal time. The next chapter of this thesis focus on that particular problem.

6

CONSTRAINT PROGRAMMING AND LOCAL SEARCH

The link guided search algorithms presented in Chapter 5 proved to be very competitive. This is especially true when considering problems of large size where MILP usually struggles to find high-quality solutions in practicable time (and amount of memory). However, while link guided search can easily be extended to handle additional constraints, such as service chaining and disjoint SR paths, its effectiveness is substantially diminished in such constrained context. In particular, our LS algorithms always assume that their incubent solutions are feasible. This is not a problem in offline contexts, in which a low-quality but feasible solution can be computed using MILP, and then optimized with LS. However, events such as link failures, which might result in a substantial change of the forwarding graphs, are likely to turn feasible solutions into unfeasible ones. For instance, nothing guarantees that two disjoint SR paths would remain disjoint under different sets of forwarding graphs. Even worse, such disruptive events might actually make the problem infeasible — if two disjoint demands have to traverse a bridge for example. Of course, MILP techniques could still be used to find a new feasible solution or to detect infeasible constraints. However, as discussed in Chapter 4, MILP might takes a long time to compute such feasible solution and is therefore not suited for the time-critical and large scale scenarios for which we initially designed our LS algorithms.

6.1 CONSTRAINT PROGRAMMING

Among the long standing traditions of the Constraint Programming (CP) community, one is to illustrate CP's basic concepts with the 8-Queens problem [94]. The 8-Queens problem is a toy puzzle, proposed in 1848 by German chess player Max Bezzel [16], that has attracted (and still attracts [13, 46]) the attention of many researchers in artificial intelligence. The problem is the one

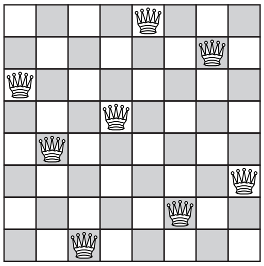


Figure 6.1: A possible solution to the 8-Queens problem.

of placing 8 queens on a 8×8 chess board so that no two queens attack each others. For the readers who are not familiar with chess rules, that means that no pair of two queens can share the same row, column, or diagonal. One such solution is depicted in Figure 6.1.

The 8-Queens problem is an example of a *combinatorial problem* as there is a finite, but astronomically large, number of possible solutions (there are 64!/56! = 178,462,987,637,760 ways to place the queens on the board) but only a few of them actually respect the rules of the game (there are 92 solutions to the 8-Queens problem). Solving such combinatorial problem is a complex task since there are too many configurations to investigate to find an actual solution (whether it is by cherry picking or by enumeration). Furthermore, there is no trivial way to infer a solution by looking at the rules of the game alone. As a matter of fact, combinatorial problems — at least the interesting ones — usually belong to the NP-Hard class.[1]

The CP way of solving a combinatorial problem is to translate it into a Constraint Satisfaction Problem (CSP). A CSP is usually defined by a triple $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ such that:

- **X** is a set of *variables* that represent the unknowns of the problem;
- **D** is a set of functions that links each variable to its *domain* which is a set of values the variable can be assigned to;
- **C** is a set of *constraints* which are relations to be respected by any solution of the problem.

1 The N-Queens problem, i.e. the generalization of the 8-Queens problem to N queens and a $N \times N$ chess board, is not NP-Hard [13]. However, the N-Queens Completion problem, i.e. a variant of the N-Queens problem in which some queens are already placed on the board, has recently been proven to be NP-Complete [46].

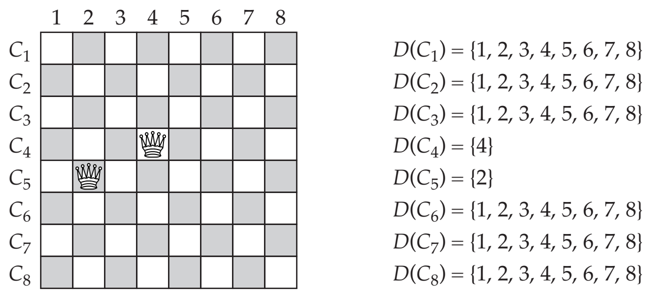

1 2 3 4 5 6 7 8
C_1
C_2
C_3
C_4
C_5
C_6
C_7
C_8
$D(C_1) = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$D(C_2) = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$D(C_3) = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$D(C_4) = \{4\}$
$D(C_5) = \{2\}$
$D(C_6) = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$D(C_7) = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$D(C_8) = \{1, 2, 3, 4, 5, 6, 7, 8\}$

Figure 6.2: CSP framing of the 8-Queens problem with two assigned queens.

A solution of a CSP is an assignment of all the variables to a value of their respective domain such that all the constraints are respected. A CSP is unfeasible if it has no solution.

The typical CSP formulation of the 8-Queens problem is the following. We assume that each queen is pre-assigned to a particular row so that the rule that queens should not share a same row is enforced by default. The problem thus becomes the one of finding a column for each queen so that:

1. no two queens share the same column;

2. no two queens share the same diagonal.

To do so, we associate each queen i to a variable C_i, with domain $D(C_i) = \{1, \ldots, 8\}$, that represents the column to which the queen could be placed in. Figure 6.2 illustrates this formulation where two queens are already placed on the board. The "column" and the "diagonal" rules are both enforced on all the C_i variables with three constraints: one for the columns, one for the upward diagonals, and one for the downward diagonals. While those constraints might look different, they actually share, thanks to a pinch of arithmetic[2], the same combinatorial *substructure*: ensure that no pair of variables is assigned to the same value. In CP, this particular substructure is better known as the `allDifferent` constraint and is part of a much larger catalogue of constraints [12]. The ability to identify those substructures is the key that allows CP practitioners to translate complex combinatorial problems into CSPs.

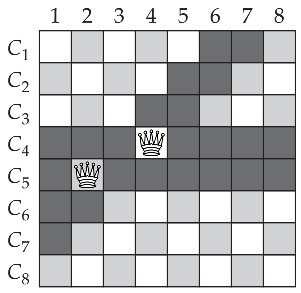
2 The Manhattan distance that separates cell (1,1) (resp. cell (1, 8)) from each cell contained in a same upward (resp. downward) diagonal is the same.

Though constraints enable the modeling of combinatorial problem in a expressive way, they also play a much significant role in the solving process of CP. Let us illustrate this by finding a solution to the problem depicted in Figure 6.2. An intuitive way to achieve this consists in removing the cells that are attacked by one of the already assigned queens. We perform such filtering by selecting one of the constraints, e.g. the upward diagonal constraint, and discarding the values that violate the constraint from the domain of the C_i variables. In CP, the process of removing values from the domain of a variable is called *filtering*, and the procedure used to perform such filtering according to a constraint definition is called a *propagator*. Figure 6.3 (a) illustrates the state of the CSP after calling the propagator of the upward diagonal constraint. We continue our filtering by calling the propagator of another constraint, the column constraint for instance (see Figure 6.3 (b)). This process of selecting a constraint and calling its propagator is repeated until no further filtering of the domains can be achieved. In CP, the algorithm that performs such process is called the *propagation algorithm* and its action usually referred to as *constraint propagation* or simply *propagation*. The state of the CSP after propagation, i.e. when no additional filtering can be performed by the propagators, is shown in Figure 6.3 (c).

Unfortunately, propagation alone is often not enough to infer a solution from a CSP. Indeed, and despite the significant reduction of the variables' domain depicted in Figure 6.3, no additional queen position was inferred by propagation. In such contexts, to continue further, we need to:

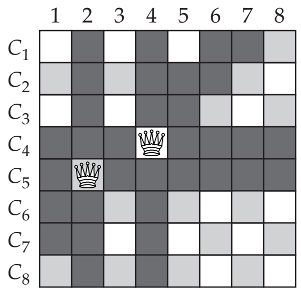
1. guess what could be the position of an unassigned queen;
2. check if our *decision* was valid or not by propagating this new piece of information.

Let us start by assigning the first queen to the 8th column, i.e. $C_1 = 8$, and then propagate. The result is shown in Figure 6.4. Like before, propagation wasn't enough to infer a solution from the assigned queens. We thus continue our guess and propagate process by assigning C_2 to 1. From this change, propagation is able to infer the position of the third queen but removes all the values from the domain of C_6 as it is impossible, given $C_1 = 8$ and $C_2 = 1$, to place a queen in the 6th row without violating at least one constraint (see the bottom left board in Figure 6.4). We say that propagation *failed*. We thus undo our last decision and try to assign the second queen to the third column instead. Unfortunately, this also results in a fail (see the bottom right board in Figure 6.4) thus proving that there is no solution such that the first queen is assigned to the 8th column. This brings us back to the beginning of our a)



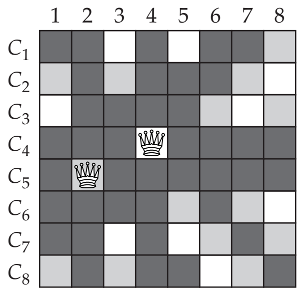
$D(C_1) = \{1, 2, 3, 4, 5, 8\}$
$D(C_2) = \{1, 2, 3, 4, 7, 8\}$
$D(C_3) = \{1, 2, 3, 6, 7, 8\}$
$D(C_4) = \{4\}$
$D(C_5) = \{2\}$
$D(C_6) = \{3, 4, 5, 6, 7, 8\}$
$D(C_7) = \{2, 3, 4, 5, 6, 7, 8\}$
$D(C_8) = \{1, 2, 3, 4, 5, 6, 7, 8\}$

b)



$D(C_1) = \{1, 3, 5, 8\}$
$D(C_2) = \{1, 3, 7, 8\}$
$D(C_3) = \{1, 3, 6, 7, 8\}$
$D(C_4) = \{4\}$
$D(C_5) = \{2\}$
$D(C_6) = \{3, 5, 6, 7, 8\}$
$D(C_7) = \{3, 5, 6, 7, 8\}$
$D(C_8) = \{1, 3, 5, 6, 7, 8\}$

c)



$D(C_1) = \{3, 5, 8\}$
$D(C_2) = \{1, 3, 7, 8\}$
$D(C_3) = \{1, 6, 7, 8\}$
$D(C_4) = \{4\}$
$D(C_5) = \{2\}$
$D(C_6) = \{5, 7, 8\}$
$D(C_7) = \{3, 5, 6, 8\}$
$D(C_8) = \{1, 3, 6, 7\}$

Figure 6.3: Constraint propagation.

search but with the additional information that value 8 can't be a member of $D(C_1)$ no matter the solution. We finally take the decision to assign the 7th queen to the 6th column, which, after inferring the positions of the remaining queens, results in the solution depicted in the middle right part of Figure 6.4.

This recursive process of guesses, propagations, and backtracks is called *backtracking search*. Backtracking search implicitly develops a *search tree* such that its root is the original CSP, leaves are either solutions or fails, and internal nodes are intermediate CSPs that still have to be completed (see Figure 6.4). Backtracking search is complete as it eventually finds all the solution to the problem or proves that the problem was *unsatisfiable* because it has no solution.

6.2 LARGE NEIGHBORHOOD SEARCH

CP is not restricted to solving CSP and can straightforwardly be extended to solve combinatorial optimization problems as well.[3] In CP, a combinatorial optimization problem is modeled as a CSP with an additional variable that represents the possible value of the objective function to be minimized. The link between the domain of that *objective variable* and the rest of the model is ensured by a constraint. Each time a solution is found, CP adds a new constraint to the model to force the objective variable to be assigned to a strictly lower value than the one of that solution. The optimality of a solution is proven when no additional solution can be found. Although this algorithm works pretty well on problems with small search trees, it often struggles to find high quality solutions when the search tree gets larger (either because of the size of the problem to be solved or the efficiency of the filtering algorithms). This weakness is inherent to the DFS nature of backtracking search which dives in the so-called "left-most" part of the search tree.

Throughout the last decades, CP researchers came up with different strategies (such as discrepancy search [57], back-jumping search [29, 71, 87], or conflict ordering search [43, 71]) to workaround this limitation. Among them, Large Neighborhood Search (LNS) [100] turned out to be particularly efficient to solve large scale real world industrial problems in the domain of vehicles routing [14, 93] and scheduling [49, 70, 82]. In essence, LNS is an LS algorithm: it starts with an initial solution and iteratively tries to improve it until a stop condition, such as a time limit, is met. The particularity of LNS stands in how the improvement step is performed. Rather than exploring neighborhood solutions defined by a set of small modifications of the best-so-far solution

3 CP can actually be extended further to solve multi-objective combinatorial problems. The author proposed two different approaches for that matter [56, 96].

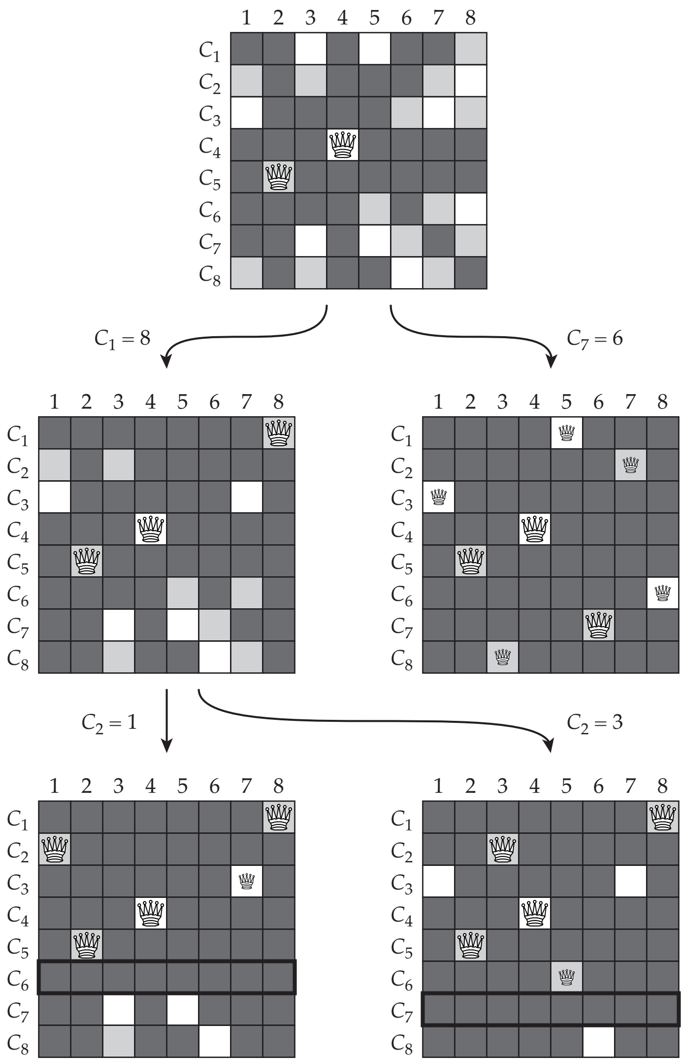


Figure 6.4: CP relies on an interleaved process made of guesses and propagations to solve CSPs. Small queens are inferred by propagation.

(like the *edit* neighborhood presented in Chapter 5), LNS selects a whole part of the best-so-far solution called a *fragment*, *deconstructs* it, and then relies on CP to *reconstruct* the missing part in a way that improves the quality of the solution (see Figure 6.5). Concretely, LNS achieves this by generating and solving a new subproblem $\mathcal{P}'$ such that

$$\mathcal{S}_{\mathcal{P}'} \subseteq \{s' \in \mathcal{S}_{\mathcal{P}} \mid f(s') < f(s)\} \tag{26}$$

Let F be a fragment made of a subset of the problem's variables and let O denote the objective variable to be minimized, subproblem $\mathcal{P}'$ is typically generated by adding the following constraints to the original problem:

$$O < f(x) \quad \text{and} \quad \forall i \notin F : X_i = s_i. \tag{27}$$

If CP finds a solution to problem $\mathcal{P}'$ then this solution becomes the new best-so-far solution. The typical LNS algorithm is the following:

```
1  function LargeNeighborhoodSearch($\mathcal{P}$):
2    $s \leftarrow$ FindSolutionCP($\mathcal{P}$)
3    if $s =$ null:
4      return null
5    while $\neg$StopCondition():
6      $\mathcal{P}' \leftarrow$ BuildNeighborhood($\mathcal{P}, s$)
7      $s' \leftarrow$ FindSolutionCP($\mathcal{P}'$)
8      if $s \neq$ null:
9        $s \leftarrow s'$
10   return $s$
```

LNS first uses CP to find an initial first solution s (line 2). If no solution exists, then LNS stops knowing that the problem is unfeasible (lines 3 and 4). The main loop of LNS (lines 5 to 9) is where the deconstruct/reconstruct iterative process occurs. Function `BuildNeighborhood` generates the subproblem $\mathcal{P}'$ (line 6) which is then solved by CP (line 7). If CP finds a solution then it becomes the new best-so-far solution (lines 8 and 9). This process is repeated, hopefully improving the quality of the solution, until a stop condition is met (line 5).

LNS provides us with two advantages compared to "classic" LS algorithms. First, it does not need tailored neighborhood functions to handle the constraints of the problem since those are naturally handled by CP during the reconstruction phase. The second advantage is that fragment selection can be done automatically without any prior knowledge of the problem to be solved (e.g. by selecting a random subset of variables).[4] Ultimately, LNS provides

4 However, problem specific fragment selection heuristics are likely to improve the efficiency of the LNS algorithm [49, 83].

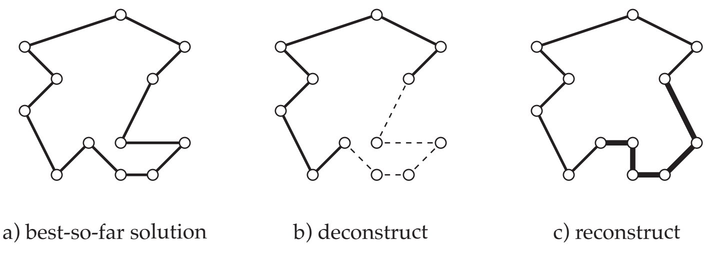


Figure 6.5: Iteration of the LNS deconstruct/reconstruct improvement process on an arbitrary instance of the Travelling Salesman Problem (TSP), i.e., the problem of finding an Hamiltonian cycle of minimum length. The best-so-far solution s is presented in (a); LNS deconstruct a part of s that looks suboptimal (b), and then tries to reconstruct it with CP in a way that improves the quality of the solution (c).

an elegant framework that combines the high expressivity and modularity of CP with the scalability of LS but at the cost of losing CP's ability to eventually find an optimal solution. However, many pragmatic solutions exist to workaround that limitation such as progressively increasing the size of the fragments or running LNS and CP in two different threads that communicate their best-so-far solutions.[5]

A lot more could be said about CP and LNS. We have only scratched the surface of the basic concepts necessary to understand the remainder of this chapter.[6] Nowadays, CP solvers such as [23, 45, 79, 81] are built with propagation algorithms and search strategies designed to be both efficient and correct but also fully generic: neither the propagation algorithm nor the backtracking search depends on the nature of the propagators or the variables as long as they both obey certain rules [54]. That makes CP solvers highly modular and easy to extend with new propagators, variables, and modeling abstractions.

6.3 structured domains

Not every combinatorial problem is easy to model using integer variables only. Indeed, a wide range of problems find a (much more) natural formu-

5 A similar strategy is implemented in CP-Optimizer [110].
6 Interested readers might want to have a look at the complete — but slightly outdated — handbook of constraint programming [94].

lation in higher-level languages such as those of sets [47, 88], strings [97, 98], graphs [33], or other discrete objects [94]. In the context of the SRTEP, it is natural to formulate the problem in terms of unknown SR paths. In this section we review different ways to extend CP solvers with such a language.

An intuitive way to implement the domain of a variable in CP is to explicitly use a set that contains all the values in that domain. This approach usually works pretty well for discrete variables with low cardinality domains such as integer or boolean variables. However, it is not well suited for SR path variables because their domain might contain an exponential number of values. To tackle this scalability issue, CP solvers usually reason on representations that overapproximate the actual domain of their variables, but provide more practical space complexities. Such representations are referred to as *structured domains* [94]. Rather than representing the domain of a variable exactly with a set, structured domains rely on smaller components (such as integers) which, when taken together in a structured way, form a representation that overapproximate the domain. In other words, structured domains trade the information contained in exact domain representations for scalability.

We illustrate the concept of structured domain with one of its most succesful application: *set variables*. Let X be a set variable defined on elements $I = \{1, \ldots, n\}$ so that $D(X) \subseteq P(I)$ where $P(I)$ is the power set of set I. The domain $D(X)$ grows exponentially with the number of elements n and contains up to 2^n different sets. Instead of representing the domain of variable X exactly with a single set, we rely on two different sets $I_{in} \subseteq I$ and $I_{out} \subseteq I$ which, respectively, represent the elements that must and cannot be part of every set in $D(X)$ [88, 115]. The structured domain built on pair $\langle I_{in}, I_{out} \rangle$ is defined as follows:

$$D(\langle I_{in}, I_{out} \rangle) = \{a \in P(I) \mid I^{in} \subseteq a \subseteq I \setminus I^{out}\} \subseteq D(X). \tag{28}$$

Note that (28) implies that sets I_{in} and I_{out} respect the following invariant:

$$I_{in} \cap I_{out} = \emptyset. \tag{29}$$

Clearly, $D(\langle I_{in}, I_{out} \rangle)$ is more scalable than the exact representation because it only requires two sets of n elements instead of one set of 2^n elements. However, it does not offer as much information as the exact representation (see Example 8). The direct implication is that structured domains might lead to faster but less effective propagation which, in turn, increases the size of the search tree to be explored with backtracking search. Finding the right balance between fast propagation and the size of the search tree is one among the many skills to be mastered by CP practitioners.

Example 8. Let X be a set variable such that $D(X) = \{\{a\}, \{b\}\} \subseteq P(\{a, b\})$. Sets I_{in} and I_{out} are empty because both elements a and b might (but are not guaranteed to) be part of the assigned set. The structured domain $D(\langle I_{in}, I_{out}\rangle) = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$ thus overapproximates $D(X)$ because it cannot exclude $\{\}$ nor $\{a, b\}$.[7]

6.3.1 *Related work on Path variables*

Path variables are useful to model combinatorial problems that involve unknown simple paths in a graph. Path variables are thus a natural fit to model our SR path variables as simple paths in the segment graph. Like set variables, path variables suffer from an important scalability issue since the number of paths (to be contained in their domain) grows exponentially with the size of the graph. It is therefore not surprising that structured domains for path variables have been particularly studied in the CP literature [94]. We briefly review the two most famous of them: the *link* and *node* representations.

LINK REPRESENTATION. In a link representation, each link $e \in \mathbf{E}$ is associated with a binary variable B_e, with domain $D(B_e) = \{0, 1\}$, that is assigned to 1 if link e is part of the path, or assigned to 0 otherwise. The attentive reader probably remembers that link representation corresponds exactly to the representation of SR paths we used in the MILP segment model presented in Chapter 4. Link representation has a space complexity of $\mathcal{O}(|\mathbf{E}|)$ which is especially appealing when dealing with sparse graphs. However, that space complexity reaches its worst case of $\Theta(|\mathbf{N}|^2)$ when link representation is used to model the domain of an SR path variable (because the segment graph is a complete graph).

Another significant issue with link representation is that only a small fraction of the set of the binary variables' possible assignments actually contains valid simple paths between the source and the destination of the represented SR path:

$$\mathcal{O}(|\mathbf{N}|!) < \mathcal{O}(|\mathbf{N}|^{|\mathbf{N}|}) = \mathcal{O}(2^{|\mathbf{N}| \log_2 |\mathbf{N}|}) \ll \mathcal{O}(2^{|\mathbf{N}|^2}).$$

In CP, inconsistent assignments are filtered out by additional constraints which, most of the time, slow down the propagation process. In Chapter 4, such filtering was performed by (i) the flow conservation constraint, and (ii) the objective function (which removed potential sub-cycles). CP solvers,

7 In practice, such overapproximations are prevented by including the cardinality of the set in its representation.

however, benefit from a tailored global constraint — originally named `path` constraint [12] — which performs the aforementioned filtering in a more efficient way.

NODE REPRESENTATION The idea behind node representation (also known as *successor model*) is to associate each node n to a variable S_n, with $D(S_n) \subseteq \mathbf{N}$, that represents the successor of n in the path. A path is thus represented as a sequence of "hops" from its source to its destination. If node n is not part of the path, then its successor variable S_n is assigned to n itself (see Figure 6.7). Node representation has been particularly successful to solve vehicle routing problems in CP [14, 94].

Like the one of link representation, the space complexity of a node representation is $\mathcal{O}(|\mathbf{E}|)$. However, the set of all possible assignments of a node representation is tighter than the one of a link representation. Indeed, node representation already filters out assignments that would have contained several links sharing the same origin. In other words, node representation can be seen as a strengthened version of link representation in which the "successor" constraint,

$$\forall u \in \mathbf{E} : \sum_{v \in \mathbf{E}} B_{u,v} = 1$$

is implicitly enforced by the structure. Although, sub-cycles still need to be removed from the possible assignments (see Figure 6.7). In CP, this is typically enforced with the `subtour` constraint [12] which, also, tends to slow down the propagation process.[8]

6.3.2 *Related work on String variables*

The domain of an SR path variable can easily be represented with either link or node representations. Both representations perform well when propagation tries to filter out paths that cannot contain a given link. In contrast, they fail to remove paths that contain a particular node at a given position as this cannot be translated in removing the value of a binary/successor variable (at least as long as all the previous nodes in the path are unknown). Another issue with those representations is that their space complexity of $\mathcal{O}(|\mathbf{E}|)$ remains expensive when the unknown path to be represented only contains a small

8 The `subtour` constraint actually ensures that the successor variables form a single sub-tour, not a simple path. This limitation is typically tackled by adding a fake edge (t, s) in the topology and assigning S_t to s.

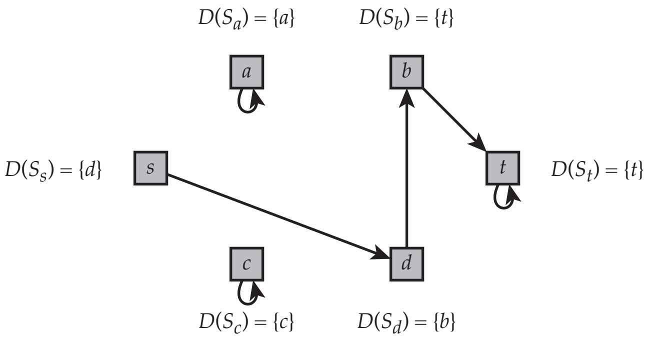


Figure 6.6: Node representation of path s, d, b, t.

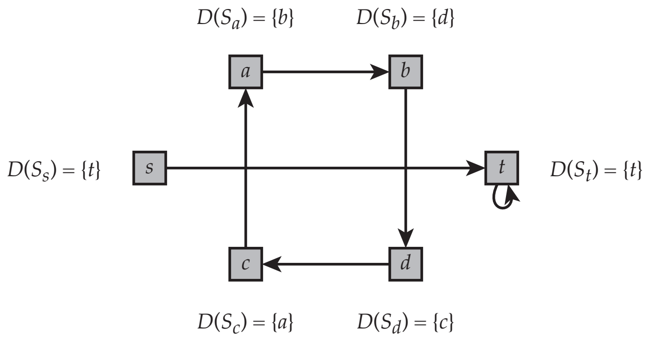


Figure 6.7: An assignment of the successor variables with a sub-cycle a, b, d, c.

number of links — which is likely to be the case of our unknown SR paths. The remainder of this section focuses on an alternative way to model our SR path variables with *string variables*. In particular, we present two structured domains for string variables which do not suffer from the two previous limitations.

String variables have numerous applications in a wide variety of combinatorial problems such as model checking [42] and software verification [19, 37, 98]. Formally, a string s of length $|s| = l$ is a finite sequence of l symbols, denoted by $s_1 s_2 \dots s_l$, where each s_i belongs to an alphabet Σ. Note that, without any loss of generality, we often represent the symbols in alphabet Σ with the natural numbers from 0 to $|\Sigma| - 1$. The infinite set of strings that can be composed over Σ, including the empty string, is a language denoted by Σ^*. Language Σ^l is a finite subset of Σ^* that contains all the strings over Σ of length l.

Naturally, an SR path from node s to node t can be seen as a string $m = m_0 \dots m_{k+1}$ where $m_0 = s$, $m_{k+1} = t$, and each symbol m_i in between corresponds to a midpoints in alphabet $\Sigma = \mathbf{N} \setminus \{s, t\}$. An unknown SR path of at most k midpoints can thus be represented by a string variable M such that

$$D(M) \subseteq \bigcup_{l \in [0,k]} \{s \cdot m \cdot t \in \mathbf{N}^{l+2} \mid m \in (\mathbf{N} \setminus \{s, t\})^l\} \tag{30}$$

where $\cdot$ is the string concatenation operator.

Like set and path variables, the exact domain of a string variable may contain an exponential (in the length of the considered strings) number of values. That makes string variables good candidates for structured domains. Structured domains for string variables are typically classified into three categories depending on the length of the unknown strings: *fixed*, *unbounded*, and *bounded*.

FIXED-LENGTH. Fixed-length string variables represent unknown strings for which the length is known a priori. In CP, fixed-length string variables can easily be modeled as an array of integer variables on which various well known constraints (e.g. the table constraint [31]) can be defined. Fixed-length string variables and constraints have been particularly studied in [66, 84, 89, 97].

UNBOUNDED-LENGTH. An unbounded-length string variable represents an unknown string of an unknown length. Unbounded string variables are typically defined by a regular language which is implemented with a finite automaton [1, 7, 42, 50]. Likewise, constraints over unbounded string variables are usually encoded as automata and enforced by performing automaton operations — mostly intersections — on the variables' domains. While expressive, this representation suffers from expensive propagation costs that are often quadratic in the size of the automata which, themselves, may grow exponentially with the complexity of the strings' regular languages. Avoiding such exponential blowup is still a very active research area [19].

BOUNDED-LENGTH. Bounded(-length) string variables are the best fit to represent our SR-path variables. A bounded string variable represents an unknown string for which its length is unknown but must remain below a known maximum allowed length. The first apparition of bounded string variables in CP can be credited to Maher [73] who described a model to perform propagation, by mean of *open* global constraints, on bounded sequences of (integer) variables. Contrarily to a *closed* global constraint, for which the cardinality of its scope is known in advance and remains fixed during the solving process, the cardinality of an open global constraint is unknown and determined by the solving process. In [73], the scope of an open global constraint is represented by (i) a sequence of integer variables, and (ii) an integer variable (with an upper bound) that corresponds to the length of the sequence. Each time the sequence needs to be extended, a new variable is added at its end and the length variable is updated accordingly. Later on, Scott et al. [97, 98], inspired by Maher's work, proposed the *open-sequence* representation to implement the domain of bounded string variables. An open-sequence variable is represented by a pair $\langle M, L\rangle$ where M is a sequence of sets (which are subsets of alphabet Σ), and L is a set of natural numbers that represents the possible length of the unknown string. Precisely, the domain $D(\langle M, L\rangle)$ of a string variable represented by pair $\langle M, L\rangle$ is the set of all strings that have a length $l \in L$ and are constructed by selecting a symbol from $M_i \subseteq \Sigma$ at each index $1 \leq i \leq l$:

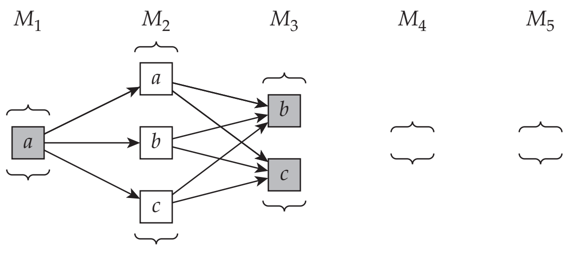
$$D(\langle M, L\rangle) = \bigcup_{l \in L} \{s \in \Sigma^l \mid \forall i \in [1, l] : s_i \in M_i\}. \tag{31}$$

Note that this representation must respect the following invariant:

$$\forall i \in [1, |M|] \; : \; M_i = \emptyset \Leftrightarrow \max(L) < i. \tag{32}$$

Figure 6.8 illustrates the open-sequence representation of an unknown string of at most 5 symbols in alphabet $\Sigma = \{a, b, c\}$. Both sets M_4 and M_5 are empty due to invariant (32) and length L. The variable's domain $D(\langle M, L\rangle)$ contains 7 strings: $a, aab, aac, abb, abc, acb,$ and acc. Note that $D(\langle M, L\rangle)$ contains no string of two symbols since $2 \notin L$.

Constraint propagation on open-sequence string variables usually provides good domain filtering but is computationally expensive. For example, let us



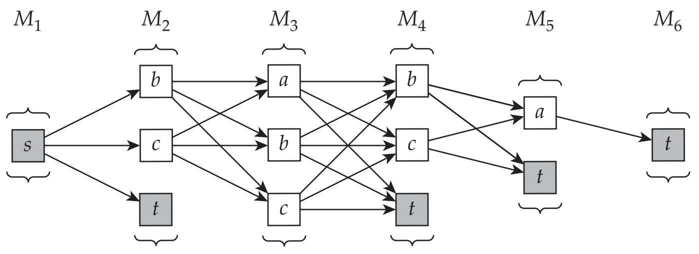
$\langle M = \{a\}\{a,b,c\}\{b,c\}\{\}\{\}, L = \{1,3\}\rangle$

Figure 6.8: Open-sequence representation of an unknown bounded string. Each path from node a in M_1 to a gray node represents a string in domain $D(\langle M, L\rangle)$.

design a simple propagator for the `maxCost` constraint (see Section 3.2.1) on a SR path variable implemented with an open-sequence string variable $\langle M, L\rangle$. Intuitively, a symbol $m \neq s, t$ cannot be a member of M_i if there is no path from s to t that (i) has m as its ith midpoint, and (ii) has a cost that is lower or equal to the maximum cost k. Let the *cost graph* of M be a DAG made of $|M|$ interconnected layers such that each symbols m in M_i corresponds to a node m_i in layer i (see Figure 6.9). Each node in the first layer is connected to the source of the SR path while every terminal layer, i.e. layer that might contain the last midpoint, is connected to its destination t. Also, each edge (u, v) has a weight that corresponds to the cost of reaching symbol v from symbol u in the cost matrix C. Let $path(u, v)$ be a function that returns the shortest path between nodes u and v in the cost graph. Similarly to the approach proposed by Sellmann et al. in [99], we rely on function $path$ to rewrite the filtering rule of the `maxCost` constraint as follows:

$$\forall i \in [1, \max(L)], \forall m \in M_i : path(s, m) + path(m, t) \leq k \qquad (33)$$

The value of $path(s, m)$ and $path(m, t)$ can be computed from scratch, for every symbol m, in $\mathcal{O}(\max(L)|M_*|^2)$ where M_* is the M_i with the largest cardinality. Precisely, we achieve this by computing the shortest paths tree from s to any node, and the shortest paths tree from any node to t, using topological ordering. While such algorithm can benefit from the typical incremental behavior of CP solvers to reduce its computation cost (i.e. we don't need to recompute the shortest paths from scratch at every change), we claim that the trade-off between filtering and computation is too poor in practice due to the high density of the cost graph.



$\langle M = \{s\}\{b, c, t\}\{a, b, c\}\{b, c, t\}\{a, t\}\{t\},\ L = \{2, 4, 5, 6\}\rangle$

Figure 6.9: The cost graph of open-sequence representation $\langle M, L\rangle$.

Next section presents an alternative representation, namely *increasing-prefix*, that overapproximates the domain of open-sequence string variables but enables much faster constraint propagation.

6.3.3 *Increasing-Prefix Representation*

Like Scott et al., we drew inspiration from Maher's work [73] to design a new representation for string variables called *increasing-prefix*.[9] Increasing-prefix representation is a lightweight representation for bounded string variables that overapproximates the domain of open-sequence string variables but enables faster constraint propagation. Let $next(M)$ be a function that returns the index of the first non-assigned sets in M,

$$next(M) = \underset{i \in [1, \max(L)]}{\operatorname{argmin}} \ |M_i| > 1 \tag{34}$$

so that the following invariant always holds:

$$\forall i \in [1, next(M) - 1] \ : \ |M_i| = 1. \tag{35}$$

The symbols assigned to the first $next(M) - 1$ sets of M form a *prefix* of the unknown string. In particular, the symbols contained in $M_{next(M)}$ are candidates to be appended to this prefix. The idea behind the increasing-prefix representation is to reason only on those candidates and to assume that they can be followed by any suffix. Formally, the domain D_{prefix} of a pair $\langle M, L\rangle$ is:

$$D_{prefix}(\langle M, L\rangle) = \bigcup_{l \in L} \{s \in \Sigma^l \mid \forall i \in [1, next(M)] : s_i \in M_i\}. \tag{36}$$

9 The work of Scott et al. was developed in parallel to this thesis.

Clearly, all pairs $\langle M, L\rangle$ that have the same known prefix p and the same set of next symbols $N = M_{next(M)}$ will have exactly the same domain under D_{prefix} no matter the value of the M_i for $next(M) < i \leq \max(L)$. In other words, given prefix p, set of next symbols N, and length L, function D_{prefix} defines the following equivalence class over the set of open-sequences:

$$\{\langle M, L\rangle \mid M_{|p|+1} = N \wedge \forall i \in [1, |p|] : M_i = \{p_i\}\}. \tag{37}$$

Example 9. Let us consider an alphabet $\Sigma = \{a, b, c\}$ and two pairs

$$\langle M_1 = \{a\}\{a, b\}\{b\}\{a, c\}, L = \{4\}\rangle,$$

$$\langle M_2 = \{a\}\{a, b\}\{a, b, c\}\{a, b, c\}, L = \{4\}\rangle.$$

Both representations share the same known prefix $p = a$ and the same set of candidates $N = \{a, b\}$. Domains $D_{prefix}(\langle M_1, L\rangle)$ and $D_{prefix}(\langle M_2, L\rangle)$ are equal and contain 18 different strings.

A direct consequence of (36) and (37) is that we can get rid of most of the sets contained in M to represent any increasing-prefix string variable with a triple $\langle p, N, L\rangle$ where p is the already known prefix of the unknown string, N is a set of symbols that can possibly be appended to prefix p, and L is a range that represent the possible length of the unknown string. The domain of an increasing-prefix string variable based on a triple $\langle p, N, L\rangle$ is the following

$$D(\langle p, N, L\rangle) = \bigcup_{l \in L} \{p' \cdot m \cdot s \in \Sigma^l \mid p' = p \wedge m \in N \wedge s \in \Sigma^*\} \tag{38}$$

To ensure the consistency of the representation, we need to ensure that the following invariant always holds:

$$|p| = L \quad \Leftrightarrow \quad N = \emptyset \tag{39}$$

Therefore, each time a new character is added to the prefix, the set of candidates N is either reinitialized to Σ if $|p| < \max(L)$, or becomes empty to indicate that the variable is assigned, i.e., p contains the whole string.

6.4 VARIABLE IMPLEMENTATIONS

This section presents two different way to implement an SR path variable based on an increasing-prefix representation. First, we show how to implement such a variable with objects that are commonly implemented by any CP solver:

ASSIGNED(P)	Return whether the variable P is assigned or not, i.e., if its prefix contains the whole path.
NEXT(P)	Return the set of candidates to be possibly appended to the prefix of P; or $\{t\}$ if the path is assigned.
PREFIX(P)	Return the known prefix of variable P.
REMOVE(P, m)	Remove midpoint m from the set of candidates. The function returns an inconsistency if m is the only remaining candidates.
APPEND(P, m)	Append candidate m to the prefix of P and reset the set of candidates if the P is not assigned. The function returns an inconsistency if m is not a valid candidate.

Table 3: List of operations supported by SR path variables.

integer variables and binary constraints. We then spend some time to present the sparse-set data structure which is at the core of our second implementation. We then conclude by presenting our second implementation which is based on a tailored data structure that achieves good time complexity while keeping a very low space complexity. In particular, both implementations implement the operations listed in Table 3 where P is an SR path variable and m is a node in $\mathbf{N}$.

6.4.1 *Hybrid implementation*

Our first implementation, namely *hybrid implementation*, consists in implementing our SR path variable as if it was relying on an open-sequence representation of its domain. This has two advantages. The first is that our hybrid implementation only relies on simple objects that are ubiquitous in CP solvers: integer variables and binary constraints. Our implementation is thus portable and easy to integrate on top of many solvers. The second advantage of our implementation is that it offers both domain representations at the same time. Constraints can thus benefit from having access to both domains to select the propagator that is the most suited for a particular context. For instance, a constraint might rely on a fast propagator that filters the domain of the increasing-prefix representation in the beginning of the solving process, and then switch to an open-sequence propagator once more information is available. Other strategies, such as relying on open-sequence propagators for specific constraints (and on increasing-prefix propagators for others) are totally possible too.

The hybrid implementation is implemented exactly like the open-sequence representation proposed by Scott et al. in [98]. Precisely, let us consider an open-sequence representation $\langle M, L\rangle$ defined on an alphabet $\mathbf{N}$ of natural numbers. The sequence of sets M is implemented with an array of integer variables $I = I_1, \dots, I_{|M|}$ such that $D(I_i) = M_i$. The length of the string is implemented with an integer variable I^L with $D(I^L) = L$. Note that invariant (32) needs to be slightly adapted since emptying the domain of variable is likely to lead to a failure in many CP solvers. We solve that issue by assigning variables that represent symbols that are not part of the path to its destination t. The invariant thus becomes the following:

$$I_1 = s \quad \wedge \quad \forall i \in 1, \dots, |I| : I_i = t \Leftrightarrow L \leq i. \tag{40}$$

It is implemented with simple binary constraints.

Pseudo code 6.1 shows how the hybrid implementation actually implements the interface of our SR path variable. We assume that methods REMOVEINT and ASSIGNINT are implemented by the CP solver and can be used to remove and assign a value from/to the domain of an integer variable.

6.4.2 *Sparse-sets*

A sparse-set [21] is a data structure to represent a subset S of the n first natural numbers:

$$S \subseteq \{0, \dots, n-1\}. \tag{41}$$

A sparse-set is implemented with three components: an integer `size`, and two arrays of length n named `values` and `positions`. Array `values` contains all the values in the range $[0, n[$ but not necessarily in order. The `size` first elements in `values` are the elements contained in S while the remaining elements are not:

$$S = \{\texttt{values}[0], \dots, \texttt{values}[\texttt{size} - 1]\}. \tag{42}$$

Array `positions` links each element to its position in `values` so that the following invariant always holds:

$$\forall i \in [0, n[\ : \texttt{values}[\texttt{positions}[i]] = i. \tag{43}$$

Using (43), we rewrite the definition of set S (42) as follows:

$$S = \{i \in [0, n[\ \mid \texttt{positions}[i] < \texttt{size}\}. \tag{44}$$

Figure 6.10 illustrates the link between `values`, `positions`, and `size` on a possible sparse-set representation of a set of 4 elements.

```
function prefixLength($P$) {
  if $|D(I_1)| > 1$ return 0
  return $\text{argmax}_{i \in 1,\ldots,|I|} |D(I_i)| = 1$
}

function assigned($P$) {
  return prefixLength($P$) $= |I|$
}

function next($P$) {
  if assigned($P$) return $D(I_{|I|})$
  return $D(I_{\text{prefixLength}(P)+1})$
}

function prefix($P$) {
  return $D(I_1), \ldots, D(I_{\text{prefixLength}(P)})$
}

method remove($P, m$) {
  removeInt(next($P$), $m$)
}

method append($P, m$) {
  assignInt(next($P$), $m$)
}
```

Pseudocode 6.1: Hybrid implementation of an SR path variable.

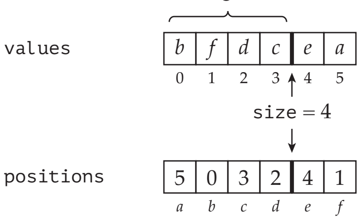


Figure 6.10: A sparse-set representing set $S = \{b, c, d, f\} \subseteq \{a, b, c, d, e, f\}$.

Iterate on the content of the set.	$\Theta(\|S\|)$
Return the size of the set.	$\mathcal{O}(1)$
Test if an element is in the set.	$\mathcal{O}(1)$
Remove an element.	$\mathcal{O}(1)$
Assign the set to a single element.	$\mathcal{O}(1)$
Undo the k last removals.	$\mathcal{O}(1)$

Table 4: Time complexity of various sparse-set operations.

The advantages of the sparse-set representation compared to other set representation, e.g. bit-vectors, is that it offers optimal time complexity for many interesting operations such as the ones listed in Table 4.[10] Indeed, most of those operations could trivially implemented by exchanging element positions and/or changing the value of `size`. For example, both remove and assign operations are presented in Pseudocode 6.2.

```
method REMOVE($m$) {
    size ← size − 1
    SWAP($m$, $size$)
}
method ASSIGN($m$) {
    SWAP($m$, 0)
    size ← 1
}
```

```
method SWAP($v_1$, $p_2$) {
    $p_1$ ← positions[$v_1$]
    $v_2$ ← values[$p_2$]
    positions[$v_1$] ← $p_2$
    positions[$v_2$] ← $p_1$
    values[$p_1$] ← $v_2$
    values[$p_2$] ← $v_1$
}
```

Pseudocode 6.2: Sparse-set remove and assign operations.

The last operation in Table 4 is particularly interesting. At a first glance, adding back k elements with a single $\mathcal{O}(1)$ operation looks suspicious. However, readers who are familiar with the implementation of a stack in an array might have identified a similar pattern in the way sparse-sets handle their set of removed elements. Let us consider a stack implemented from right to left in the `values` array such that `values`$[n-1]$ is the first element of the stack, its size is $n -$ `size`, and `values[size]` is its last element. Each time an element is removed from the set, using the `remove` operation presented in Pseudocode 6.2, the size of the stack is increased (by decreasing `size`) and the

10 In [27], le Clément et al. compare the efficiency of sparse-set operations to the ones of other representations. Note that Table 1 in [27] is not complete as it is also possible to find the minimum/maximum value contained in a sparse-set in $\mathcal{O}(N)$ by iterating on the `positions` array.

removed element is moved at the end of the stack. The direct consequence of this invariant is that the stack maintains the order in which elements have been removed from set S. Therefore, to restore the k last removed elements, we just have to increment `size` by k. In particular, we don't have to change anything in `values` and `positions` to ensure that invariants (43) and (44) still hold (see Figure 6.11).[11]

6.4.3 *Compact representation*

We now show how to benefit from sparse-sets to implement an SR path variable in a compact and efficient way. In contrast with the hybrid implementation, which has a space complexity of $O(\max(L)|\mathbf{N}|)$, our compact implementation has a tight space complexity of $\mathcal{O}(|\mathbf{N}|)$. Moreover, the compact implementation offers an optimal time complexity for all operations listed in Table 4.

In the compact implementation, each component of the increasing-prefix representation $\langle p, N, L \rangle$ is implemented in a different data structure. As before, we use an integer variable I^L, such that $D(I^L) = L$, to implement the length of the string. Prefix p is implemented with a stack of integers so that the top element of the stack corresponds to the last (right most) node in the prefix. This allows us to easily restore the previous state of the prefix by popping its last elements when a backtrack requires it. Set of candidates N is implemented with a sparse-set. Each time N is assigned to a node, that node is appended to the prefix by pushing it on the stack. Set N is then reset to contain a new whole set of candidates.[12]

The consistency between p, N, and the length variable I^L is ensured by the three following invariants:

$$|p| \leq \min(I^L) \tag{45}$$

$$|p| = \max(I^L) \;\Leftrightarrow\; p_{|p|} = t \;\Leftrightarrow\; N = \emptyset \tag{46}$$

11 The efficiency of this operation is one of the main reasons that explain the success of sparse-sets in CP solvers.

12 The cost of restoring the state of set N significantly depends on the state restoration mechanism implemented by the CP solver. We assume that the solver relies on *trailing* [54, 94] which is a common design choice in CP solvers. Otherwise, the solver might need to store any change applied to the structure which increases the actual space complexity of the representation to $\mathcal{O}(\max(L)|\mathbf{N}|)$ in the worst case. In such context, we recommend to rely on the hybrid implementation which has a similar cost.

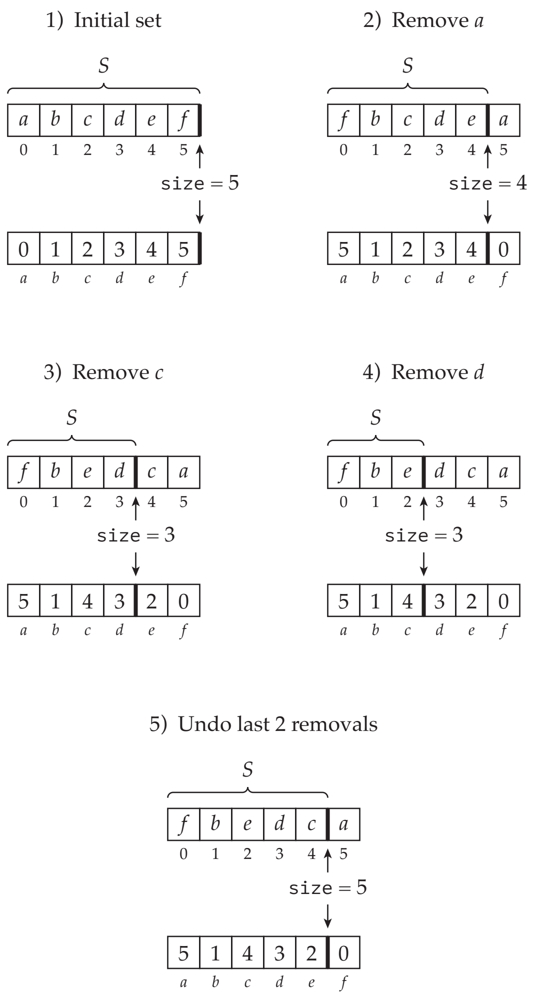


Figure 6.11: Remove and undo operations of a sparse-set. The k last removals can be undone in $\mathcal{O}(1)$.

$$|p| + 1 < \min(I^L) \Leftrightarrow t \notin N \tag{47}$$

The first invariant (45) ensures that the length variable is at least equal to the size of the current prefix. The second invariant (46) defines the state of each object when the SR path variable is assigned:

- the length variable I^L is assigned to the length of p;
- the last element prefix p is the destination t; and
- set of candidates N is empty.[13]

Finally, invariant (47) ensures that destination t is not a midpoint.

More often than not, it is possible to turn an SR path that contains a cycle to an SR path that doesn't without breaking any constraint (service chaining aside) and without impacting (negatively) the quality of the solution. In such context, it might be interesting to forbid cycles at first hand. Although this could be easily be achieved with a constraint, we propose to extend the compact implementation so that it implicitly enforces this property. We achieve this by implementing both p and N in a single sparse-set.[14] Rather than using a single integer `size`, our data structure now uses two integers `N` and `R`. Both integers partition array `values` into three parts. The first is made of the `N` elements of values and represents prefix p:

$$p = \texttt{values}[0], \ldots, \texttt{values}[\texttt{L - 1}]. \tag{48}$$

Elements from position `N` to `R` excluded represent the set of candidates N (with no guarantee on their order):

$$D(N) = \{i \in [0, |\mathbf{N}|[\mid \texttt{N} \leq \texttt{positions}[i] < \texttt{R}\}. \tag{49}$$

The last part, starting with the element at position `R`, represents the nodes that have been removed from the set of candidates.

The implementation of all the SR path variable's operations is presented in Pseudo code 6.3. We can see that most of them can be easily implemented in terms of sparse-set operations. The time complexity of those operations (and a few more) is presented in Table 5.

13 This is not an issue here because N is not implemented with an integer variable.
14 The idea is actually quite similar to the way set variables are implemented with sparse-sets [27]

```
function ASSIGNED($P$) {
  return N = R and values[N] = $t$
}

function NEXT($P$) {
  return values[N],...,values[R − 1]
}

function PREFIX($P$) {
  return values[0],...,values[N − 1]
}

method REMOVE($P, m$) {
  R = R − 1
  SWAP($m$, R)
}

method APPEND($P, m$) {
  SWAP($m$, N)
  N = N + 1
  R = N
}
```

Pseudocode 6.3: Compact implementation of an SR path variable.

Iterate on the known prefix.	$\Theta(\|p\|)$
Iterate on the content of the set of candidates.	$\Theta(\|N\|)$
Test if a node is a candidate.	$\mathcal{O}(1)$
Test if a node is in the prefix.	$\mathcal{O}(1)$
Get the position of a node in the prefix.	$\mathcal{O}(1)$
Remove an element from the set of candidates.	$\mathcal{O}(1)$
Append an element to the prefix and reset the set of candidates.	$\mathcal{O}(1)$

Table 5: Time complexity of various compact implementation's operations.

6.5 CONSTRAINTS ON SR PATH VARIABLES

Let $P = \langle p, N, L \rangle$ be an SR path variable implemented with a increasing prefix representation of its domain. Increasing prefix representation $P = \langle p, N, L \rangle$ is not a conventional representation of string variables. Therefore, dedicated propagators might be needed to maintain the property of the constraints. In this section, we describe such propagators.

All propagators follow the same incremental scheme, they maintain an incremental structure based on the value of p, this structure is restored each time p is restored during search. At each step, the constraint assumes that p does not break the filtering rule of the constraint and filters out candidate that would break this property if they were added to the prefix.

6.5.1 *MaxCost*

Let C be a cost matrix such that $C_{u,v}$ is the cost of edge (u, v) in the segment graph (i.e. the cost of forwarding graph $FG_{u,v}$). We define function *slack* such that, given SR path variable $P = \langle p, N, L \rangle$ and maximum cost k, it returns the total cost still available for P to reach its destination t:

$$slack(P) = k - \sum_{i=2}^{|p|} C_{p_{i-1},p_i} \tag{50}$$

Definition (1) of constraint `maxCost`(P, C, k) can thus be rewritten as follows:

$$slack(P) \geq 0. \tag{51}$$

Given cost matrix C, we compute a shortest path matrix C^* such that $C^*_{u,v}$ is the cost of the shortest path between node u and node v with regard to C. Shortest path matrix C^* allows us to define the following filtering rule:

$$\forall m \in N : C_{p_{|p|},m} + C^*_{m,t} \leq slack(P) \tag{52}$$

We can easily strengthen this filtering rule by considering the length L of SR path P. Let C^l be a shortest path matrix such that $C^l_{u,v}$ is the cost of a shortest path of exactly l edges between node u and node v. In particular, $C^1 = C$ and C^0 has a cost of 0 for any pair (u, v). The strengthened filtering rule of our propagator is the following:

$$\forall l \in L, \forall m \in N : C_{p_{|p|},m} + C^{l-|p|-1}_{m,t} \leq slack(P) \tag{53}$$

Given matrices C^l for $l \in [1, \max(L)]$, both filtering rules (strengthened or not) can be evaluated in constant time for each value of N and L. The light and strengthened propagators thus have a time complexity of $\Theta(|N|)$ ans $\Theta(|N||L|)$ respectively. Note that each matrix C^l only needs to be computed once for all the search process.[15]

6.5.2 *Service Chaining*

Let $P = \langle p, N, L \rangle$ be an SR path variable and $S = S_1, \ldots, S_k$ be a sequence of services such that $S_i \subseteq \mathbf{N} \setminus \{s, t\}$ for all $i \in \{1, \ldots, k\}$. The service chaining constraint, `serviceChaining`(P, S), enforces the following property:

$$p = s, \ldots, m_1, \ldots, m_2, \ldots, m_k, \ldots, t \tag{54}$$

where $m_i \in S_i$. Let suffix_S be a function that returns the suffix of $S = S_1, \ldots, S_k$ that still have to be visited by SR path P. We rely on function suffix_S to maintain (54) with two simple filtering rules to:

$$|\mathit{suffix}_S(P)| \leq \min(L) - |p|, \tag{55}$$

and

$$\forall m \in N : |\mathit{suffix}_S(P)| < \max(L) - |p| \vee m \in \mathit{suffix}_S(P)_1. \tag{56}$$

where $\mathit{suffix}_S(P)_1$ denotes the next set of service to visit. Filtering rule (55) ensures that length L is large enough to visit all the services that still have to be visited. In contrast, filtering rule (56) forces the visit of the next services when length L does not allow SR path P to visit any other node. First filtering rule is performed in $\mathcal{O}(1)$ while the second filtering rule is performed in $\mathcal{O}(|\mathbf{N}|)$ since it might remove up to $|\mathbf{N}| - 3$ nodes from N.

6.5.3 *Disjoint Paths*

Let function *visited* return the set of links visited by SR path variable P:

$$\mathit{visited}(P) = \bigcup_{i=2}^{|p|} \{(u, v) \in \mathbf{E} \mid FG_{p_{i-1}, p_i}(u, v) > 0\} \tag{57}$$

Constraint `disjoint`(P_1, P_2) ensures that SR paths P_1 and P_2 do not visit the same link:

$$\mathit{visited}(P_1) \cap \mathit{visited}(P_2) = \emptyset. \tag{58}$$

15 Cost matrices C^l for all $l \in [1, \max(L)]$ can be computed in $\mathcal{O}(|\mathbf{N}|^3 \max(L))$ with a dynamic programming algorithm [32].

where link (u, v) and link (v, u) are considered to be different. Let us define a second function, *shared*, which returns the set of links that will be visited next by SR path P no matter the candidate appended to its known prefix:

$$shared(P) = \bigcap_{m \in N} \{(u, v) \in \mathbf{E} \mid FG_{p_{|p|},m}(u, v) > 0\} \tag{59}$$

We finally use both functions (58) and (59) to define a third function

$$visited^*(P) = visited(P) \cup shared(P) \tag{60}$$

which returns the set of links that have already been visited or that will be visited next by SR path P. The filtering rule for the `disjoint` constraint is the following:

$$\forall m \in N^1 : \{(u, v) \in \mathbf{E} \mid FG_{p^1_{|p^1|},m}(u, v) > 0\} \cap visited^*(P_2) = \emptyset \tag{61}$$

Given $visited(P_2)$, a propagator for filtering rule (61) has a time complexity of $\mathcal{O}(|\mathbf{E}|)$ for each candidate m. Note that the filtering rule needs to applied symmetrically to P_2 too.

6.5.4 *Flow*

To the contrary of `maxCost`, `serviceChaining`, or `disjoint`, constraint `flow` does not aim at stating a property to be respected by the problem's solutions, but to help to formulate the problem as a CP model. Such constraints are often referred to as *channeling* constraint and are essential to ensure the link between different combinatorial structures within a CP model. Precisely, constraint `flow` maintains the link between an SR path variable P and a float variable $F^P_{u,v}$ which represents the quantity of flow that P could possibly send through link (u, v). Let $flow(P, u, v)$ be the quantity of flow already sent through link (u, v) by SR path variable P:

$$flow(P, u, v) = \sum_{i=2}^{|p|} FG_{p_{i-1},p_i}(u, v) \tag{62}$$

so that constraint `flow`$(P, F^P_{u,v})$ ultimately enforces the following property:

$$flow(P, u, v) = F^P_{u,v}. \tag{63}$$

This definition is maintained with three filtering rules:

$$\forall m \in N \; : \; FG_{p_{|p|},m}(u, v) + flow(P, u, v) \leq \max(F^P_{u,v}), \tag{64}$$

$$\min_{m \in N} FG_{p_{|p|},m}(u,v) + flow(P,u,v) \leq \min(F^P_{u,v}), \tag{65}$$

and

$$N = \emptyset \quad \Rightarrow \quad \max(F^P_{u,v}) = \min(F^P_{u,v}) = flow(P,u,v) \tag{66}$$

Filtering rule (64) is used to filter candidates that would send too much flow on link (u,v). Note that we can't use the lower bound of $F_{u,v}$ to filter candidates because SR path P might visit link (u,v) more than once.[16] Filtering rule (65) updates the lower bound to match the minimum quantity of flow that could be sent by the next candidate. Filtering rule (66) ensures that flow variable $F_{u,v}$ is assigned when SR path P is.

6.6 LINK-GUIDED LARGE NEIGHBORHOOD SEARCH

The SR path variables and constraints developed in the previous sections provide us with a complete language to naturally model the SRTEP as a CP problem. First, we associate each demand $d \in \mathbf{D}$ with an SR path variable P_d and $|\mathbf{E}|$ float variables $F^d_{u,v}$, where $(u,v) \in \mathbf{E}$, that represent the quantity of data sent by SR path P_d on each link:[17]

$$\forall (u,v) \in \mathbf{E} \;:\; \texttt{flow}(F^d_{u,v}, P_d). \tag{67}$$

For each link (u,v), we have an additional float variable $L_{u,v}$ to represent the load of link (u,v). The link between the flow variables and the load variable is ensured by a `sum` and `division` constraints:

$$L_{u,v} = \texttt{division}(\texttt{sum}(\{F^d_{u,v} \mid d \in \mathbf{D}\}), c_{u,v}) \tag{68}$$

where $c_{u,v}$ is the capacity of link (u,v). Finally, we model the network utilization with the an objective variable L and a `maximum` constraint over all the $L_{u,v}$ variables so that:

$$L = \texttt{maximum}(\{L_{u,v} \mid (u,v) \in \mathbf{E}\}). \tag{69}$$

In total, our model contains $|\mathbf{D}|$ SR path variables and $|\mathbf{D}||\mathbf{E}| + |\mathbf{E}| + 1$ float variables. Assuming that SR path variables are implemented with the compact representation, which has a linear space complexity in the number of nodes,

16 We could actually filter candidates using the lower bound of $F^P_{u,v}$ if no possible SR path, given the current prefix and set of candidates, would visit link (u,v).

17 Float variables are not mandatory and it is possible to model the SRTEP with integer variables only by scaling up and rounding the forwarding graph functions (see Definition 6).

the total space complexity of the model is $\mathcal{O}(|\mathbf{D}||\mathbf{E}|)$ which is competitive with the MILP models from Chapter 4.

We provide the CP solver with a basic search heuristic, that we refer to as *complete-first*, to drive its solving process. Let $\mathbf{D}_{unassigned}$ be the set of demands that have an unassigned SR path variable. At each node of the search tree, heuristic complete-first selects one demand $d \in \mathbf{D}_{unassigned}$ randomly where the probability p_d of being selected is determined by the demand's bandwidth bw_d:

$$p_d = \frac{bw_d}{\sum_{d' \in \mathbf{D}_{unassigned}} bw_{d'}}. \tag{70}$$

The heuristic then tries to *complete* SR path P_d by extending its prefix with its destination. If the path cannot be completed (i.e. if the destination is not a valid candidate), then complete-first tries to extend the prefix with the midpoint that induces the smaller increase of the network utilization (ties are broken randomly).

In theory, any CP solver running this framework should, eventually, find an optimal solution (and prove its optimality) to any instance of the SRTEP. In practice, though, CP solvers are unlikely to optimally solve the SRTEP in reasonable time. We evaluated our CP framework by solving all instances with less than 40 nodes on the 3-SRTEP. Results are presented in Figure 6.12 where the vertical axis is the ratio obtained by dividing the best objective value found, in a 5 minutes search, by the optimal one. While the results don't necessarily look bad — the algorithm reached optimality (though it wasn't able to prove it) on 32% of the considered instances and was "only" 84.11% higher than the optimal solution in the worst case — they are actually only comparable to the one obtained by the simple LS greedy algorithm presented in Chapter 5.

Those results are not surprising because the language we previously defined is purposely built on building lightweight structured domains and constraints that will perform fast constraint propagation but poor pruning. The goal of this trade-off is to build an LNS algorithm with our CP representation. We do that by updating the Link Guided Variable Search so that it relies on CP to explore its neighborhoods. The algorithm is the following:

```
function LinkGuidedLargeNeighborhoodSearch($\mathcal{P}, maxSize$):
  $s \leftarrow$ FindSolutionCP($\mathcal{P}$)
  if $s =$ null:
    return null
  $n \leftarrow 1$
```

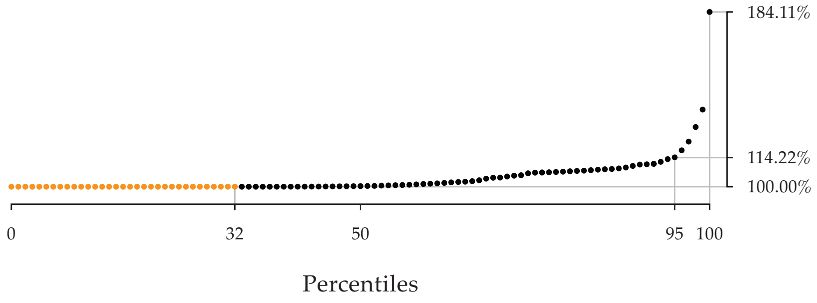


Figure 6.12: Comparison of the CP with the optimal instance of 3-SRTEP. The algorithm reaches optimality on 32% of the instances and is 84.11% higher than the optimal solution in the worst case.

```
6    while ¬StopCondition() :
7        $(u,v) \leftarrow$ SelectLink($s$)
8        $d_1,\ldots,d_n \leftarrow$ SortedDemands($u,v$)
9        $s' \leftarrow$ ExploreNeighborhood($\mathcal{P}, s, d_1,\ldots,d_n$)
10       if $s' \neq$ null :
11           $s \leftarrow s'$
12           $n \leftarrow 1$
13       else :
14           $n \leftarrow n+1$
15   return $s$
```

While most of LGLNS operates like LGVS, it differs in two simple but important ways. The first is that LGLNS does not rely on a provided initial solution and is able to compute its own with CP or to report that the problem is unfeasible if no solution exists (lines 2 to 4). The second difference appears in the neighborhood evaluation (line 9) which relies on `ExploreNeighborhood` (see below) instead of OrderedImprovement. Those differences aside, the algorithm operates exactly like LGVS: it first selects one of the most loaded link, selects n demands to be moved, and then explores the neighborhood.[18] The main difference is that we've replaced the call to OrderedImprovement in line 7 by a call to `ExploreNeighborhood`:

```
1  function ExploreNeighborhood($\mathcal{P}, s, d_1,\ldots,d_n$) :
2      $\mathcal{P}' \leftarrow \mathcal{P}$
3      add($\mathcal{P}', L < f(s)$)
```

18 LGLNS and LGVS have the same behavior when solving the 2-SRTEP with $n = 1$.

```
4     for $d \in \mathbf{D} \setminus \{d_1, \ldots, d_n\}$:
5         ADD($\mathcal{P}', P_d = s_d$)
6     return FindSolutionCP($\mathcal{P}'$)
```

First, a new problem $\mathcal{P}'$ is created as a copy of problem $\mathcal{P}$ (line 2). New constraints are added to $\mathcal{P}'$ to force the objective to be better (line 3), and to force unselected demands to be assigned to the same exact path than in the best-so-far solution (lines 4 and 5). CP finally tries to solve problem $\mathcal{P}'$ and returns the solution if any (line 6).

6.7 EVALUATION

In this section, we compare the efficiency of LGLNS with the models and algorithms presented in Chapters 4 and 5. In particular, we structure this section to answer three questions:

1. Can LGLNS solve constrained instances efficiently?

2. Can LGLNS find high quality solutions?

3. Can LGLNS scale?

As before, we base our answers on extensive experiments realized on the data set presented in Appendix A.

6.7.1 *Can LGLNS solve constrained instances efficiently?*

Yes, We have evaluated LGLNS by solving constrained instances of the 3-SRTEP with no more than 40 nodes.[19] For each instance, we've compared the value of the optimal solution (computed with the MILP path model) to the one obtained by LGLNS after 10000 iterations.[20] Our results, presented in Figure 6.13, suggest that LGLNS is a competitive technique to solve constrained instances of the SRTEP when MILP techniques are not applicable, e.g., for scalability reasons. Indeed, LGLNS found solutions that are less than 1% apart from the optimal ones in 11% of the cases, and less than 1.31% apart in 75% of the cases. However, the quality of the solutions degrades for the last 25 percentiles, especially after the 95th. The most extreme case being 35.66% higher than the optimal objective value.

19 Constraints have been generated as described in Section A.3.

20 LGLNS required 92 seconds to perform its 10000 iterations in the most extreme case.

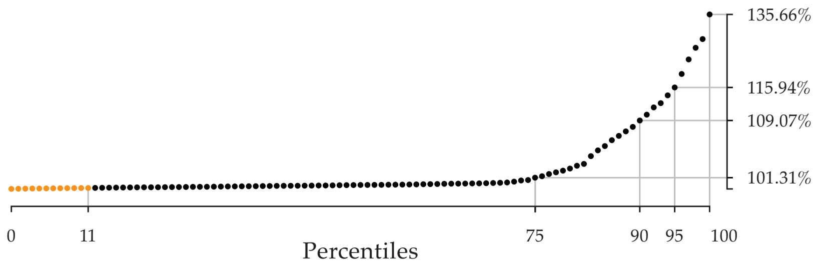


Figure 6.13: Efficiency of LGLNS on constrained instances of the 3-SRTEP. LGLNS reaches optimality in 11% of the cases and is 35.66% higher than the optimal value in the most extreme case.

6.7.2 *Can LGLNS find high quality solutions?*

Yes, but not as fast as "pure" LS algorithms. We have compared the results obtained by LGS, LGVS and LGLNS on our usual data set of instances of the 3-SRTEP with less than 40 nodes. For each instance, each algorithm was ran twice: with a limit of 10000 iterations and with a time limit of 30 seconds. Figure 6.14 shows that LGLNS achieves the best results in terms of iterations. This is interesting because LGS and LGLNS have exactly the same behavior if $n = 1$. Therefore, the fact that LGLNS manages to tackle the outliers (see oranges points) on which LGS gets stuck is due to LGLNS's ability to increase the size of its neighborhoods to escape local optima. Results are less convincing on the time limit side. Indeed, LGLNS performed pretty badly within the 30 seconds. This is interesting because LGLNS was able to perform its 10000 iterations in 92 seconds in the worst case. We explain this drastic difference by the computational cost induced by the additional data structures and algorithms used in LGLNS (e.g. state restoration, backtracking search, ...) that add a substantial overhead to its internal operations when used without any constraint.

6.7.3 *Can LGLNS scale?*

Looking at our previous results, the answer to that question was probably going to be "no" though it turned out that LGLNS has a very good scaling behavior if it is given with enough time to "warm-up". For this final experiment, we compared the results of the best approaches presented in this thesis

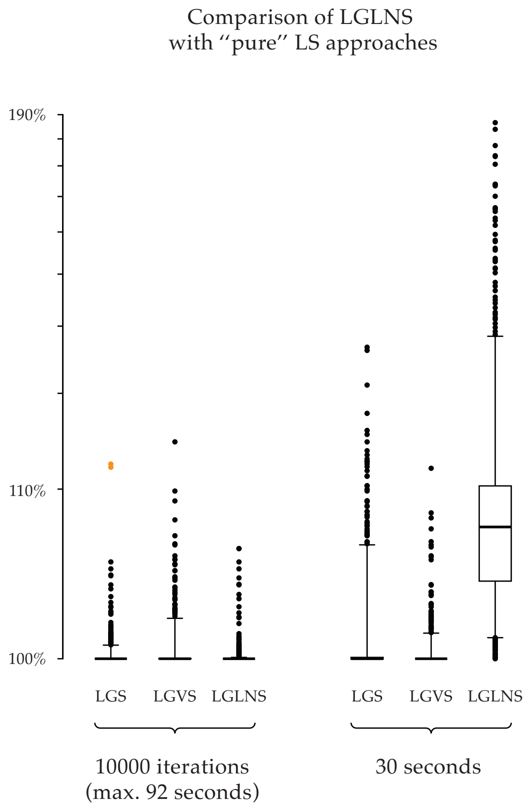


Figure 6.14: Comparison of LGS, LGVS, and LGLNS. LGNS performs better than both LS algorithms in terms of iterations but is significantly slowed down by its more complex internal data structures.

on our entire data set made of 3615 instances ranging from 3 to 284 nodes. In particular, we've divided our dataset into the three following categories:

- Small: instances with no more than 40 nodes;
- Medium: instances with a number of nodes that is between 41 and 80;
- Large: instances with more than 80 nodes.

Unfortunately, we weren't able to compute the optimal solution of all the 3-SRTEP instances neither with the path nor the segment models due to the size of the largest instances.[21] Therefore, for each instance, we ran each of the approaches presented in this thesis for 30 minutes and used the best solution found as a baseline for this experiment. Figure 6.15 shows the performance of the path model (MILP), LGVS, and LGLNS on the above categories given a time limit of 300 seconds.

We see that LGLNS is the clear winner on all instances. While MILP remains competitive on small instances, it disappears quickly from the ranking when the size of the instances increase. Indeed the path model was only able to solve a few instances (but not to prove their optimality) in the medium category and none in the large category. Results from LGVS and LGLNS are consistent on all the instances. On the first hand, LGLNS always performs better than LGVS in 300 seconds. On the other hand, LGVS is able to find similar solutions but much faster than LGLNS. Ultimately, both LGLNS and LGVS have very good scaling behavior compared to the path and segment models. It is also interesting to see that LGVS and LGLNS performs better on large instances. The main reason is that the best solutions found (on the medium and large datasets) were actually provided by LGVS and LGLNS. Therefore, results on the medium and large datasets essentially measure the ability of LGVS and LGLNS to converge quickly. However, we observed that most of these solutions are really close (if not equal) to the value of lower bound obtained by solving the MCFP. Our assumption is that large instances offer more flexibility (in terms of decisions) thus increasing the likelihood of escaping a local optima by moving a single demand.

6.8 CONCLUSION

In this chapter, we considered the use of Constraint Programming (CP) and Large Neighborhood Search (LNS) to design a fast and modular local search

21 Solving instances of the 3-SRTEP with 40 nodes on Gurobi 7.0 [52] already requires approximatively 30GB of memory.

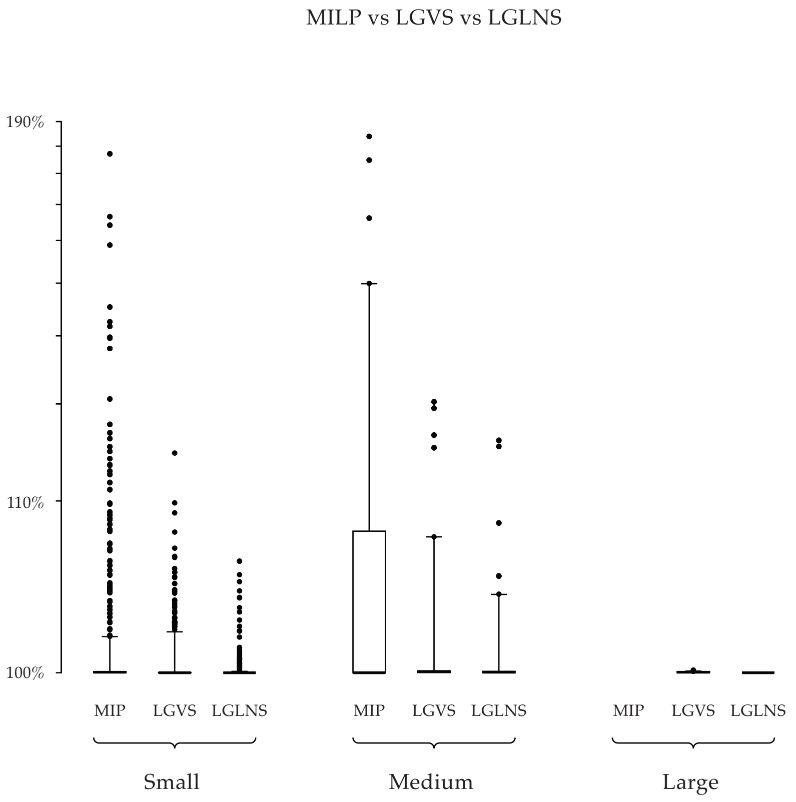


Figure 6.15: Comparison of MILP (path model), LGVS and LGLNS on all instances with a time limit of 300 seconds.

algorithm, called Link-Guided Large Neighborhood Search (LGNS), to solve constrained instances of the SRTEP. To achieve this, we defined a new domain representation as well as new propagators which, together, form a new language that extends CP solvers to model the SRTEP in a natural way. In particular, we focus on the design of lightweight structured domains that enable fast constraint propagation but at the cost of a weak filtering of the domain. This trade-off allows LGLNS to scale well to large instances while keeping a low space complexity of $\mathcal{O}(|\mathbf{D}||\mathbf{E}|)$ no matter the number of segments. Indeed, our results show that LGLNS is the best approach presented in this thesis to find high quality solutions on large instances. However, our results also highlighted that LGVS remains the best approach in terms of speed. We explain this by the computational cost associated to the more complex data structure and algorithms embedded in LGLNS. Though, this limitation is nothing that can't be tackled by a careful implementation of LGLNS's components as well as more computational power.

7 CONCLUSION AND FUTURE WORK

This thesis aimed at providing network operators with the necessary traffic engineering tools to build reliable and efficient segment routing SDN controllers. The first step of this long journey was to formalize the problem to be solved as a combinatorial optimisation problem that we refer to as the Segment Routing Traffic Engineering Problem (SRTEP). We showed that the SRTEP is difficult to solve in the general case as it belongs to the class of NP-Hard problems. Providing network operators with high quality solutions of the SRTEP is thus not a trivial task and we dedicated this thesis to the design of different algorithms built on a large set of technologies to tackle this combinatorial problem as efficiently as possible.

Our first step was to study the properties of the SRTEP using two Mixed Linear Integer Programming (MILP) models, namely, the path and segment models. Those models allowed us to answer important open questions about SR. First, it showed us that SR is a good technology from a traffic engineering stand point as it is able to reach the best possible network utilization on most of the considered instances. Our second — more interesting — observation is that IGP weights (which define the underlying forwarding graph that compose the SR paths) are critical to SR's ability to reach good network configuration. Indeed, we showed that random IGP weights often result in poor optimal solution while unary weights tend to behave extremely well (and better than the famous "inverse capacity" industry standard). Our third observation is that allowing SR paths with more than one midpoint can significantly improve the quality of the network configuration. In particular, we showed that increasing the number of midpoints from one to two typically results in significant improvement (however, increasing the limit to more than two midpoints only has a minor impact).

The path and segment MILP models allowed us to answer important questions about SR. However, they turned out to be too limited by their poor scaling behavior when used to solve large instances for the SRTEP. While more sophisticated MILP techniques such as column generation (see below) could have been considered, we chose to tackle this scalability issue by designing two Local Search (LS) algorithms: Link-Guided Variable Search (LGVS) and Link-Guided Large Neighborhood Search (LGLNS). LGVS is a pure local search algorithm that focuses on the fast re-optimisation of a given solution. LGLNS improves over LGVS by its ability to (i) compute the initial solution to be optimized by itself, and (ii) to dynamically adjust the size of its neighborhood to escape local optima. Our results showed that, despite being slower than LGVS, LGLNS remains highly scalable and performs better than any of the considered approaches to solve large instances of the SRTEP under reasonable time constraint (and small computational resources).

So, what techniques should you implement? The answer can actually vary from "all" to "none" depending on your requirements. If your network is small (less than 30 nodes) then MILP is probably the only approach you need to consider as it will be able to provide you with optimal solutions (in relatively short delays) while taking constraints in consideration. Another advantage of MILP is that state-of-the-art MILP solvers are in constant evolution and that you'd be able to benefit from the latest improvements in the domain without having to change anything to your model (thanks to declarativity) thus reducing your maintenance cost. Of course, the story is different if you need to build a fast controller for a large network (more than 100 nodes). In such contexts, LGVS and/or LGLNS might be the only practical solutions (especially if your computational resources are limited). The truth is that none of the proposed approaches wins in all contexts. Therefore, the best algorithm is probably the one that takes advantage of all the techniques simultaneously. For instance, one can easily come up with an hybrid approach that would run a MILP model, LGVS, and LGLNS in parallel, exchanging the best-so-far solutions found by each approach (thus speeding up the solving process of MILP) while letting MILP take care of eventually finding the optimal solution. As mentioned above, the technique presented in this thesis are not the only ones that could have been considered to solve the SRTEP. For example, the path model is particularly suited for column generation algorithms which are likely to improve the scalability of the MILP. Pseudo boolean SAT solvers [18, 34] (an extension of SAT to solve weighted binary sums) as well as LCG solvers [38, 79, 103] are good candidates to solve the SRTEP. Of course, all the local search algorithms presented in this thesis could easily be extended and improved with new neighborhood and meta-heuristics.

From a traffic engineering perspective, a few questions remain open on segment routing. The impact of the IGP weights on the ability of SR to find good solution is not well understood yet. We showed in Chapter 4 that poorly chosen weights can drastically hamper the ability of SR to find a high quality solution. However, we still don't know how to compute IGP weights that would leverage the power of SR. IGP weights that respect a strict triangle inequality (thus allowing the definition of any simple path as a sequence of midpoints) tend to behave very well in general but it is easy to come up with a small network topology that contradicts this conjecture. Optimizing both the IGP weights and the SR paths simultaneously is a challenging computational problem. Designing algorithms to solve this problem is definitely an interesting research topic but we believe that the impact of such research are limited in practice due to the disadvantages of dynamically changing the weights (see Chapter 2). We encourage network operators to optimize their link weight only once, to be good for a large set of scenario and to rely on SR to "fix" the configuration by rerouting the traffic and enforcing SLAs such as service chaining.

Minimizing the number of SR tunnels, i.e. the number of demands that are not routed the shortest paths directly, is another important issue that we did not directly addressed in this thesis. The problem with SR tunnels is that they need to be configured on the ingress node and therefore induce a configuration cost each time they are modified. Providing solutions that minimize the network utilization while minimizing the number of tunnels thus have significant practical advantages. Fortunately, the hybrid model can easily be extended for this matter by adding a new term to the objective function that is the sum of the number of paths that are the shortest paths. Network operators can then fine tune the coefficient to find different compromises and pick the one that best fits their needs. Measuring the impact of the number of midpoints on the number of tunnels is also an interesting research topic for future work.

This paragraph brings this thesis to an end. We hope that the reader enjoyed learning about the algorithms and techniques we developed to tackle the SRTEP and that our work will ultimately contribute to the development of new and innovative ideas. In particular, we hope that our model and algorithms will help to promote SR as a new standard for traffic engineering, and — who knows — to drive the implementation of the next generation SDN controllers responsible for managing many autonomous systems.

A

DATASET

An instance of the SRTEP is made of a network topology, a set of demands, and an optional set of additional constraints to be respected by the SR path of those demands. While random topologies and demands are easy to produce, they are likely to be far from the real world traffic engineering problems on which traffic engineering algorithms eventually have to exhibit good performances. This stresses the need to rely on a dataset made of real, or at least realistic, networks and demands. Unfortunately, real network datasets are likely to contain sensitive information for the network stakeholders and are, therefore, kept confidential.[1] This privacy consideration reduces the reproducibility — and thus the credibility — of important researches in the network community. It also hampers comparisons of approaches on the same scenario and ultimately hurts innovation. For all those reasons, we chose to (re-) evaluate the algorithms presented in this thesis on a large open-source dataset.

In this chapter, we present the dataset we built and used throughout this thesis. Basically, our dataset is an adaptation of the Repetita open source dataset [44] which regroups various open source datasets into a consistent and uniform one (see Section 1). We performed three transformations on Repetita to build our dataset. The first one, presented in Section 2, was to reduce the size of the topologies by removing parts that are not relevant for our traffic engineering problem. The second transformation was to extend Repetita with randomly generated sets of constraints. The way we have generated those constraints is explained in Section 3. Finally we rescaled all the demands of

1 We experienced a similar situation when developing Defo (see Chapter 6) conjointly with researchers from Cisco Systems. Defo was first evaluated on real topologies kindly provided by Cisco Systems but unfortunately protected by a non-disclosure agreement. Therefore, for the sake of reproducibility, the experiments presented in the Defo paper [55] were conducted on topologies randomly generated using IGen [91] and coming from the Rocket Fuel dataset [102].

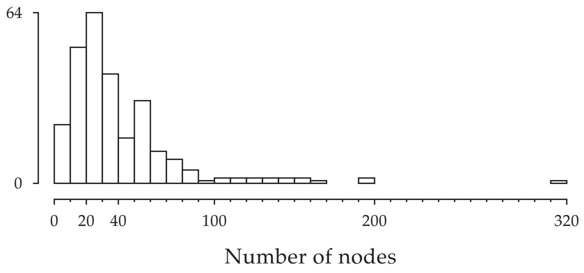


Figure 1.1: Distribution of Repetita's network sizes in terms of nodes. The 25th, 50th, and 75th percentiles are respectively 19, 29 and 53 nodes.

each instances so that the optimal network utilisation is 100%. The resulting dataset is presented in Section 4.

A.1 REPETITA

The Repetita dataset, or simply Repetita, is made of 269 strongly connected topologies. Those topologies range from tiny infrastructures of 4 nodes to mastodons of 319 nodes, but most of them are made of roughly 40 nodes (see Figure 1.1). Repetita's topologies mainly come from open-source datasets such as Topology Zoo [68] and Rocket Fuel [102]. Because of their various origins, the topologies in those datasets are often encoded in different file formats and provided with incomplete information. Rocket-fuel topologies, for instance, do not contain information about their link capacities. We briefly describe the methodology Repetita's authors used to normalize and complete those topologies to provide users with a consistent dataset.[2]

CONNECTIVITY If the topology is not connected, i.e. it is made of several connected components, only the largest component is kept and completed such that each link (u, v) has exactly one sibling (v, u).

LINK CAPACITY Missing link capacities in a topology are set to the median of all the provided ones. If the topology contains no link capacities, then they are all set to a symbolic value of 1 Gb/s. A post processing phase is then performed to increase the lowest link capacities so that they are not smaller

2 Repetita is still being developed at the time of writing this chapter. The observations and remarks made in this chapter are thus likely to become outdated with an upcoming version of Repetita.

than 1/20-th of the largest capacity. This aims at avoiding too large capacity range (e.g. including both Kb/s and Gb/s link capacities) that would likely lead to unavoidable performance bottlenecks which, in turn, hamper traffic engineering solutions.

DEMANDS Every topology in Repetita is provided with five randomly generated set of demands, and each of those sets contains exactly one demand between each pair of distinct nodes in the topology. To generate such demands, Repetita's authors relied on a variation of the gravity model [95]. Precisely, the demand to be forwarded from node u to node v, denoted by $\mathbf{D}_{u,v}$, is generated using the following function

$$\mathbf{D}_{u,v} = \frac{(p_u^{out} \sum_{(u,w)\in\mathbf{E}} c_{u,w})(p_v^{in} \sum_{(w,v)\in\mathbf{E}} c_{w,v})}{\sum_{(w,v)\in\mathbf{E}} c_{w,v}} \tag{71}$$

where p_u^{out} and p_v^{in} are independent, identically distributed exponential random variables. Demands are then normalized so that the value of the optimal network utilisation, assuming no constraint on the path demands are forwarded on, is 90%.[3]

A.2 ISLAND REMOVAL

This section presents a simple way to reduce the size of traffic engineering instances — and thus SRTEP instances — by analyzing the structure of the network topology. In Definition 1, we made the assumption that each edge $(u, v) \in \mathbf{E}$ has a sibling $(v, u) \in \mathbf{E}$. This assumption is convenient to analyze the properties of network topologies as if they were undirected graphs — which is precisely what we are doing here.[4]

Trees are boring from a traffic engineering perspective. The reason is that there is exactly one path between each pair of nodes in a tree. In other words, there is only one possible trivial solution to the SRTEP if the topology is a tree. While trees nullify the need for traffic engineering solutions, network designers tend to avoid them because they are extremely vulnerable to link failures. Indeed, any link failure will cut the network into two separated sub-networks that won't be able to communicate until the failure is repaired.[5]

3 Techniques to compute this optimal value are presented in Chapter 4.

4 This assumption does not reduce the generality of our approach since we can implicitly forbid the use of an edge by setting its capacity to 0.

5 Robust networks should actually be able to stay connected even when a node is removed thus removing several edges at the same time.

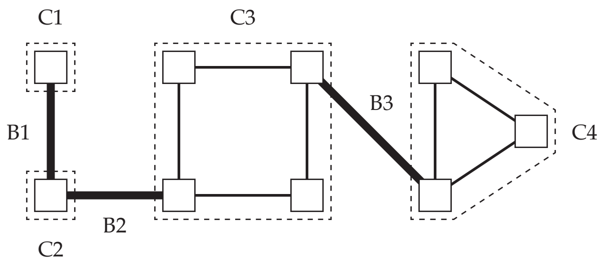


Figure 1.2: A topology made of 3 bridges (B1 to B3), 4 biconnected components (C1 to C4) and an island (C1).

Such critical links are best known as *bridges* and have been the subject of much study in graph theory.

Definition 17. A *connected component*, or simply *component*, of an undirected graph is a subgraph in which any pair of nodes is connected by a path in that subgraph.

Definition 18. A *bridge* is an edge of a component whose removal splits the component into two disconnected components.

Definition 19. a *biconnected component* — or *1-link-failure resilient component* in networking jargon — is a maximal component that contains no bridge. A biconnected component can be made of a single node though there is no biconnected component of two nodes by definition.

To sum up, a topology can be decomposed into a set of biconnected components that are connected by bridges. If the topology is a tree, then it is composed of $|N|$ biconnected components, each being made of a single node, connected by $|N| - 1$ bridges.

Definition 20. An *island* is a component of one node that is connected, by a bridge, to a single component.

Figure 1.2 illustrates the concepts of bridge, biconnected component, and island.

We now analyze some properties and theorems of islands and demands. Note that for the sake of conciseness, we only describe them for demands having an island as source. Demands having an island as destination are handled in a symmetrical way.

Property 2. Let us consider a topology made of an island i connected to component C by bridge (i,b). Then the forwarding graph from i to b is made of edge (i,b) only, i.e.,

$$\forall (u,v) \in \mathbf{E} \setminus \{(i,b)\} \,:\, FG_{i,b}(u,v) = 0.$$

Proof. The proof directly comes from Definitions 6 and 18. Since forwarding graphs are DAGs and that (i,b) is the only way to reach b from i then any other path contains a cycle and cannot be part of a forwarding graph. □

Property 3. A bridge that connects an island to another component C is the first edge visited by any path that connects that island to a node in C.

Property 3 is illustrated in Figure 1.2 where bridge B1 is the first edge of any path that connects C1 to a node in C2, C3, or C4. Similarly, B2 is the first edge of any path that connects C2 to a node in C3 or C4.

Theorem 2. *Let us consider an instance of the SRTEP such that its topology is made of an island i connected to a component C by bridge (i,b). Then every solution S such that an SR path p has i among its midpoints, $p = s, ..., i, ..., t$, can be turned into a solution S' such that $p = s, ..., b, ..., t$ and that $\phi_{S'} \leq \phi_S$.*

Proof. By Definition 18 and Property 2, we know that every SR path p having i as midpoint must pass by b before and after reaching i such that b can be inserted before and after i in the SR path of the demand without affecting the value of the solution:

$$\forall (u,v) \in \mathbf{E} : flow(s, ..., i, ..., t)(u,v) = flow(s, ..., b, i, b, ..., t)(u,v)$$

The subsequence b, i, b introduces a cycle and can be removed such that:

$$\forall (u,v) \in \mathbf{E} : flow(s, ..., b, ..., t)(u,v) =$$
$$flow(s, ..., i, ..., t)(u,v) - FG_{b,i}(u,v) - FG_{i,b}(u,v)$$

□

Theorem 3. *Let $\mathcal{I} = \langle T(\mathbf{N}, \mathbf{E}), \mathbf{D} \rangle$ be an instance of the SRTEP such that the topology is made of an island i, a bridge made of (i,b) and (b,i), and a component $C(\mathbf{N}_C, \mathbf{E}_C)$ such that*

$$\mathbf{N}_C = \mathbf{N} \setminus \{i\}, \quad \mathbf{E}_C = \mathbf{E} \setminus \{(i,b), (b,i)\}.$$

Then there is a reduced instance $\mathcal{I}^R = \langle T(\mathbf{N}_C, \mathbf{E}_C), \mathbf{D}^R \rangle$ such that $|\mathbf{D}^R| \leq |\mathbf{D}|$ and that the optimal solution of $\mathcal{I}^R$ can be turned into an optimal solution of $\mathcal{I}$ in polynomial time.

Proof. Let us partition the set of demands $\mathbf{D}$ into three subsets $\mathbf{D}_{i,b}$, $\mathbf{D}_{i,C}$, and $\mathbf{D}_C$ such that:

- $\mathbf{D}_{i,b} = \{(i,b,k) \in \mathbf{D}\}$ contains the demands from island i to node b;
- $\mathbf{D}_C = \{(c_1, c_2, k) \in \mathbf{D} \mid c_1, c_2 \in \mathbf{N}_C\}$ contains the demands between two nodes in C;
- $\mathbf{D}_{i,C} = \mathbf{D} \setminus (\mathbf{D}_{i,b} \cup \mathbf{D}_C)$ contains the demands from island i to a node $c \in C$ with $c \neq b$.

Clearly, the path of each demand in $\mathbf{D}_{i,b}$ is already fixed to i, b. By Theorem 2, we know that every demand in $\mathbf{D}_C$ will never visit island i thus making C the only part of the topology on which those demands will be routed. Demands in $\mathbf{D}_{i,C}$ are less convenient because they visit both the bridge (i,b) and at least one edge of C. In other words, demands in $\mathbf{D}_{i,C}$ prevent us from splitting the instance into two independent smaller instances. However, by Properties 3 and 2, we know that demands in $\mathbf{D}_{i,C}$ start by forwarding graph $FG_{i,b}$ no matter the rest of their path. This allows us to split every demand $d = (i, c \in \mathbf{N}_C, k)$ from $\mathbf{D}_{i,C}$ into two subdemands $d'_{i,b} = (i,b,k)$ and $d'_C = (b,c,k)$. We denote the set of all the $d'_{i,b}$ as $\mathbf{D}'_{i,b}$ and the set of all the d'_C as $\mathbf{D}'_C$. As before, we know that the path of each demand in $\mathbf{D}_{i,b} \cup \mathbf{D}'_{i,b}$ is already fixed to i, b. Also, we know that all demands in $\mathbf{D}_C \cup \mathbf{D}'_C$ will only be routed through component C. We can thus compute the load of bridge (i,b) and use it to compute a lower bound on the network utilisation:

$$\phi_{lb} = \sum_{(i,b,k) \in \mathbf{D}_{i,b} \cup \mathbf{D}'_{i,b}} k / c_{i,b}.$$

Since all demands that will visit (i,b) have been processed and that we know the load of (i,b), we can remove it from the topology and remove $\mathbf{D}_{i,b} \cup \mathbf{D}'_{i,b}$ from the demands. This leaves us with instance $\mathcal{I}^R$ made of topology $T(\mathbf{N}_C, \mathbf{E}_C)$, demands $\mathbf{D}^R = \mathbf{D}_C \cup \mathbf{D}'_C$, and lower bound ϕ_{lb}.

Once found, we can turn the optimal solution of $\mathcal{I}^R$ into an optimal solution of $\mathcal{I}$ by fixing all demands in $\mathbf{D}_{i,C}$ to the same path as their corresponding subdemand in $\mathbf{D}'_{i,b}$ — we just have to change the source of the SR path from b to i. Demands in $\mathbf{D}_{i,b}$ and $\mathbf{D}_C$ keep the same SR path as their corresponding demands in $\mathbf{D}^R$. □

Figure 1.3 illustrates the use of the *island removal* theorem to reduce the instance on the left to the instance on the right. Of course, the reduction rule of Theorem 3 can be applied recursively to the reduced instance to produce

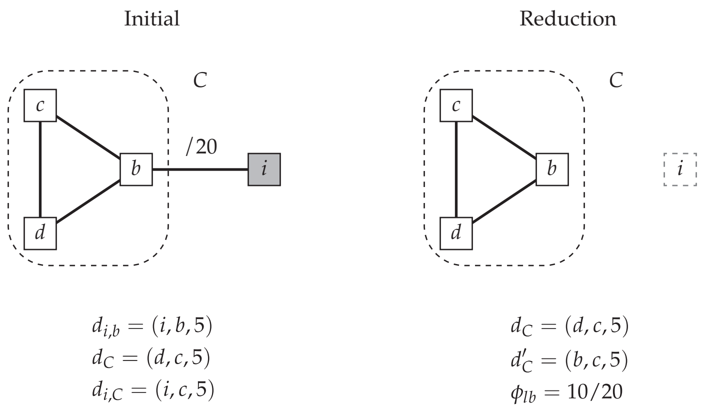


Figure 1.3: An instance of the SRTEP (left) and its reduction (right).

an instance that contains no island. Algorithm A.1 describes how to compute such a reduction in $\mathcal{O}(|\mathbf{D}| + |\mathbf{E}|)$. The resulting instance is such that it contains no component that contains less than tree nodes. For example, Figure 1.4 shows the reduction obtained on Repetita's Colt topology.

We measured the impact of the island removal on Repetita's topologies by comparing the number of nodes before and after applying Algorithm A.1 (see Figure 1.5 or Tables 6–14 for more precise results). From the 269 topologies contained in Repetita, 28 were trees and thus completely removed from the dataset. The remaining reduced topology are the basis on which our dataset is built.

A.3 ADDITIONAL CONSTRAINTS

Each topology in Repetita is provided with 5 different sets of demands (thus making 5 different instances per topology) but no set of constraints to be respected by the way those demands must be forwarded. That makes sense since Repetita is intended to be used to compare various traffic engineering approaches. This section describes how we randomly generated segment routing forwarding constraints in order to evaluate the efficiency of our algorithms in constrained traffic engineering scenarios. In particular, we generated three type of constraints, namely: `delay`, `disjoint`, and `firewall`.

DELAY The `delay` constraint is an instantiation of the `maxCost` constraint in which demands must be routed on an SR path that does not exceed 120% of

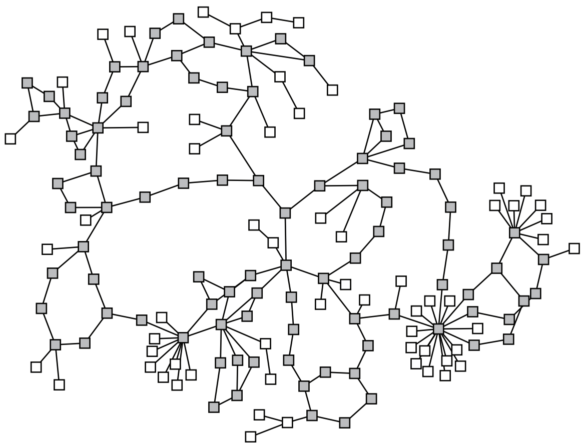


Figure 1.4: Colt topology where the white nodes have been removed by Algorithm A.1.

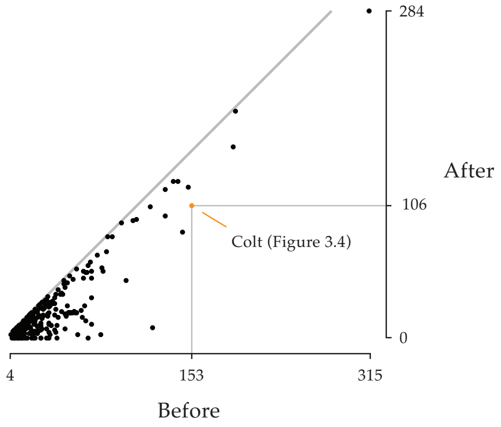


Figure 1.5: Number of nodes before and after applying Algorithm A.1 to Repetita's topologies.

```
function REDUCE($T(\mathbf{N},\mathbf{E}),\mathbf{D}$):
   $\mathbf{N}^R = \mathbf{N}$ , $\mathbf{E}^R = \mathbf{E}$ , $\mathbf{D}^R = \mathbf{D}$
   $deg = [a_1, ..., a_{|\mathbf{E}|}]$
   $S = \emptyset$
   $\phi_{lb} = 0$

   for $(u,v) \in \mathbf{E}^R$:
      $deg[u] = deg[u] + 1$

   for $n \in \mathbf{N}^R$:
      if $deg[n] = 1$:
         $S = S \cup \{n\}$

   while $S \neq \emptyset$:
      remove a node $i$ from $S$
      let $(i,b)$ and $(b,i)$ be both links of the only bridge of node $i$

      $load_{i,b} = \sum_{(i,c,k) \in \mathbf{D}^R} k$
      $load_{b,i} = \sum_{(c,i,k) \in \mathbf{D}^R} k$
      $\phi_{lb} = \max(\phi_{lb}, load_{i,b}/c^R_{i,b}, load_{b,i}/c^R_{b,i})$

      $\mathbf{N}^R = \mathbf{N}^R \setminus \{i\}$
      $\mathbf{E}^R = \mathbf{E}^R \setminus \{(i,b),(b,i)\}$
      $\mathbf{D}^R = \mathbf{D}^R \setminus (\{(i,c,k) \in \mathbf{D}^R\} \cup \{(c,i,k) \in \mathbf{D}^R\})$

      $deg[b] = deg[b] - 1$
      if $deg[b] = 1$:
         $S = S \cup \{b\}$

   return $\langle T(\mathbf{N}^R,\mathbf{E}^R), \mathbf{D}^R, \phi_{lb} \rangle$
```

Pseudocode A.1: A linear algorithm for island removal.

the delay of the SR path with no midpoint (i.e. the delay of the forwarding graph between the demand's source and destination). We assume that each link in the network is associated with a constant delay so that (i) the delay of a (simple) path is the sum of the delay of its links, and (ii) the delay of a forwarding graph is the maximum delay of any path in that forwarding graph. In other words, the delay of a forwarding graph corresponds to the delay-wise worst path on which a packet could be forwarded. We chose to enforce the `delay` constraint on 20% of the demands, selected randomly.

DISJOINT The `disjoint` constraint is exactly the same as the one presented in the previous chapter. We randomly selected 5% of the pairs of demands and constrained them to be routed on SR paths that do not share a same edge (if possible).

FIREWALL The `firewall` constraint is a `serviceChaining` constraint in which a demand must visit one of the firewalls of the network. We generate the set of firewall by selecting 10% of the nodes randomly, where the probability p_n of selecting node n is proportional to the total capacity of all the links connected to that node

$$p_n = \frac{\sum_{(n,v)\in \mathbf{E}} c_{n,v}}{2\sum_{(u,v)\in \mathbf{E}} c_{w,v}}.$$

The intuition behind this probability is that services, like firewalls, are more likely to be deployed on nodes that can handle important volume of traffic. We randomly selected 5% of the demands and constrained them to visit at least one of those firewalls.

Randomly generated sets of constraints are likely to lead to unfeasible instance of the SRTEP — especially if the generated sets are large. For instance, depending on the forwarding graphs, it might be impossible to forward two demands on two link-disjoint SR paths. Also, it might be impossible for a demand to visit a firewall (far in the network) without exceeding its maximum delay. Therefore, for each instance, we only kept the first random set of constraints for which a feasible solution exists.[6]

6 We used the MILP models presented in the next chapter to check the feasibility of each set of constraints.

A.4 NEW DATASET

The result of all the transformation presented in this chapter is a dataset made of the 241 (non-empty) reduced topology of Repetita (see Tables 6–14). For each of these topology, we rescaled the sets of demands, provided in Repetita, so that the optimal network utilisation, assuming no constraint on the path demands are forwarded on, is exactly 100% — which, we believe, eases the reading of the analyses presented in the subsequent chapters[7]. We then used each of these instances to create three additional instances in which shortest paths are computed using unary, inverse (of link capacity), and random IGP cost. Our final dataset is made of 3615 instances (241 topologies $\times$ 5 sets of demands $\times$ 3 sets of IGP weights) for which we have randomly generated a set of constraints as explained in the previous section.

7 We do understand that this might look extreme from a networking perspective.

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
Aarnet	19	48	15	40	19.04%
Abilene	11	28	11	28	25.45%
Abvt	23	62	22	60	12.98%
Aconet	23	62	18	52	16.99%
Agis	25	60	16	42	17.5%
Ai3	10	18	-	-	-
Airtel	16	74	8	36	64.28%
Amres	25	48	-	-	-
Ans	18	50	17	48	17.64%
Arn	30	58	-	-	-
Arnes	34	94	31	86	9.24%
Arpanet196912	4	8	3	6	100.00%
Arpanet19706	9	20	8	18	32.14%
Arpanet19719	18	44	18	44	14.37%
Arpanet19723	25	56	24	54	9.78%
Arpanet19728	29	64	29	64	7.88%
AsnetAm	65	158	25	74	12.33%
Atmnet	21	44	19	40	11.69%
AttMpls	25	114	25	112	18.66%
Azrena	22	50	5	8	40.00%
Bandcon	21	56	20	54	14.21%
Basnet	7	12	-	-	-
Bbnplanet	27	56	10	22	24.44%
Bellcanada	48	130	39	110	7.42%
Bellsouth	51	132	21	72	17.14%
Belnet2003	23	86	19	70	20.46%
Belnet2004	23	86	19	70	20.46%
Belnet2005	23	88	20	76	20.00%
Belnet2006	23	88	20	76	20.00%
Belnet2007	21	62	21	48	11.42%

Table 6: Topologies contained in our dataset (1/9).

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
Belnet2008	21	62	21	48	11.42%
Belnet2009	21	62	21	48	11.42%
Belnet2010	22	64	22	50	10.82%
BeyondTheNetwork	53	130	28	80	10.58%
Bics	33	96	27	84	11.96%
Biznet	29	66	26	60	9.23%
Bren	37	76	13	28	17.94%
BsonetEurope	18	48	12	34	25.75%
BtAsiaPac	20	62	14	50	27.47%
BtEurope	24	74	17	60	22.05%
BtLatinAmerica	45	100	19	48	14.03%
BtNorthAmerica	36	152	36	152	12.06%
Canerie	32	82	20	58	15.26%
Carnet	44	86	-	-	-
Cernet	41	118	30	94	10.80%
Cesnet1993	10	18	-	-	-
Cesnet1997	13	24	-	-	-
Cesnet1999	13	24	-	-	-
Cesnet2001	23	46	4	8	66.66%
Cesnet200304	29	66	11	30	27.27%
Cesnet200511	39	88	11	32	29.09%
Cesnet200603	39	88	11	32	29.09%
Cesnet200706	44	102	11	36	32.72%
Cesnet201006	52	126	19	60	17.54%
Chinanet	42	132	20	88	23.15%
Claranet	15	36	9	24	33.33%
Cogentco	197	490	167	426	1.53%
Colt	153	382	106	260	2.33%
Columbus	70	170	57	144	4.51%
Compuserve	14	34	11	28	25.45%

Table 7: Topologies contained in our dataset (2/9).

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
CrlNetworkServices	33	76	32	74	7.45%
Cudi	51	104	8	18	32.14%
Cwix	36	82	21	52	12.38%
Cynet	30	58	-	-	-
Darkstrand	28	62	28	62	8.20%
Dataxchange	6	22	5	20	100.00%
Deltacom	113	366	104	304	2.83%
DeutscheTelekom	30	110	26	102	15.69%
Dfn	58	174	51	160	6.27%
DialtelecomCz	138	302	106	238	2.13%
Digex	31	76	31	70	7.52%
Easynet	19	58	13	40	25.64%
Eenet	13	32	5	10	50.00%
EliBackbone	20	60	20	60	15.78%
Epoch	6	14	6	14	46.66%
Ernet	30	64	7	18	42.85%
Esnet	68	184	30	82	9.42%
Eunetworks	14	38	14	32	17.58%
Evolink	37	90	22	60	12.98%
Fatman	17	42	8	24	42.85%
Fccn	23	54	7	18	42.85%
Forthnet	62	124	3	6	100.00%
Funet	26	62	23	54	10.67%
Gambia	28	56	6	12	40.00%
Garr199901	16	36	4	12	100.00%
Garr199904	23	50	4	12	100.00%
Garr199905	23	50	4	12	100.00%
Garr200109	22	48	4	12	100.00%
Garr200112	24	52	4	12	100.00%
Garr200212	27	58	4	10	83.33%

Table 8: Topologies contained in our dataset (3/9).

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
Garr200404	22	48	4	12	100.0%
Garr200902	54	142	23	74	14.62%
Garr200908	54	136	22	72	15.58%
Garr200909	55	138	22	72	15.58%
Garr200912	54	136	22	72	15.58%
Garr201001	54	136	22	72	15.58%
Garr201003	54	142	23	74	14.62%
Garr201004	54	142	23	74	14.62%
Garr201005	55	144	23	74	14.62%
Garr201007	55	148	25	78	13.00%
Garr201008	55	148	25	78	13.00%
Garr201010	56	150	25	78	13.00%
Garr201012	56	150	25	78	13.00%
Garr201101	56	152	25	78	13.00%
Garr201102	57	154	25	78	13.00%
Garr201103	58	162	28	84	11.11%
Garr201104	59	166	28	86	11.37%
Garr201105	59	168	30	90	10.34%
Garr201107	59	170	30	90	10.34%
Garr201108	59	170	30	90	10.34%
Garr201109	59	172	31	92	9.89%
Garr201110	59	174	31	92	9.89%
Garr201111	60	174	30	88	10.11%
Garr201112	61	178	31	90	9.67%
Garr201201	61	178	31	90	9.67%
Gblnet	8	14	-	-	-
Geant2001	27	76	18	58	18.95%
Geant2009	34	104	30	96	11.03%
Geant2010	37	116	33	104	9.84%
Geant2012	40	122	32	106	10.68%

Table 9: Topologies contained in our dataset (4/9).

Name	$\lvert\mathbf{N}\rvert$	$\lvert\mathbf{E}\rvert$	$\lvert\mathbf{N}^R\rvert$	$\lvert\mathbf{E}^R\rvert$	Density
Getnet	7	16	5	12	60.00%
Globalcenter	9	72	9	72	100.00%
Globenet	67	226	58	172	5.20%
Goodnet	17	62	13	54	34.61%
Grena	16	30	-	-	-
Gridnet	9	40	9	40	55.55%
Grnet	37	94	13	36	23.07%
GtsCe	149	386	136	360	1.96%
GtsCzechRepublic	32	66	19	40	11.69%
GtsHungary	30	62	13	28	17.94%
GtsPoland	33	74	26	60	9.23%
GtsRomania	21	48	9	24	33.33%
GtsSlovakia	35	74	8	20	35.71%
Harnet	21	50	8	20	35.71%
Heanet	7	26	7	22	52.38%
HiberniaCanada	13	28	11	24	21.81%
HiberniaGlobal	55	162	54	160	5.59%
HiberniaIreland	8	16	5	10	50.00%
HiberniaNireland	18	44	18	42	13.72%
HiberniaUk	15	30	13	26	16.66%
HiberniaUs	22	58	15	44	20.95%
Highwinds	18	106	17	60	22.05%
HostwayInternational	16	42	15	40	19.04%
HurricaneElectric	24	74	17	60	22.05%
Ibm	18	48	17	46	16.91%
Iij	37	132	27	110	15.66%
Iinet	31	70	8	24	42.85%
Ilan	14	30	5	12	60.00%
Integra	27	72	23	64	12.64%
Intellifiber	73	194	66	176	4.10%

Table 10: Topologies contained in our dataset (5/9).

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
Internetmci	19	90	19	66	19.29%
Internode	66	156	19	60	17.54%
Interoute	110	316	104	280	2.61%
Intranetwork	39	106	32	88	8.87%
Ion	125	300	116	274	2.05%
IowaStatewideFiberMap	33	82	27	70	9.97%
Iris	51	128	45	116	5.85%
Istar	23	46	3	6	100.0%
Itnet	11	20	-	-	-
Janetbackbone	29	90	29	90	11.08%
JanetExternal	10	18	-	-	-
Janetlense	20	80	17	62	22.79%
Jgn2Plus	18	34	-	-	-
Karen	25	60	14	34	18.68%
KentmanApr2007	22	48	6	14	46.66%
KentmanAug2005	27	58	6	16	53.33%
KentmanFeb2008	26	56	6	14	46.66%
KentmanJan2011	38	78	5	10	50.00%
KentmanJul2005	16	34	8	18	32.14%
Kreonet	13	24	-	-	-
LambdaNet	42	92	35	78	6.55%
Latnet	69	148	15	40	19.04%
Layer42	6	14	4	10	83.33%
Litnet	43	86	5	10	50.00%
Marnet	20	54	10	34	37.77%
Marwan	16	36	8	18	32.14%
Missouri	67	166	58	148	4.47%
Mren	6	10	-	-	-
Myren	37	80	6	16	53.33%
Napnet	6	14	4	10	83.33%

Table 11: Topologies contained in our dataset (6/9).

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
Navigata	13	34	7	22	52.38%
Netrail	7	20	7	20	47.61%
NetworkUsa	35	78	33	74	7.00%
Nextgen	17	40	17	38	13.97%
Niif	36	82	19	48	14.03%
Noel	19	50	16	44	18.33%
Nordu1989	7	12	-	-	-
Nordu1997	14	26	-	-	-
Nordu2005	9	20	4	8	66.66%
Nordu2010	15	30	3	6	100.00%
Nsfcnet	9	20	6	14	46.66%
Nsfnet	13	30	10	24	26.66%
Ntelos	47	122	40	102	6.53%
Ntt	32	432	28	118	15.60%
Oteglobe	83	204	61	154	4.20%
Oxford	20	52	20	52	13.68%
Pacificwave	18	54	11	30	27.27%
Packetexchange	21	54	19	50	14.61%
Padi	7	12	-	-	-
Palmetto	45	140	45	128	6.46%
Peer1	16	40	12	32	24.24%
Pern	127	258	9	22	30.55%
PionierL1	36	82	29	68	8.37%
PionierL3	38	104	28	70	9.25%
Psinet	24	50	15	32	15.23%
Quest	20	62	19	60	17.54%
RedBestel	84	202	59	136	3.97%
Rediris	19	64	18	60	19.60%
Renam	5	8	-	-	-
Renater1999	24	46	-	-	-

Table 12: Topologies contained in our dataset (7/9).

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
Renater2001	24	54	12	30	22.72%
Renater2004	30	72	19	50	14.61%
Renater2006	33	86	27	74	10.54%
Renater2008	33	86	27	74	10.54%
Renater2010	43	112	38	102	7.25%
Restena	19	42	10	24	26.66%
Reuna	37	72	-	-	-
RF1221	104	302	50	194	7.91%
RF1239	315	1944	284	1882	2.34%
RF1755	87	322	75	298	5.36%
RF3257	161	656	115	564	4.30%
RF3967	79	294	72	280	5.47%
RF6461	138	744	129	726	4.39%
Rhnet	16	36	13	30	19.23%
Rnp	31	68	17	40	14.70%
Roedunet	42	100	8	24	42.85%
RoedunetFibre	48	104	23	54	10.67%
Sago	18	34	-	-	-
Sanet	43	90	25	54	9.00%
Sanren	7	14	7	14	33.33%
Savvis	19	40	17	36	13.23%
Shentel	28	70	20	54	14.21%
Sinet	74	152	12	28	21.21%
Singaren	11	20	-	-	-
Spiralight	15	32	15	32	15.23%
Sprint	11	36	10	34	37.77%
Sunet	26	98	26	64	9.84%
Surfnet	50	146	48	132	5.85%
Switch	74	184	52	140	5.27%
SwitchL3	42	126	30	102	11.72%

Table 13: Topologies contained in our dataset (8/9).

Name	$\|\mathbf{N}\|$	$\|\mathbf{E}\|$	$\|\mathbf{N}^R\|$	$\|\mathbf{E}^R\|$	Density
Synth100	100	572	100	572	5.77%
Synth200	199	1044	197	1040	2.69%
Synth50	50	276	50	276	11.26%
Syringa	74	148	35	70	5.88%
TataNld	145	388	137	356	1.91%
Telcove	71	140	-	-	-
Telecomserbia	6	12	6	12	40.00%
Tinet	53	178	48	168	7.44%
TLex	12	32	5	12	60.0%
Tw	71	236	64	216	5.35%
Twaren	20	40	4	8	66.66%
Ulaknet	82	164	3	6	100.00%
UniC	25	58	15	34	16.19%
Uninet	13	50	12	34	25.75%
Uninett2010	74	202	58	170	5.14%
Uninett2011	69	196	57	168	5.26%
Uran	24	48	4	8	66.66%
UsCarrier	158	378	135	332	1.83%
UsSignal	61	158	60	154	4.35%
Uunet	49	168	38	146	10.38%
Vinaren	25	52	6	14	46.66%
VisionNet	24	46	-	-	-
VtlWavenet2008	88	184	88	184	2.40%
VtlWavenet2011	92	192	88	184	2.40%
WideJpn	30	66	11	28	25.45%
Xeex	24	68	22	64	13.85%
Xspedius	34	98	33	96	9.09%
York	23	48	20	42	11.05%
Zamren	35	68	-	-	-

Table 14: Topologies contained in our dataset (9/9).

B

BIBLIOGRAPHY

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *International Conference on Computer Aided Verification*, pages 150–166. Springer, 2014.

[2] Ilan Adler, Mauricio GC Resende, Geraldo Veiga, and Narendra Karmarkar. An implementation of karmarkar's algorithm for linear programming. *Mathematical programming*, 44(1):297–335, 1989.

[3] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows: theory, algorithms, and applications. 1993.

[4] M. Al-Fares et al. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[5] Ayşegül Altın, Pietro Belotti, and Mustafa Ç Pınar. Ospf routing with optimal oblivious performance ratio under polyhedral demand uncertainty. *Optimization and Engineering*, 11(3):395–422, 2010.

[6] Ayşegül Altin, Bernard Fortz, and Hakan Ümit. Oblivious ospf routing with weight optimization under polyhedral demand uncertainty. *Networks*, 60(2):132–139, 2012.

[7] Roberto Amadini, Graeme Gange, Peter J Stuckey, and Guido Tack. A novel approach to string constraint solving. In *International Conference on Principles and Practice of Constraint Programming*, pages 3–20. Springer, 2017.

[8] D Awduche, L Berger, D Gan, T Li, V Srinivasan, and G Swallow. Rsvp-te: Extensions to rsvp for lsp tunnels (rfc 3209), 2001.

[9] D Awduche, A Chiu, A Elwalid, I Widjaja, and X Xiao. Overview and principles of internet traffic engineering–rfc3272. *IETF*, 2002.

[10] Simon Balon, Fabian Skivée, and Guy Leduc. How well do traffic engineering objective functions meet te requirements? In *International Conference on Research in Networking*, pages 75–86. Springer, 2006.

[11] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Oper. Res.*, 46(3):316–329, March 1998.

[12] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.

[13] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009.

[14] R. Bent and P.V. Hentenryck. A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers & Operations Research*, 33(4):875–893, 2006.

[15] Dimitri P Bertsekas and Robert G Gallager. Distributed asynchronous bellman-ford algorithm. *Data networks*, page 4, 1987.

[16] M Bezzel. Proposal of 8-queens problem, berliner schachzeitung 3 (1848) 363. *Schachfreund*.

[17] Randeep Bhatia, Fang Hao, Murali Kodialam, and TV Lakshman. Optimized network traffic engineering using segment routing. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 657–665. IEEE, 2015.

[18] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[19] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321, 2009.

[20] Beranek Bolt. *A History of the ARPANET: The First Decade*. Bolt Beranek and Newman, 1981.

[21] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):59–69, 1993.

[22] M. Casado et al. Rethinking enterprise network control. *Trans. Netw.*, 17(4):1270–1283, 2009.

[23] Xavier Lorca Charles Prud'homme, Jean-Guillaume Fages. Choco3 documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[24] Cisco. Planning and Designing Networks with the Cisco MATE Portfolio. white paper, 2013.

[25] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[26] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1991.

[27] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[28] Jeff Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 1, 2009.

[29] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

[30] Natalie Degrande, Gert Van Hoey, Paloma De La Vallée Poussin, and Sven Van den Bosch. Inter-area traffic engineering in a differentiated services network. *Journal of Network and Systems Management*, 11(4):427–445, 2003.

[31] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–223. Springer, 2016.

[32] Martin Desrochers and François Soumis. A generalized permanent labeling algorithm for the shortest path problem with time windows. *INFOR Information Systems and Operational Research*, 1988.

[33] Grégoire Dooms, Yves Deville, and Pierre Dupont. Cp (graph): Introducing a graph computation domain in constraint programming. In *Principles and Practice of Constraint Programming-CP 2005*, pages 211–225. Springer, 2005.

[34] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[35] Anwar Elwalid et al. MATE: multipath adaptive traffic engineering. *Computer Networks*, 40(6):695–709, 2002.

[36] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. Mate: Mpls adaptive traffic engineering. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1300–1309. IEEE, 2001.

[37] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162. ACM, 2007.

[38] Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *International Conference on Principles and Practice of Constraint Programming*, pages 352–366. Springer, 2009.

[39] C. Filsfils et al. Segment Routing Architecture. Internet draft, draft-filsfils-spring-segment-routing-00, work in progress, 2014.

[40] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM 2000. Nineteenth annual joint conference of the IEEE computer and communications societies. Proceedings. IEEE*, volume 2, pages 519–528. IEEE, 2000.

[41] Bernard Fortz and Mikkel Thorup. Optimizing ospf/is-is weights in a changing world. *IEEE journal on selected areas in communications*, 20(4), 2002.

[42] Graeme Gange, Jorge A Navas, Peter J Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In *TACAS*, pages 277–291. Springer, 2013.

[43] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *International conference on principles and practice of constraint programming*, pages 140–148. Springer, 2015.

[44] Steven Gay, Pierre Schaus, and Stefano Vissicchio. REPETITA: repeatable experiments for performance evaluation of traffic-engineering algorithms. *CoRR*, abs/1710.08665, 2017.

[45] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[46] Ian P Gent, Christopher Jefferson, and Peter Nightingale. Complexity of n-queens completion. *Journal of Artificial Intelligence Research*, 59:815–848, 2017.

[47] Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In *ILPS*, volume 94, pages 339–358, 1994.

[48] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[49] Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized large neighborhood search for cumulative scheduling. In *ICAPS*, volume 5, pages 81–89, 2005.

[50] Keith Golden and Wanlin Pang. Constraint reasoning over strings. In *International Conference on Principles and Practice of Constraint Programming*, pages 377–391. Springer, 2003.

[51] MM Group et al. World internet users statistics and 2015 world population stats, 2015.

[52] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2015.

[53] F. Hao, M. Kodialam, and T. V. Lakshman. Optimizing restoration with segment routing. In *INFOCOM*, 2016.

[54] Renaud Hartert. Kiwi-a minimalist cp solver. *arXiv preprint arXiv:1705.00047*, 2017.

[55] Renaud Hartert et al. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *SIGCOMM*, 2015.

[56] Renaud Hartert and Pierre Schaus. A support-based algorithm for the bi-objective pareto constraint. 2014.

[57] William D Harvey and Matthew L Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

[58] John Hawkinson and Tony Bates. Rfc 1930: Guidelines for creation, selection, and registration of an autonomous system (as). *BBN Planet/MCI, March*, page 10, 1996.

[59] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT press, 2009.

[60] C-Y Hong et al. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*, 2013.

[61] Cisco Visual Networking Index. The zettabyte era–trends and analysis. *Cisco white paper*, 2016.

[62] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[63] Kim L Jones, Irvin J Lustig, Judith M Farvolden, and Warren B Powell. Multicommodity network flows: The impact of formulation on decomposition. *Mathematical Programming*, 62(1):95–117, 1993.

[64] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *Proceedings of ACM SIGCOMM*, New York, New York, USA, August 2014.

[65] Jeff L Kennington. A survey of linear cost multicommodity network flows. *Operations Research*, 26(2):209–236, 1978.

[66] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. Hampi: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116. ACM, 2009.

[67] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[68] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[69] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[70] P. Laborie and D. Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. *Proceedings MISTA-07, Paris*, pages 276–284, 2007.

[71] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

[72] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[73] Michael J Maher. Open constraints in a boundable world. In *CPAIOR*, volume 9, pages 163–177. Springer, 2009.

[74] N. McKeown et al. OpenFlow: enabling innovation in campus networks. *ACM CCR*, 38(2):69–74, 2008.

[75] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[76] C. Metz, C. Barth, and C. Filsfils. Beyond MPLS ... Less is More. *IEEE Internet Computing*, 11(5):72–76, Sept 2007.

[77] John Moy. rfc 2328: Ospf version 2, 1998.

[78] Juniper Networks. What's Behind Network Downtime? Proactive Steps to Reduce Human Error and Improve Availability of Networks. Technical report, 2008.

[79] Or-tools Team. or-tools: Google optimization tools, 2015. Available from `https://developers.google.com/optimization/`.

[80] David Oran. Is-is intra-domain routing protocol rfc 1142. 1990.

[81] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[82] D. Pacino and P. Van Hentenryck. Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pages 1997–2002. AAAI Press, 2011.

[83] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. *Principles and Practice of Constraint Programming–CP 2004*, pages 468–481, 2004.

[84] Gilles Pesant. A regular language membership constraint for finite sequences of variables. *CP*, 3258:482–495, 2004.

[85] Michal Pióro and Deepankar Medhi. *Routing, flow, and capacity design in communication and computer networks*. Elsevier, 2004.

[86] M Piórob, Á Szentesia, J Harmatosa, A Jüttnera, P Gajowniczekc, and S Kozdrowskic. On ospf related network optimisation problems. In *Proc. IFIP ATM IP*, 2000.

[87] Patrick Prosser. Mac-cbj: maintaining arc consistency with con ict-directed backjumping. *Submitted to ECAI-96*, 1995.

[88] Jean-François Puget. Set constraints and cardinality operator: Application to symmetrical combinatorial problems. In *Third Workshop on Constraint Logic Programming–WCLP93*, 1993.

[89] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. *Principles and Practice of Constraint Programming-CP 2006*, pages 751–755, 2006.

[90] P Quinn and T Nadeau. Service function chaining problem statement. *draft-ietf-sfc-problem-statement-07 (work in progress)*, 2014.

[91] B. Quoitin, V. Van den Schrieck, P. Francois, and O. Bonaventure. IGen: Generation of router-level Internet topologies through network design heuristics. In *ITC*, 2009.

[92] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN*, 2013.

[93] S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[94] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier Science, 2006.

[95] Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *SIGCOMM Comput. Commun. Rev.*, 35(5):93–96, October 2005.

[96] Pierre Schaus and Renaud Hartert. Multi-objective large neighborhood search. In *Principles and Practice of Constraint Programming*, pages 611–627. Springer, 2013.

[97] Joseph Scott. *Other things besides number: Abstraction, constraint propagation, and string variable types*. PhD thesis, Acta Universitatis Upsaliensis, 2016.

[98] Joseph D Scott, Pierre Flener, and Justin Pearson. Constraint solving on bounded string variables. In *CPAIOR*, pages 375–392, 2015.

[99] Meinolf Sellmann, Thorsten Gellermann, and Robert Wright. Cost-based filtering for shorter path constraints. *Constraints*, 12(2):207–238, 2007.

[100] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *Principles and Practice of Constraint ProgrammingâĂŤCP98*, pages 417–431, 1998.

[101] Henk Smit and Tony Li. Intermediate system to intermediate system (is-is) extensions for traffic engineering (te). 2004.

[102] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1), 2004.

[103] Peter J Stuckey. Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 5–9. Springer, 2010.

[104] Cisco Systems. Cisco visual networking index: Forecast and methodology 2016-2021, 2017.

[105] Renata Teixeira, Aman Shaikh, Tim Griffin, and Geoffrey M. Voelker. Network sensitivity to hot-potato disruptions. In *Proc. SIGCOMM*, 2004.

[106] Dave Thaler and C Hopps. Multipath issues in unicast and multicast next-hop selection. Technical report, RFC 2991, November, 2000.

[107] Olivier Tilmans and Stefano Vissicchio. IGP-as-a-Backup for Robust SDN Networks. In *CNSM*, 2014.

[108] Hakan Umit. *Techniques and tools for intra-domain traffic engineering*. PhD thesis, UCL-Université Catholique de Louvain, 2009.

[109] Laurent Vanbever, Stefano Vissicchio, Luca Cittadini, and Olivier Bonaventure. When the cure is worse than the disease: The impact of graceful IGP operations on BGP. In *Proc. INFOCOM*, 2013.

[110] Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *Integration of AI and OR Techniques in Constraint Programming*, pages 437–453. Springer, 2015.

[111] Christos Voudouris, Edward PK Tsang, and Abdullah Alsheddy. Guided local search. In *Handbook of metaheuristics*, pages 321–361. Springer, 2010.

[112] Patrick Wendell and Michael J. Freedman. Going viral: Flash crowds in an open cdn. In *Proc. IMC*, 2011.

[113] Laurence A Wolsey. *Integer programming*. Wiley, 1998.

[114] S. Yasukawa, A. Farrel, and O. Komolafe. An Analysis of Scaling Issues in MPLS-TE Core Networks. RFC 5439, 2009.

[115] Yue Kwen Justin Yip. *The length-lex representation for constraint programming over sets*. PhD thesis, Ph. D. thesis, Brown University, 2011.