Solving Segment Routing Problems with Hybrid Constraint Programming Techniques

Renaud Hartert, Pierre Schaus, Stefano Vissicchio, and Olivier Bonaventure

UCLouvain, ICTEAM,
Place Sainte Barbe 2,
1348 Louvain-la-Neuve, Belgium
`firstname.lastname@uclouvain.be`

Abstract. Segment routing is an emerging network technology that exploits the existence of several paths between a source and a destination to spread the traffic in a simple and elegant way. The major commercial network vendors already support segment routing, and several Internet actors are ready to use segment routing in their network. Unfortunately, by changing the way paths are computed, segment routing poses new optimization problems which cannot be addressed with previous research contributions. In this paper, we propose a new hybrid constraint programming framework to solve traffic engineering problems in segment routing. We introduce a new representation of path variables which can be seen as a lightweight relaxation of usual representations. We show how to define and implement fast propagators on these new variables while reducing the memory impact of classical traffic engineering models. The efficiency of our approach is confirmed by experiments on real and artificial networks of big Internet actors.

Keywords: traffic engineering, segment routing, constraint programming, large neighborhood search.

1 Introduction

During the last decades, the Internet has quickly evolved from a small network mainly used to exchange emails to a large scale critical infrastructure responsible of significant services including social networks, video streaming, and cloud computing. Simultaneously, Internet Service Providers have faced increasing requirements in terms of quality of service to provide to their end-users, e.g., low delays and high bandwidth. For this reason, controlling the paths followed by traffic has become an increasingly critical challenge for network operators [22] – especially those managing large networks. Traffic Engineering – a field at the intersection of networking, mathematics, and operational research – aims at optimizing network traffic distribution. Among its main objectives, avoiding link overload is one of the most important as it leads to drop of network reliability, e.g., loss of packets and increasing delays [1]. New traffic engineering objectives recently emerged [29]. For instance, a network operator may want specific demands to get through different sequences of network services, e.g., suspect traffic through a battery of firewalls and high-priority traffic via load-balancer and on low-delay paths [13].

Segment Routing (SR) [12] has been recently proposed to cope with those challenges. It is an emerging network architecture that provides enhanced packet forwarding capabilities while keeping a low configuration impact on networks. Segment Routing is both an evolution of MPLS (MultiProtocol Label Switching) [23] and of IPv6 [31]. Many actors of the network industry support segment routing and several internet service providers will implement segment routing to manage their networks [12,13,14,28]. All-in-one, segment routing seems to be a promising technology to solve traffic engineering problems.

The basic idea of Segment Routing is to prepend packets with a stack of labels, called segments, contained in a segment routing header. A segment represents an instruction. In this work, we focus on node segments that can be used to define paths in a weighted graph that represents the network topology. A node segment contains the unique label of the next router to reach. When such a router is reached, the current node segment is popped and the packet is sent to the router referenced by the next segment and so on. Note that segment routing exploits all the equal cost shortest-paths to reach a given destination. Such equal cost shortest-paths – called Equal-Cost Multi-Paths (ECMP) paths – are extremely frequent in network architectures and it is not rare to see as much as 128 different shortest-paths between a source and a destination within a single network [13]. Fig. 1 illustrates the use of two node segments to define a segment routing path from router s to router t. Note that the use of segments provides extreme flexibility in the path selection for different demands.

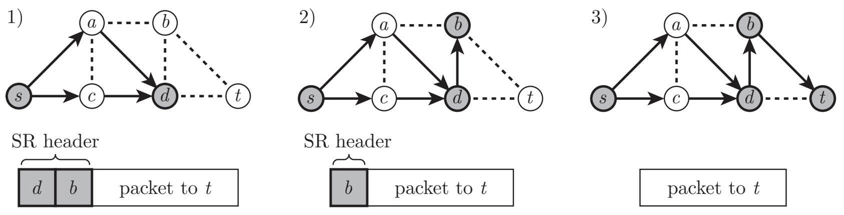

Fig. 1. A segment routing header with two segments prepended to a packet. First, the packet is sent to router d using the ECMP path from s to d (assuming unary link costs). Then, the packet is sent to the next label b following the ECMP path from d to b. Finally, all the segments have been processed and the original packet is sent to its destination t using the ECMP path from b to t.

Unfortunately, optimization approaches proposed in the past years do not consider the enhanced forwarding capabilities of segment routing [12]. Indeed, they all incur two major limitations. First, they typically focus on the basic problem of avoiding network congestion while not considering specific additional requirements [13,29]. Second, all past work assumes network technologies different from segment routing. For example, one of the most successful approaches also used in commercial traffic engineering tools is a tabu search algorithm [18] proposed by Fortz and Thorup in [15]. However, this method is based on the assumption that the network paths are computed as shortest paths in the network topology, hence it simply cannot be used to optimize segment routing networks. Similar considerations also apply to optimization frameworks used for more flexible protocols, like RSVP-TE [11,20].

In this work, we focus on the traffic placement problem using segment routing. Precisely, this problem consists in finding an SR-path for each demand such that the usage of each link is minimized while respecting a set of side constraints on the demands and the network (§2). We target large networks, like those of Internet Service Providers, hence both performance and scalability of the optimization techniques are crucial requirements, in addition to good quality of the solution. We note that classical constraint programming encoding of the problems generally lead to expensive memory consumption that cannot be sustained for large networks (several hundreds of nodes) (§3.1). We thus propose a data structure to encode the domain of our demands that is dedicated to this problem (§3.2 and §3.3). This encoding has the advantage of reducing the memory consumption of classical constraint programming from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3)$. We describe specific and light propagators for constraints defined on top of this new representation (§4). Our results are finally evaluated on synthetic and real topologies (§5 and §6). They highlight that constraint programming and large neighborhood search is a winning combination for segment routing traffic engineering.

2 The General Segment Routing Problem

Let us introduce some notations and definitions. A network is a strongly connected directed graph that consists of a set of nodes $\mathbf{N}$ (i.e. the routers) and a set of edges $\mathbf{E}$ (i.e. the links). An edge $e \in \mathbf{E}$ can be represented as the pair (u, v) where $u \in \mathbf{N}$ is the source of the edge and $v \in \mathbf{N}$ is its destination. The capacity of an edge is denoted $\texttt{capa}(e) \in \mathbb{N}$. For every demand d in the set $\mathbf{D}$, we have an origin $\texttt{src}(d) \in \mathbf{N}$, a destination $\texttt{dest}(d) \in \mathbf{N}$, and a bandwidth requirement $\texttt{bw}(d) \in \mathbb{N}$. We now introduce the notions of *forwarding graph* and *segment routing path.*

Definition 1 (Forwarding Graph). *A forwarding graph describes a flow between a pair of nodes in the network. Formally, a forwarding graph $FG(s, t)$ is a non-empty directed acyclic graph rooted in $s \in \mathbf{N}$ that converges towards $t \in \mathbf{N}$ and such that $s \neq t$.*

Definition 2 (Flow Function). *A forwarding graph $FG(s, t)$ is associated with a flow function $\texttt{flow}_{(s,t)}(e, b) \rightarrow \mathbb{N}$ that returns the amount of the bandwidth $b \in \mathbb{N}$ received at node s that is forwarded to node t through edge $e \in \mathbf{E}$ by the forwarding graph. Particularly, flow functions respect the equal spreading mechanism of ECMP paths. That is, each node of the forwarding graph $FG(s, t)$ splits its incoming traffic equally on all its outgoing edge contained in $FG(s, t)$. Flow functions thus respect the flow conservation constraints.*

Fig. 2 illustrates the flow function of some ECMP paths.

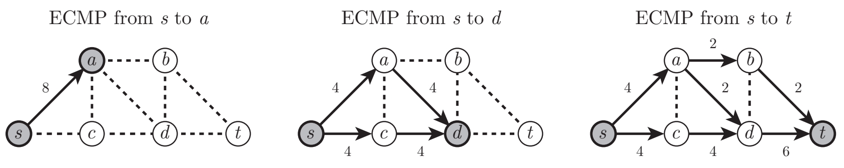

Fig. 2. Three different Equal-Cost Multi-Paths used to transfer 8 units of bandwidth on a network with unary link costs. Each router splits its incoming traffic equally on its outgoing edges. In this context, the value of $\texttt{flow}_{(s,a)}((a,d),8)$ is 0, the value of $\texttt{flow}_{(s,d)}((a,d),8)$ is 4, and the value of $\texttt{flow}_{(s,t)}((a,d),8)$ is 2.

Definition 3 (Segment Routing Path). *A Segment Routing path (SR-path) from $s \in \mathbf{N}$ to $t \in \mathbf{N}$ is a non-empty sequence of forwarding graphs*

$$FG(s, v_1), FG(v_1, v_2), \dots, FG(v_i, v_{i+1}), \dots, FG(v_k, v_{k-1}), FG(v_k, t)$$

denoted $(s, v_1, \dots, v_k, t)$ and such that the destination of a forwarding graph is the the source of its successor in the sequence. Also, the source of the first forwarding graph and the destination of the last forwarding graph respectively correspond to the source and the destination of the SR-path.

Segment routing paths are illustrated in Fig. 3.

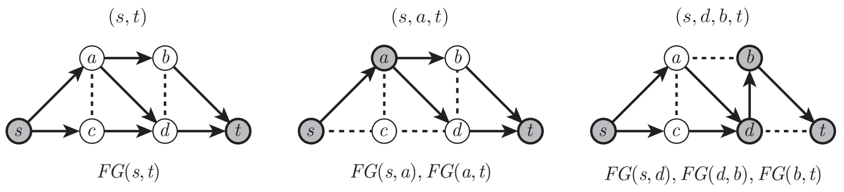

Fig. 3. Three different SR-paths based on the forwarding graphs of Fig. 2. An SR-path is represented by the sequence of nodes (top) corresponding to the extremities of its forwarding graphs (bottom).

We can now formalize the problem we want to solve. Let **FG** be a set of forwarding graphs on a network such that there is at most one forwarding graph for each pair of nodes $(u, v) \in \mathbf{N} \times \mathbf{N}$ with $u \neq v$. Let $SR(d)$ be the SR-path of demand $d \in \mathbf{D}$ using the forwarding graph in **FG**. The General Segment Routing Problem (GSRP) consists in finding a valid SR-path for each demand d such that the capacity of each edge is not exceeded

$$\forall e \in \mathbf{E} : \sum_{d \in \mathbf{D}} \sum_{FG(i,j) \in SR(d)} \texttt{flow}_{(i,j)}(e, \texttt{bw}(d)) \leq \texttt{capa}(e) \tag{1}$$

and such that a set of constraints on the SR-paths and the network is respected.

Proposition 1. *The general segment routing problem is $\mathcal{NP}$-Hard.*

Proof. The Partition problem [6] is an $\mathcal{NP}$-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 GSRP depicted in Fig. 4 with all edge capacities fixed to $\sum_{i=1}^{n} c_i/2$. First, consider n demands $d_1, \ldots, d_n$ from node s to node t such that $\mathtt{bw}(d_i) = c_i$. Then, consider that forwarding graphs are defined such that there are only two possible SR-paths from node s to node t (see Fig. 4). Finding a valid SR-path for each demand 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 $\overline{A}$. $\square$

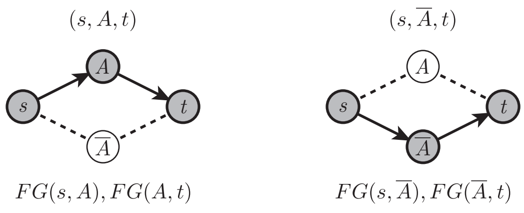

Fig. 4. The $\mathcal{NP}$-complete Partition problem can be encoded as an instance of the GSRP on this network. Forwarding graphs are defined such that there are only two possible SR-paths from node s to node t.

Practical considerations. Due to hardware limitations, segment routing headers usually contain no more than $k \in [4, 16]$ segments which limits the number of forwarding graphs to be contained in an SR-path to k. Furthermore, an SR-path should not include a loop, i.e., packets should never pass twice on the same link in the same direction.

3 Segment Routing Path Variables

This section is dedicated to the ad-hoc data structure we use to model the decision variables of the GSRP. Specialized domain representations are not new in constraint programming [34] and several data structures have already been proposed to represent abstractions such as intervals, sets, and graphs [8,10,16,17].

3.1 Shortcomings of Classical Path Representations

Observe that an SR-path from node s to node t can be seen as a simple path from s to t in a complete graph on $\mathbf{N}$ where each link (u, v) corresponds to the forwarding graph $FG(u, v)$. With this consideration in mind, let us motivate the need of an alternative representation by reviewing classical ways to model path variables, i.e., path-based, link-based, and node-based representations [39].

In *path-based* representations, a single variable is associated to each path. The domain of this variable thus contains all the possible paths to be assigned to the variable. This representation is usual in column generation frameworks [2]. In the context of the GSRP, complete path-based representations have an impracticable memory cost of $\Theta(|\mathbf{D}||\mathbf{N}|^k)$ where k is the maximal length of the SR-paths.

Link-based representations are surely the most common way to model path variables [25,42]. The idea is to associate a boolean variable $\mathbf{x}(d, e)$ to each demand $d \in \mathbf{D}$ and edge $e \in \mathbf{E}$. The boolean variable is set to true if edge e is part of the path, false otherwise. Hence, modeling SR-paths with link-based representations requires $\Theta(|\mathbf{D}||\mathbf{N}|^2)$ boolean variables.

Basically, a *node-based* representation models a path as a sequence of visited nodes [33]. This could be easily modeled using a discrete variable for each node that represents the successor of this node. The memory impact of such representation is $\Theta(|\mathbf{D}||\mathbf{N}|^2)$.

As mentioned above, memory and computational costs are of practical importance as we want to solve the GSRP on networks containing several hundreds of nodes with tens of thousands of demands. In this context, even link-based and node-based representations suffer from important shortcomings in both aspects.

3.2 Segment Routing Path Variables

Given the impossibility of using classic representations, we propose a new type of structured domain which we call *SR-path variable*. An SR-path variable represents a sequence of visited nodes[1] from the source of the path to its destination. The domain representation of this variable contains (see Fig. 5):

1. A prefix from the source to the destination that represents the sequence of already *visited* nodes ;
2. The set of possible nodes, called *candidates*, to append to the prefix of already visited nodes.

Basically, the domain of an SR-path variable is the set of all the possible extensions of the already visited nodes followed directly by a node contained in the set of candidates and finishing at its destination node. An SR-path variable can thus be seen as a relaxation of classic node-representations. An SR-path variable is assigned when the last visited node is the path destination. It is considered as invalid if it is unassigned and if its set of candidates is empty.

The particularity of SR-path variables is that the filtering of the domain is limited to the direct successor of the last visited node, i.e., the set of candidates

[1] Recall that an SR-path is a sequence of forwarding graphs denoted by a sequence of nodes (see Definition 3).

(see Fig. 5). Hence, no assumption can be made on the part of the sequence that follows the prefix of already visited nodes and the set of candidates. As a side effect, visiting a new node c automatically generates a new set of candidates as assumptions on the successor of c were impossible until now. It is then the responsibility of constraints to reduce this new set of candidates.

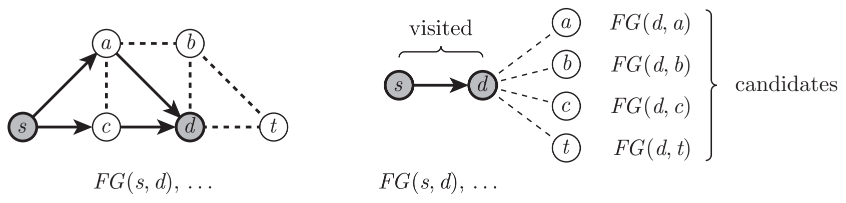

Fig. 5. A partial SR-path (left) and its corresponding SR-path variable (right). The SR-path variable maintains the partial sequence of visited nodes from the source s to the destination t and a set of candidates. Candidates represent the possible direct successors of the last visited node.

An SR-path variable S supports the following operations:

- `visited`(S): returns the sequence of visited nodes.
- `candidates`(S): returns the set of candidates.
- `visit`(S, c): appends c to the sequence of visited nodes.
- `remove`(S, c): removes c from the current set of candidates.
- `position`(S, c): returns the position of c in the sequence of visited nodes.
- `isAssigned`(S): returns true iff the path has reached its destination.
- `length`(S): returns an integer variable representing the length of the path.
- `src`(S), `dest`(S), `last`(S): returns the source node, the destination node, and the last visited node respectively.

3.3 Implementation

We propose to use array-based sparse-sets [5] to implement the internal structure of SR-path variables. Our data structure is similar to the one proposed to implement set variables in [8]. The sparse-set-based data structure relies on two arrays of $|\mathbf{N}|$ elements, `nodes` and `map`, and two integers `V` and `R`. The `V` first nodes in `nodes` correspond to the sequence of visited nodes. The nodes at positions $[\texttt{V}, \ldots, \texttt{R}[$ in `nodes` form the set of candidates. The `map` array maps each node to its position in `nodes`. Fig. 6 illustrates this data structure and its corresponding partial SR-path.

The sparse-set-based implementation of SR-path variables offers several advantages. First, it is linear in the number of nodes since it only relies on two arrays and two integers. Also, it implicitly enforces the `AllDifferent` constraint on the sequence of visited nodes. Finally, it allows optimal computational complexity for all the operations presented in Table 1. Many of these operations

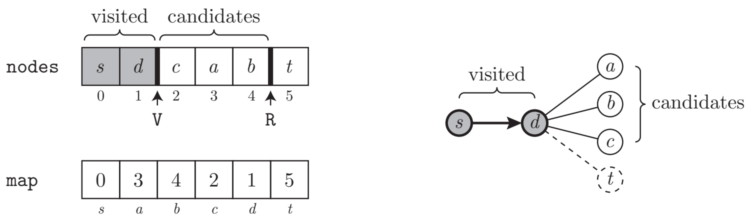

Fig. 6. An incremental sequence variable implemented in a sparse set. The sequence of visited nodes is s, d the set of candidates is $\{a, b, c\}$. Node t is not a valid candidate.

can be implemented trivially by comparing node positions to the value of `V` and `R`. However, the `visit` and `remove` operations require more sophisticated manipulations of the data structure.

Iterate on the sequence of visited nodes	$\Theta(\lvert visited \rvert)$
Iterate on the set of candidates	$\Theta(\lvert candidates \rvert)$
Iterate on the set of removed candidates	$\Theta(\lvert removed \rvert)$
Returns the last visited node in the sequence	$\mathcal{O}(1)$
Test if a node has been visited, removed, or is still a candidate	$\mathcal{O}(1)$
Returns the position of a visited node in the sequence	$\mathcal{O}(1)$
Remove a candidate	$\mathcal{O}(1)$
Visit a candidate	$\mathcal{O}(1)$

Table 1. A sparse-set-based implementation offers optimal time-complexities for several operations.

Visit a new node. Each time a node is visited, it swaps its position in `nodes` with the node at position `V`. Then, the value of `V` is incremented to append the visited node to the sequence of visited nodes. Finally, the value of `R` is set to $|\mathbf{N}|$ to restore the set of candidates to all the non-visited nodes. The `visit` operation is illustrated in Fig. 7. Observe that the sequence of visited nodes keeps its chronological order.

Remove a candidate. The remove operation is performed in a similar way as the visit operation. First, the removed node swaps its positions in `nodes` with the node at position `R`. Then, the value of `R` is decremented to exclude the removed node from the set of candidates. Fig. 8 illustrates this operation.

Backtracking is achieved in $\mathcal{O}(1)$. Indeed, we only need to trail the value of `V` to restore the previous sequence of visited nodes. Unfortunately, this efficient backtracking mechanism cannot restore the previous set of candidates. Nevertheless, this problem could be addressed by one of the following ways:

- The set of candidates could be recomputed on backtrack ;
- Changes in the set of candidates could be trailed during search ;
- Search could be restricted to visit all valid candidates with n-ary branching.

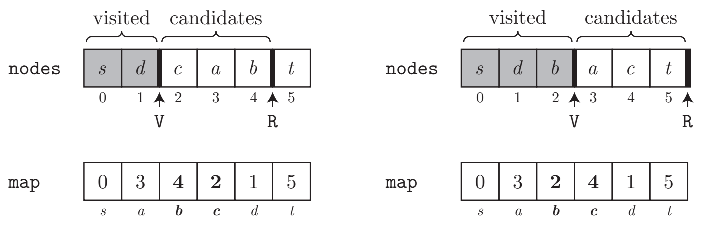

Fig. 7. State of the internal structure before (left) and after (right) visiting node b. First, node b exchanges its position with the node at position `V` in `nodes`, i.e., c. Then, `V` is incremented and `R` is set to $|\mathbf{N}|$.

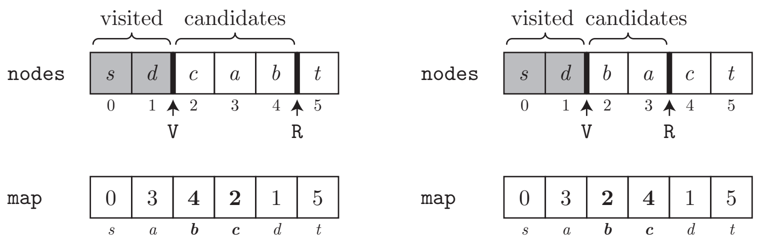

Fig. 8. State of the internal structure before (left) and after (right) removing node c. First, node c exchanges its position with the node at position `R` in `nodes`, i.e., b. Then, `R` is decremented.

We chose to apply the third solution as visiting a new node automatically restores the set of candidates to all the non-visited nodes (by updating the value of `R` to $|\mathbf{N}|$). This search mechanism thus allows us to keep backtracking in constant time since previous sets of candidates do not need to be recovered.

4 Constraints on SR-Path Variables

We model the GSRP by associating an SR-path variable to each demand in $\mathbf{D}$. These variables are the decisions variables of the problem. Also, each link of the network $e \in \mathbf{E}$ is associated with an integer variable `load`(e) that represents the total load of this link, i.e., the total amount of traffic routed through e. We now present some constraints developed on top of the SR-path variables. These constraints are designed to meet requirements of network operators. Constraints on SR-path variables are awaken if one of both following events occurs in a variable of their scope:

- `visitEvent`(S, a, b) : tells the constraints that node b has been visited just after node a in the SR-path variable S ;
- `removeEvent`(S, a, b) : tells the constraints that node b is not a valid candidate to be visited after node a in the SR-path variable S.

We implemented a propagation algorithm specialized for these events in an AC5-like framework [41].

4.1 The `Channeling` constraint

The role of the `Channeling` constraint is to ensure consistency between an SR-path variable S and the load of each link in the network. The `Channeling` constraint maintains the following property:

$$\begin{aligned}&\forall c \in \texttt{candidates}(S), \forall e \in FG(\texttt{last}(S), c): \\ &\texttt{load}(e) + \texttt{flow}_{(\texttt{last}(S),c)}(e, \texttt{bw}(d)) \leq \texttt{capa}(e)\end{aligned} \tag{2}$$

The filtering of the `Channeling` constraint is enforced using two filtering procedures. The first filtering procedure is triggered each time a new candidate is visited by an SR-path variable to adjust the load variable of all the links traversed by the visited forwarding graph. Then, the second filtering procedure is called to reduce the set of candidates by removing forwarding graphs which cannot accommodate demand d due to the insufficient remaining bandwidth of their links. This second filtering procedure is also called each time the load variable of a link is updated.

The `Channeling` constraint relies intensively on forwarding graphs and flow functions. In our implementation, we chose to precompute these flow functions for fast propagation of the `Channeling` constraint. Such data structures may have an important impact of $\mathcal{O}(|\mathbf{N}|^2|\mathbf{E}|)$ in memory. Fortunately, this cost can be reduced to $\mathcal{O}(|\mathbf{N}||\mathbf{E}|)$ by exploiting the shortest-paths DAG[2] returned by Dijkstra's algorithm.[3] Indeed, we can test the presence of an edge (u, v) in a shortest-path from s to t using the transitive property of shortest-paths as follows:

$$\texttt{dist}(s, u) + \texttt{cost}(u, v) + \texttt{dist}(v, t) = \texttt{dist}(s, t). \tag{3}$$

This property allows one to use shortest-paths DAGs to recompute flow functions and forwarding graphs in $\mathcal{O}(|\mathbf{E}|)$ when required.

4.2 The `Length` constraint

The `Length` constraint limits the number of forwarding graphs contained in an SR-path variable S:

$$|\texttt{visited}(S)| = \texttt{length}(S). \tag{4}$$

This constraint aims to respect real network hardware limitations in terms of maximum number of segments to be appended to packets [12]. We designed a simple filtering procedure for the `Length` constraint by comparing the number

[2] Shortest-paths DAGs can be easily computed by maintaining several shortest-paths to a given node in Dijkstra's algorithm.

[3] Recall that we make the assumption that forwarding graphs correspond to ECMP paths.

of visited nodes to the bounds of the length variable. Let $\min(\texttt{length}(S))$ and $\max(\texttt{length}(S))$ respectively be the lower bound and the upper bound of the length variable of S. Changes on the bounds of this variable trigger both following filtering procedures :

- If $|\texttt{visited}(S)| < \min(\texttt{length}(S)) - 1$, we know that the destination of the SR-path is not a valid candidate as visiting it next will result in a path that is smaller than the lower bound of its length variable ;
- If $|\texttt{visited}(S)| = \max(\texttt{length}(S)) - 1$, the destination of the SR-path must be visited next to respect the upper bound of its length variable.

The `Length` constraint is also awakened when a new node is visited. In this case, the filtering procedure first checks if the visited node is the destination of the path. If it is the case, the length variable is assigned to $|\texttt{visited}(S)|$. Otherwise, the lower bound of the length variable is updated to be strictly greater than $|\texttt{visited}(S)|$ as the path destination still has to be visited.

4.3 The `ServiceChaining` constraint

Many operators have lately shown interest for *service chaining*, i.e., the ability to force a demand to pass through a sequence of services [29]. The aim of the `ServiceChaining` constraint is to force an SR-path variable S to traverse a particular sequence of nodes described by a sequence of services. Each service is traversed by visiting one node in a given set, i.e., the set of nodes providing the corresponding service in the network. Let $services = (service_1, \ldots, service_k)$ denote the sequences of services to be traversed. The filtering rule of the `ServiceChaining` constraint enforces the sequence of visited nodes to contain a non-necessarily contiguous subsequence of services:

$$\begin{aligned} \texttt{visited}(S) &= \ldots, s_1, \ldots, s_2, \ldots, s_k, \ldots \\ \forall i \in \{1, \ldots, k\} &: s_i \in service_i \end{aligned} \tag{5}$$

The filtering of the `ServiceChaining` constraint uses similar procedures as those used by the `Length` constraint.

4.4 The `DAG` constraint

The `DAG` constraint prevents cycles in the network. A propagator for this constraint has already been proposed in [10]. The idea behind this propagator is to forbid the addition of any link that will result in a cycle in the transitive closure of the set of edges visited by an SR-path variables. In the context of the GSRP, the filtering of the `DAG` constraint can be strengthened by taking in consideration that an SR-path is a sequence of forwarding graphs which are acyclic by definition. First, we know that the source (resp. destination) of an SR-path variable S has no incoming (resp. outgoing) traffic:

$$\nexists FG(u,v) \in S : \texttt{src}(S) \in \texttt{nodes}(FG(u,v)) \tag{6}$$

$$\nexists FG(u,v) \in S \,:\, \texttt{dest}(S) \in \texttt{nodes}(FG(u,v)) \tag{7}$$

Additional filtering is achieved using the definition of SR-paths and the following proposition.

Proposition 2. *Let c be a node traversed by two forwarding graphs such that c is not one of the extremities of those forwarding graphs. Then, there is no acyclic SR-path that visits both forwarding graph.*

Proof. Let c be a node traversed by $FG(i,j)$ and $FG(u,v)$ such that $c \notin \{i,j,u,v\}$ and that $FG(i,j)$ and $FG(u,v)$ are part of the same SR-path. As SR-paths are defined as a sequence of forwarding graphs, we know that $FG(i,j)$ is traversed before $FG(u,v)$ or that $FG(u,v)$ is traversed before $FG(i,j)$. Let us consider that $FG(i,j)$ is traversed before $FG(u,v)$. In this case, we know that there is a path from node j to node u. According to the definition of forwarding graphs, we also know that there is a path from c to j and from u to c. Therefore, there is a path from c to c which is a cycle. The remaining part of the proof is done symmetrically. □

The `DAG` constraint thus enforce the following redundant property:

$$\begin{aligned} &\forall \{FG(u,v), FG(i,j)\} \subseteq S \,: \\ &\texttt{nodes}(FG(u,v)) \cap \texttt{nodes}(FG(i,j)) \subseteq \{u,v\}. \end{aligned} \tag{8}$$

The filtering procedures of the `DAG` constraint are triggered each time an SR-path variable visits a new node. Note that the set of nodes traversed by the visited forwarding graph of an SR-path variable – necessary to implement these filtering procedures – can be maintained incrementally with a memory cost of $\mathcal{O}(|\mathbf{D}||\mathbf{N}|)$ or it can be recomputed when required with a time complexity of $\mathcal{O}(|\mathbf{N}|)$.

4.5 The `MaxCost` constraint

Often, service level agreements imply that some network demands must be routed on paths with specifics characteristics, e.g., low delays. Such requirements can easily be enforced using the `MaxCost` constraint that ensures that the total cost of an SR-path does not exceed a maximum cost C. To achieve this, we associate a positive cost to each forwarding graph $FG(u,v)$. Let $\texttt{cost}(FG(u,v)) \in \mathbb{N}$ denote this cost and $\texttt{minCost}(s,t)$ denote the minimum cost of reaching node t from node s with an SR-path of unlimited length. Such minimum costs could be easily precomputed using shortest-paths algorithms and only require a space complexity of $\Theta(|\mathbf{N}|^2)$ to be stored. The filtering rule of the `MaxCost` constraint (based on the transitive property of shortest-paths) enforces:

$$\begin{aligned} &\forall c \in \texttt{candidates}(S) \,: \\ &\texttt{cost}(\texttt{visited}(S)) + \texttt{cost}(\texttt{last}(S), c) + \texttt{minCost}(c, \texttt{dest}(S)) \leq C \end{aligned} \tag{9}$$

where $\texttt{cost}(\texttt{visited}(S))$ is the total cost of already visited forwarding graphs.

Substantial additional filtering could be added by specializing the `minCost` function to also consider the length variable of an SR-path variable. This would require to pre-compute the all pair shortest-distance for all the k possible lengths of the path with, for instance, labeling algorithms [3,9].

5 Hybrid Optimization

Finding a first feasible solution of the GSRP can be a difficult task. In this context, it is often more efficient to relax the capacity constraint of each link and to minimize the load of the maximum loaded links until respecting the original capacity constraints.

Frameworks such as Large Neighborhood Search (LNS) [38] have been shown to be very efficient to optimize large scale industrial problems in many domains [4,21,26,27,32,37]. The idea behind LNS is to iteratively improve a best-so-far solution by relaxing some part of this solution. In the context of GSRP, this could be easily achieved by removing demands that seem inefficiently placed and to replace them in the network to improve the objective function. The selected set of removed demands defines the neighborhood to be explored by a branch-and-bound constraint programming search.

Instead of dealing with all the load variables with a unique aggregation function (e.g. the maximum load), we designed a specific hybrid optimization scheme to make the optimization more aggressive. At each iteration, we force a randomly chosen most loaded link to strictly decrease its load. The load of the remaining links are allowed to be degraded under the following conditions:

- The load of the most loaded links cannot increase;
- The load of the remaining links are allowed to increase but must remain under the maximum load, i.e., the set of the most loaded links cannot increase;
- The load of the non-saturated links must remain under their link capacity to not increase the number of saturated links.

This optimization framework can be seen as particular instantiation of the variable-objective large neighborhood search framework [36].

Neighborhoods to be explored by constraint programming are generated using the following procedure [38]:

1. Let $\mathbf{D}^{\max}$ be the set of demands routed through the most loaded links;
2. While fewer than k demands have been selected:
   (a) Let A be the array of the not yet selected demands in $\mathbf{D}^{\max}$ sorted by non-increasing bandwidth;
   (b) Generate a random number $r \in [0, 1[$;
   (c) Select the $\lfloor r^{\alpha} |A| \rfloor$ demand in array A.

The parameter $\alpha \in [1, \infty[$ is used to control the diversification of the selection procedure. If $\alpha = 1$ all demands have the same chance to be selected. If $\alpha = \infty$ the k demands with the highest bandwidth are always selected. The sub-problem generated by removing the selected demands is then explored by constraint programming with a CSPF-like (Constrained Shortest Path First) heuristic [7]:

1. Select the unassigned demand with the largest bandwidth;
2. Try to extend the SR-path variable of this demand as follows:
   (a) If the demand destination is a valid candidate, visit the destination to assign the SR-path variable;

(b) Otherwise, visit all the candidates sorted by non-decreasing order of their impact on the load of the most loaded links.

3. If the SR-path variable has no valid candidate, backtrack and reconsider the previous visit.

We call this heuristic an *assign-first* heuristic because it tends to assign SR-path variables as soon as possible.

6 Experiments and Results

We performed many experiments on real and synthetic topologies and demands. Real topologies are meant to assess the efficiency of our approach on real practical situations. Conversely, synthetic topologies were used to measure the behavior of our approach on complex networks. The data set is summarized in Table 2. Eight of these topologies are large real-world ISP networks and from the publicly available Rocket Fuel topologies [40]. Synthetic topologies were generated using the Delaunay's triangulation algorithm implemented in IGen [30]. Notice that these synthetic topologies are highly connected from a networking point of view, i.e., they have an average node degree of 5.72.

Whenever available – that is for topology RealF, RealG, and RealH – we used the real sets of demands measured by operators. In the other cases, we generated demands with similar characteristics as the ones in our real data sets as described in [35]. Finally, we proportionally scaled up all the demands until a maximum link usage of approximatively 120% was reached.[4] This allowed us to consider critical situations in our analysis.

We measured the efficiency of our approach by minimizing the maximum link load on each network presented in Table 2. Our experiments were performed on top of the open-source OscaR Solver [24] with a MacBook Pro configured with 2.6 GHz Intel CPU and 16GB of memory, using a 64-Bit HotSpot^{TM} JVM 1.8.

Topology	**|N|**	**|E|**	**|D|**	**Before**	**Relaxation**	**Max load**
RealA	79	294	6160	120%	68%	72%
RealB	87	322	7527	142%	72%	82%
RealC	104	302	10695	130%	**86**%	**86**%
RealD	141	418	19740	140%	**96**%	**96**%
RealE	156	718	23682	121%	76%	78%
RealF	161	656	25486	117%	65%	74%
RealG	198	744	29301	162%	37%	74%
RealH	315	1944	96057	124%	**76**%	**76**%
SynthA	50	276	2449	103%	69%	71%
SynthB	100	572	9817	100%	59%	75%
SynthC	200	1044	37881	135%	53%	84%

Table 2. Topologies and results.

[4] In a solution with all demands routed through their ECMP path.

For each instance, we computed a linear programming lower bound on the maximum load by solving the linear multi-commodity flow problem on the same topology and demands with Gurobi 6.0 [19] (see column "Relaxation" of Table 2). We also show the initial maximum load of the network if no segment is used to redirect demands through the network (see column "Before" of Table 2). The results provided by our optimization framework after 60 seconds of computations are presented in the last column of Table 2. Note that the linear relaxation cannot be reached in real networks because it assumes that flows can be arbitrarily split, a function that does not exist in current routers. In practice, even if uneven splitting became possible in hardware, it would be impossible from an operational viewpoint to maintain different split ratio for each flow on each router.

As we see, our approach is able to find close to optimal solutions at the exception of instance RealG. Moreover, the solutions found by our approach are optimal for instances RealC, RealD, and RealH. This is due to the fact that these instances contain links that must be traversed by many demands. Fig. 9 illustrates the changes in link loads to highlight such bottleneck links.

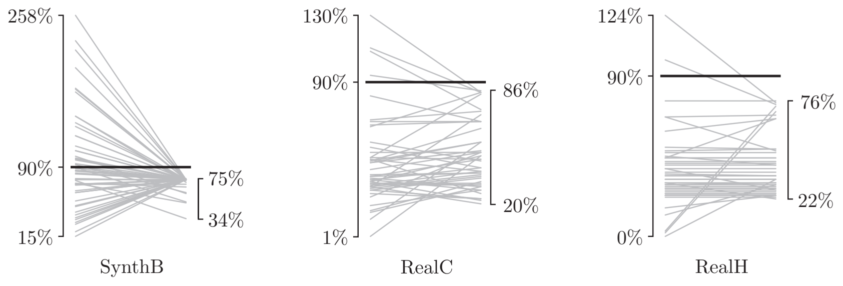

Fig. 9. Load of the 40 most loaded edges before (left) and after (right) optimization on a synthetic topology and two real ones. The load of many links have to be reduced to improve the maximum load on the artificial topologies while only a few have to be considered on the real ones. Real topologies contain more bottleneck links.

7 Conclusion

This paper presented an hybrid framework to solve segment routing problems with constraint programming and large neighborhood search. First, we introduced and formalized this new problem encountered by network operators. Then, we analyzed the shortcomings of classical constraint programming models to solve this problem on large networks. We thus proposed a new domain structure for path variable that we called *SR-path variable*. The particularity of this structure is that it sacrifices tight domain representation for low memory cost and fast domain operations. We explained how to implement common requirements of network operators in terms of constraint on this new variable. The efficiency of our approach was confirmed on large real-world and synthetic topologies.

References

1. D Awduche, A Chiu, A Elwalid, I Widjaja, and X Xiao. Overview and principles of internet traffic engineering–rfc3272. *IETF*, 2002.
2. Cynthia Barnhart, Christopher A Hane, and Pamela H Vance. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. *Operations Research*, 48(2):318–326, 2000.
3. JE Beasley and Nicos Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, 1989.
4. 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.
5. 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.
6. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.
7. Bruce Davie and Yakov Rekhter. *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc., 2000.
8. 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.
9. 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.
10. 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.
11. Gustavo B Figueiredo, Nelson LS da Fonseca, and José Augusto Suruagy Monteiro. A minimum interference routing algorithm. In *ICC*, pages 1942–1947, 2004.
12. C. Filsfils et al. Segment Routing Architecture. Internet draft, draft-filsfils-spring-segment-routing-00, work in progress, 2014.
13. C. Filsfils et al. Segment Routing Use Cases. Internet draft, draft-filsfils-spring-segment-routing-use-cases-00, work in progress, 2014.
14. C. Filsfils et al. Segment Routing with MPLS data plane. Internet draft, draft-filsfils-spring-segment-routing-mpls-01, work in progress, 2014.
15. B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proc. INFOCOM*, March 2000.
16. Christian Frei and Boi Faltings. Resource allocation in networks using abstraction and constraint satisfaction techniques. In *Principles and Practice of Constraint Programming–CP?99*, pages 204–218. Springer, 1999.
17. Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In *ILPS*, volume 94, pages 339–358, 1994.
18. Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.
19. Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2015.
20. Murali Kodialam and TV Lakshman. Minimum interference routing with applications to mpls traffic engineering. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 884–893. IEEE, 2000.

21. J.B. Mairy, Y. Deville, and P. Van Hentenryck. Reinforced adaptive large neighborhood search. In *The Seventeenth International Conference on Principles and Practice of Constraint Programming (CP 2011)*, page 55, 2011.
22. Antonio Nucci and Konstantina Papagiannaki. *Design, Measurement and Management of Large-Scale IP Networks - Bridging the Gap between Theory and Practice.* Cambridge University Press, 2008.
23. RFC3031 ŒTF. Multiprotocol label switching architecture, 2001.
24. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
25. Wided Ouaja and Barry Richards. A hybrid multicommodity routing algorithm for traffic engineering. *Networks*, 43(3):125–140, 2004.
26. 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.
27. Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. *Principles and Practice of Constraint Programming–CP 2004*, pages 468–481, 2004.
28. S. Previdi et al. IPv6 Segment Routing Header (SRH). Internet draft, draft-previdi-6man-segment-routing-header-00, work in progress, 2014.
29. P Quinn and T Nadeau. Service function chaining problem statement. *draft-ietf-sfc-problem-statement-07 (work in progress)*, 2014.
30. 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.
31. S Deering RFC2460 and R Hinden. Internet protocol.
32. 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.
33. Lluís Ros Giralt, Tom Creemers, Evgueni Tourouta, Jordi Riera Colomer, et al. A global constraint model for integrated routeing and scheduling on a transmission network. 2001.
34. Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming.* Elsevier Science, 2006.
35. Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *SIGCOMM Comput. Commun. Rev.*, 35(5):93–96, October 2005.
36. Pierre Schaus. Variable objective large neighborhood search. *Submitted to CP13*, 2013.
37. Pierre Schaus and Renaud Hartert. Multi-Objective Large Neighborhood Search. In *19th International Conference on Principles and Practice of Constraint Programming*, 2013.
38. P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *Principles and Practice of Constraint ProgrammingCP98*, pages 417–431, 1998.
39. Helmut Simonis. Constraint applications in networks. *Handbook of constraint programming*, 2:875–903, 2006.
40. Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1), 2004.
41. Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2):291–321, 1992.

42. Quanshi Xia and Helmut Simonis. Primary/secondary path generation problem: Reformulation, solutions and comparisons. In *Networking-ICN 2005*, pages 611–619. Springer, 2005.