Expect the Unexpected: Sub-Second Optimization for Segment Routing

Steven Gay
UCLouvain, ICTEAM
steven.gay@uclouvain.be

Renaud Hartert
Google, Inc.
rhartert@google.com

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

***Abstract*—In this paper, we study how to perform traffic engineering at an extremely-small time scale with segment routing, addressing a critical need for modern wide area networks.**

Prior work has shown that segment routing enables to better engineer traffic, thanks to its ability to program detours in forwarding paths, at scale. Two main approaches have been explored for traffic engineering with segment routing, respectively based on integer linear programming and constraint programming. However, no previous work deeply investigated how quickly those approaches can react to unexpected traffic changes and failures.

We highlight limitations of existing algorithms, both in terms of required execution time and amount of path changes to be applied. Thus, we propose a new approach, based on local search and focused on the quick re-arrangement of (few) forwarding paths. We describe heuristics for sub-second recomputation of segment-routing paths that comply with requirements on the maximum link load (e.g., for congestion avoidance). Our heuristics enable a prompt answer to sudden criticalities affecting network services and business agreements. Through extensive simulations, we indeed experimentally show that our proposal significantly outperforms previous algorithms in the context of time-constrained optimization, supporting radical traffic changes in few tens of milliseconds for realistic networks.

I. Introduction

The capability to promptly answer to unexpected network dynamics in a very short time is more critical than ever in large (wide area) networks, like inter-datacenter or Internet Service Provider ones. On one hand, new network architectures, as SDN [1], enable networks to run at almost-full capacity, and let links carry more traffic without having to heavily overprovision networks [2], [3]. On the other hand, forwarding optimality can be disrupted by the increasing number of unexpected, sudden traffic surges that change demand volumes and matrices, e.g., due to new popular content, social networks and related flash crowds [4]: those unpredictable traffic fluctuations have to be managed in addition to (and likely more often than) traditionally-considered events like link failures. Globally, running networks closer to their physical limits in a highly-dynamic environment increases the risk to experience congestion at any moment in time.

To answer this need of modern networks, recent works [2], [3] have focused on online traffic-engineering approaches, where network paths are re-optimized periodically, typically every few (e.g., 5) minutes. Those works complement long-lasting efforts to quickly re-optimize traditional networks, e.g., running tunnelling technologies like MPLS [5] or plain shortest-path routing [6]. By running periodically, this approach can only remove congestion after the fixed time period, but does not avoid it completely. This also means that the network performance can be deteriorated (e.g., with traffic lost) for several minutes. A classic alternative is to pre-compute a congestion-free reaction to events, e.g., pre-provisioning backup paths in the case of failure [7], [8] or optimizing against a set of traffic matrices [9]. However, the latter approach is impossible to apply to all possible events, especially the unexpected ones that unpredictably change traffic distributions.

In this paper, we explore the feasibility of a different approach, where a centralized network controller re-optimizes forwarding paths as soon as the utilization of a link increases too much, in consequence of significant traffic changes or unexpected network failures.

Two building blocks are key in this approach. First, a monitoring or alarming primitive to quickly detect unexpected events. Second, a very fast and effective re-optimization of network paths. We focus on the latter building block, as the former can be implemented with the most recent monitoring techniques [10], [11].

In order to achieve a proper reaction to unexpected events, we propose an extremely-fast traffic-engineering algorithm computing a limited set of quickly-implementable path changes to avoid congestion.

To enforce path changes that can be quickly implemented by traditional network equipment, we build upon the increasingly-popular Segment Routing (SR) protocol [12]. SR enables effective centralized control over network paths by configuring ingress routers to enrich traversing packets with information about nodes (or links) to be crossed before reaching their respective destinations.

In contrast to prior SR traffic-engineering contributions [9], [13], we however face additional levels of algorithmic complexity, due to our strong commitment to minimizing execution time and balancing forwarding optimality with quick implementation of path changes (e.g., by minimizing them). This exacerbates the computational complexity of solving already-hard traffic-engineering problems with SR — e.g., minimizing the maximum link utilization with SR is NP-hard [14].

We tackle this algorithmic challenge by relying on Local Search (LS), which combines two advantages. First, LS is an anytime optimization technique, meaning that it always returns

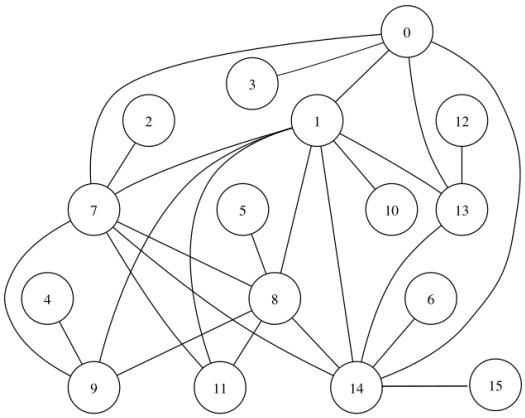


Fig. 1: Topology of a network (Airtel) as reported in Topology Zoo [15]. Links between nodes represent edges in both directions. All edges have the same capacity and unitary weight.

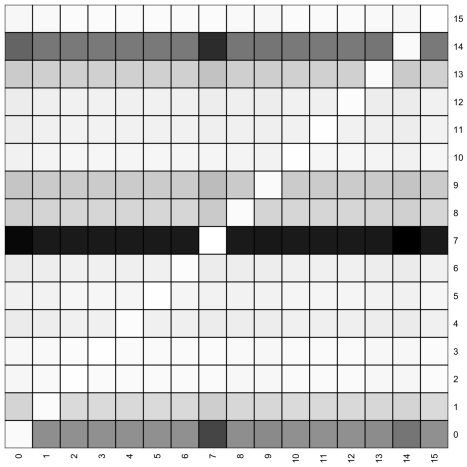


Fig. 2: Representation of the demand matrix used in our experiments on the Airtel network (see Fig. 1). Rows and columns respectively represent demand sources and destinations. The color intensity of every cell is directly proportional to the traffic volume for the corresponding demand.

a solution no matter when the algorithm is stopped — thus fitting our strict time constraints. Second, by considering many small variations of a current solution, LS also tends to find traffic-engineering paths that are structurally close from the initial ones (those currently installed in the network). In other words, LS naturally limits the number of path changes.

Within the LS framework, we use tailored data structures to compute moves much faster than a simulation approach would achieve; further, we design aggressive heuristics that quickly converge to qualitatively-good solutions, rather than aiming to find the global optimum.

In the rest of the paper, we first state our traffic-engineering problem and exemplify the shortcomings of related work for prompt reaction to unexpected events (§II). Then, we describe the design of our LS algorithm (§III). Also, we delve into the data structures and main implementation details that we adopted to improve the speed of our algorithm (§IV). Finally, we evaluate our approach in realistic time-constrained scenarios (§V), and conclude (§VI).

II. Traffic Engineering with Segment Routing

In this section, we introduce our network and routing model, as well as our notation. We then formalize two variants of the traffic-engineering problem that differently express the ability to quickly answer unexpected events. Finally, we describe the limitations of existing techniques to solve both variants.

For illustration, we use the network depicted in Fig. 1 with traffic demands set as in Fig. 2.

A. Model

Network. We define a network as a strongly-connected directed graph with a set of nodes $\mathbf{N}$ and a set of edges $\mathbf{E}$ that respectively represent network routers and the physical links between them. Each edge $e \in \mathbf{E}$ is represented by a pair of nodes (u, v) where u is the source of e and v is its sink. A capacity $capa(e) \in \mathbb{N}$ and a weight $w(e) \in \mathbb{N}$ are associated to every edge e. The capacity of an edge indicates how much traffic that link can carry. The weight of an edge is used by intra-domain routing protocols to decide paths followed by traffic traversing the network (as we detail below). In Fig. 1, for any pair of adjacent nodes u and v, edges (u, v) and (v, u) are collapsed into a single (undirected) link, for brevity.

Traffic. We group the traffic that traverses an input network $(\mathbf{N}, \mathbf{E})$ in *demands*. We indeed define a set $\mathbf{D}$ that contains all the demands to be forwarded in the network. Each demand $d \in \mathbf{D}$ has a source node $src(d) \in \mathbf{N}$, a destination node $dest(d) \in \mathbf{N}$, and a bandwidth requirement $bw(d)$. Several demands with the same source and destination may exist, making $\mathbf{D}$ potentially larger than $|\mathbf{N}|^2$. In the example of Fig. 1, we assume that there is exactly one demand per pair of nodes, as illustrated in Fig. 2.

Segment Routing. Traffic is assigned to forwarding paths, hence to network edges, according to routing protocols and their configuration. In order to fulfil our goal of quickly reacting to unexpected events, we rely on Segment Routing (SR) to compute the paths followed by every input demand.

In SR [12], the path used for any demand $d \in \mathbf{D}$ is decided by the (ingress) router from which d enters the network. Such an ingress router, indeed, can be configured to specify a list of nodes that have to be traversed (in sequence) by packets in d, before reaching the destination. We denote all those nodes to be traversed by packets in d as *segments* (for d), and always conventionally include the source and destination of d among the corresponding segments. We also refer to the ordered list of segments to be sequentially traversed by packets for d as the *SR path* for d. Information on the SR path for d (with the exception of demand source and destination) is added by

$src(d)$ to the header of packets destined to d, in order to ensure consistent forwarding across the network. For example, in Fig. 1, we can configure router 9 to impose the SR path $[9, 11, 1]$ on packets for 1, i.e., forcing the demand sourced at 9 and destined to 1 to cross 11 before being delivered to 1.

The paths followed by packets derive from the concatenation of shortest paths (computed according to edge weights) between segments. For instance, the SR path $[9, 11, 1]$ represent all paths deriving from the concatenation of the shortest paths from node 9 to 11 with the shortest paths from 11 to 1. In Fig. 1, this implies that packets from 9 to 1 would be forwarded over network paths $(9, 7, 11, 1)$ and $(9, 8, 11, 1)$.

The SR path of some demands may contain only its source and destination: the shortest paths from the source to the destination are used in this case.

B. Problem Statement

By imposing how demands are forwarded through the network, configured SR paths control the traffic load on every link. In particular, traffic volumes are assigned to edges according to even load-balancing. A more precise definition follows. We denote the set of links in the shortest paths from a node s to a node t with $S_{s,t}$. For every demand d with an SR path including $[s, t]$ as a sub-sequence, every source of any link in $S_{s,t}$ distributes $bw(d)$ evenly among its outgoing edges in $S_{s,t}$. Consider again the previously-discussed case where $[9, 11, 1]$ is the SR path associated to a demand d. In that case, all edges in $S_{9,11}$ are assigned to a traffic volume of $bw(d)/2$ — because packets for d are load-balanced over two network paths $(9, 7, 11)$ and $(9, 8, 11)$. In contrast, edge $(11, 1) \in S_{11,1}$ is assigned to a traffic volume of $bw(d)$.

We are actually interested in the *edge utilization*, that is, the relative load of an edge with respect to its capacity. More formally, we define the load of an edge e, denoted with $load(e)$, as the sum across all demands of the traffic assigned to e, according to a given set of SR paths. In addition, we define the utilization $util(e)$ of an edge e as the ratio between its load and its capacity, i.e., $util(e) = load(e)/capa(e)$

In this paper, we aim at studying extremely-fast traffic engineering techniques that reduce edge utilizations upon unexpected events. We consider two scenarios.

The first scenario is the *time-constrained min-max utilization*. It consists in minimizing the maximum edge utilization, under the constraint that the computation time has to be lower than a given amount (e.g., 1 second). In other words, this scenario adds a time threshold to the classic goal (considered in early traffic-engineering works [6] as well as prior contributions on SR [9], [13]) of optimizing resource allocation.

The second scenario, that we called *fast-reaction TE*, consists in efficiently computing an SR path for every input demand such that the resulting SR paths (1) satisfy an input constraint defined on edge utilization, and (2) can be quickly installed in the network. More precisely, we constrain the edge utilization to be lower than a certain threshold (e.g., 1.0, to avoid congestion), and we try to minimize both the path computation time and the number of changed SR paths.

C. Shortcomings of previous approaches

Two main algorithms, respectively based on mixed integer linear programming (2-SR MILP [9]) and constraint programming heuristics (DEFO [13]), have been proposed to compute SR paths given an input network and a set of demands. Unfortunately, both algorithms exhibit limitations for our considered scenarios, hence to quickly react to unexpected events.

We illustrate their shortcomings on the network in Fig. 1, assuming a realistic demand matrix generated with the technique described in [16] – the resulting distribution of demand bandwidths is represented in Fig. 2. On those network and demands, we consider the time-constrained min-max utilization scenario with a time budget of 1 second.

Results of previous algorithms and our proposal are summarized in Table I. Both 2-SR MILP and DEFO return SR paths inducing a maximum edge utilization is significantly higher than the optimum of 0.9, computed as solution of multi-commodity flow problem [17]. In contrast, our LS algorithm finds SR paths leading to maximum edge utilization of 0.94, which is much lower than previous SR algorithms and closer to the optimum. While performance of all techniques can be improved by relying on more powerful servers, those results (as well as those in §V) highlight the competitive advantage of our LS algorithm with respect to the previous ones when run on the same hardware platform.

Lower bound	2-SR MILP	DEFO	Our LS
0.9	1.15	2	0.94

TABLE I: Max edge utilization of solutions provided by candidate algorithms to solve the min-max utilization problem defined on Airtel (see Figs. 1 and 2) in 1 second.

MILP algorithms hardly scale. A MILP model for traffic-engineering with SR has been proposed in [9]. This model is targeted to compute SR paths with maximum length of 3, that is, with at most one detour from the source to the destination.

While this model has been shown to be practical for offline traffic engineering on relatively-small networks (up to 30 nodes), it inherits the drawbacks of the MILP optimization framework. Namely, to guarantee optimality of its final solutions, it sacrifices time efficiency. Hence, despite limiting the length of SR paths structurally reduces the search space and improves its scalability, this approach quickly solves only small traffic-engineering problem instances, where network size and number of demands are limited.

As an illustration, we considered the problem instance in Figs. 1 and 2. We ran a state-of-the-art solver (Gurobi [18]) on a variant of the original MILP model, that we subsequently refer as 2-SR MILP, where we avoid demands to be fractionally split at the ingress – which we consider unpractical. The best solution found by the solver after 1 second has been quite far away from the optimum. Indeed, solving the MILP model returned a solution where the maximum link utilization is 1.15, while the multi-commodity flow lower bound is 0.9. A much

better solution (with maximum edge utilization of 0.95) has been found by running the same algorithm for 2 more seconds.

Those results confirm that the performance of MILP-based techniques quickly degrades as soon as problem instances grow, with optimization solvers unable to efficiently compute even LP relaxations for medium-size networks (e.g., with more than 20 nodes). Further confirmations of this intuition are provided by our large-scale evaluation (see §V).

DEFO is relatively slow. An alternative to MILP has been proposed in [13], where a quite different approach and feature set has been targeted. The resulting proposal, DEFO, is a heuristic implemented within the constraint programming (CP) optimization framework. This heuristic is tailored to (i) address several traffic-engineering problems, even for huge networks, under a common framework; and (ii) sacrifice optimality for time efficiency, by exploring randomly-chosen, distant portions of the search space, to escape local minima.

While DEFO has been shown to have good performance at scale, its generality and specific design choices tend to hamper its ability to find a solution in a very short time. Indeed, we experimentally found that DEFO typically requires a certain amount of time to find good solutions, even for traffic-engineering problem instances of limited size.

As an example, we ran DEFO to solve time-constrained min-max utilization scenario on the case represented in Figs. 1 and 2. The best solution found by the solver after 1 second has been even farther from the optimum than 2-SR MILP. Indeed, DEFO returned a solution where the maximum link utilization is slightly more than 2. Even with a time budget of 10 seconds, it has only been able to decrease the maximum edge utilization to 0.96, i.e., returning a solution worse than the one found by our LS algorithm in 1 second.

Our extended evaluation (see §V) confirms the good scalability of DEFO, but also the much higher effectiveness of our LS approach with respect to DEFO for small time budgets.

III. Local Search

Local search (LS) is a general optimization approach that has been used to quickly find good solutions to (several) hard optimization problems.

Basically, LS starts from an initial solution and iteratively goes from that solution to another one, by applying local changes called *moves*, until a stop criterion is met — e.g. the solution is good enough, or a time limit. The *neighborhood* of a solution is the set of all the solutions that can be reached with a single move. In our context, a solution is the set of the segment routing paths used by the demands; a move consists in changing the segment routing path of one demand; and the neighborhood is thus a set of solutions that only differ by a single segment routing path.

LS algorithms must use moves that are adapted to the problem, i.e. that are likely to improve the solution in one iteration, and reach the optimum after a limited amount of such iterations. When the neighborhoods are too large to explore, a heuristic can be used to focus on the moves that most likely lead to a better solution.

One of the simplest LS algorithms, known as hill climbing, consists in systematically moving to the neighbor solution that leads to the best improvement of the objective function. When no such solution exists, we say that the search is stuck on a locally optimal solution or *local optimum*; those can be much worse than the actual optimal solution. To cope with this problem, LS algorithms are typically guided at a higher level by meta-heuristics (e.g., simulated annealing [19] or tabu search [20]), that focus less on immediate improvement of the current solution to try and escape optima.

In the following, we provide details on our proposed LS algorithms for the time-constrained min-max utilization and fast-reaction TE scenarios (see §II).

A. Moves and Neighborhood

We use moves that change the current SR-path of one demand, since rerouting a demand is likely to decrease the load on the maximally-utilized edge. Unfortunately, the number of possible alternative SR-paths is exponential in the number of segments. We thus focus on smaller neighborhoods defined by the following moves:

- `insert`: insert a segment (in the middle of the path);
- `remove`: remove a segment;
- `replace`: replace a given segment;
- `reset`: remove all the segments.

Using those moves, it is possible to iteratively reach any solution contained in the original exponential neighborhood by applying a sequence of moves to a given path. This guarantees that there is always a sequence of moves that connects the current solution to the optimal solution. We say that the solution space is *connected* by our neighborhoods.

Note that only the `insert` and `remove` moves are required to ensure that our neighborhoods connect the solution space. However, `replace` and `reset` provide important shortcuts that explore the search space more efficiently. The `reset` move is particularly important to reduce the number of demands actually using segment routing.

B. Intensification and Diversification

While the neighborhoods seem reduced, exploring them is still an expensive task due to the large number of solutions to evaluate. We bias this exploration by focusing on the parts of the neighborhood that are the most likely to improve the current solution. Our LS algorithm relies on a stochastic heuristic to detect such parts. The intuition behind this heuristic is that changing the path of a demand that is using one of the most loaded edges is likely to improve the solution. To achieve this, we first randomly select an edge and then randomly select a demand that is routed on that edge. The probability p_e of selecting edge e is determined by its utilization and by an *intensification coefficient* denoted α:

$$p_e = \frac{util(e)^\alpha}{\sum_{e \in \mathbf{E}} util(e)^\alpha}. \quad (1)$$

High values of α increase the chance of selecting the most loaded edges — i.e. intensification — while low values flatten

the edge distribution — i.e. diversification. Particularly, setting α to 0 results in a uniform selection of edges. The probability p_d of selecting demand d is similarly determined by the quantity of flow that demand d forwards on edge e and by a second intensification coefficient denoted β:

$$p_d = \frac{load(d,e)^\beta}{\sum_{d \in \mathbf{D}(e)} load(d,e)^\beta}. \quad (2)$$

where $\mathbf{D}(e)$ denotes the set of demands routed on edge e.

We measured the impact of both coefficients on two large instances. Fig. 3 and Fig. 4 respectively illustrate the impact of both coefficients on the quality of the solution returned after 1 second and 2 minutes of computation[1]. We observe that preferring highly loaded edges is important while focusing too much on heavy demands has a negative impact on the minimization process.

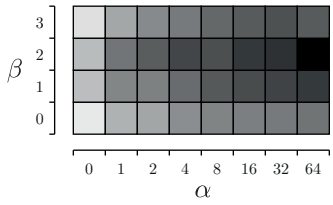


Fig. 3: Impact of different values for both coefficients with a timeout of 1 second. The darker the better.

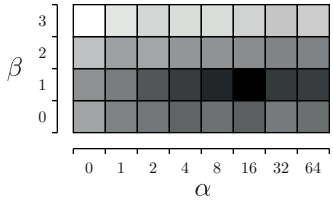


Fig. 4: Impact of different values for both coefficients with a timeout of 120 seconds. The darker the better.

C. Meta-heuristic

The combination of our moves and the intensification process aggressively improve the objective function. Unfortunately, we experimentally observed that the simple hill climbing heuristics quickly reaches local optima, tends to remain stuck inside them in many cases. To escape local optima, we added *perturbations*, a meta-heuristic component used in Variable Neighborhood Search (VNS) [21].

When stagnation is detected, i.e. after visiting several neighborhoods without improvement, our algorithm tries to force a movement, that perturbates the solution. Contrary to simulated annealing, we do not choose the movement according to the way it affects the objective function; instead we choose a *random* move. In particular, we randomly force either a `remove` or a `reset` move. Then, we expect the aggressive intensification process to quickly reach a new local optimum that might improve on the pre-perturbation solution. If it does not, our version of VNS goes back to the best-so-far solution after a few hundred iterations.

Our meta-heuristic is engineered to get good solutions quickly, instead of exploring the solution space more broadly like a simulated annealing-based algorithm would do. Using `remove` and `reset` moves often enable to backtrack to a configuration where a demand uses less physical links, since it has less segments. This tends to decrease the general load of the network despite increasing maximum utilization, freeing some capacity for the aggressive intensification to use.

Our experiments show that our meta-heuristic tends to find nearly optimal solutions for realistic networks (see §V).

IV. Implementation

One of the main concerns when implementing LS algorithms is the speed of its operations. To explore the search space, MILP relies on global LP reasoning and DEFO on CP's inference. Instead, LS assumes that a selection of cheap-to-compute moves and heuristics will be enough to explore the most interesting solutions for the input problem instance. Thus, the sheer speed of the building blocks is central to the success of any LS approach. For instance, computing the load of every edge from scratch after every candidate move would result in an overly costly $\mathcal{O}(|\mathbf{N}||\mathbf{E}|)$ effort, so we try to recompute only the parts that are affected by changes[2].

In this section, we present the data structures that we used and the design choices that we made to implement our LS algorithm. More precisely, we describe how to efficiently evaluate our admitted moves, and how to select edges and demands efficiently with *cumulative trees*. We also show how to avoid some useless computation to speed up our neighboorhood exploration.

A. Solution State

A solution of the considered traffic-engineering problems is characterized by the SR path on which each demand is forwarded. In our LS algorithm, we store each path as a vector representing the sequence of nodes: for instance, a vector of two elements corresponds to an SR path with only demand source and destination as segments.

To maintain the state of the links, we use an array `load` that maps every edge to the total amount of traffic forwarded on that edge. This array allows us to efficiently evaluate the impact of each move on the objective function (see below).

B. Preprocessed Data Structures

In addition to the solution state, we maintain two data structures to manipulate shortest paths efficiently. Let $S_{s,t}(e)$ be the fraction of flow forwarded on edge e when a demand is routed on the equal cost multi-paths from node s to node t. The first data structure maps each pair of distinct nodes (s,t) and each edge e to $S_{s,t}(e)$ while the second data structure is used to iterate efficiently on the edges in $S_{s,t}$.

[1]Greys are computed with this formula $output = (\frac{input-min}{max-min})^{1/2}$.

[2]Similar mechanisms are used in [6] to reduce recomputation.

We implemented these data structures with arrays to perform queries on the shortest paths with optimal time complexities. This choice however has a substantial memory cost of $\mathcal{O}(|\mathbf{E}||\mathbf{N}|^2)$. Despite the high memory cost, we are able to use these structures to solve instances with several hundred of nodes with less than 16GB of RAM.

C. Move Evaluation

Efficient move evaluation is the most critical operation of our LS algorithm — and probably of any LS algorithm. To evaluate the impact of a move acting on a demand d, we first need to modify the state of the current solution by applying this move. We perform such modifications in two steps:

1) subtracting d's bandwidth requirement from the load on the links belonging to the old paths for d;
2) adding d's bandwidth requirement to the links in the new paths for d.

Of course, only the load of the edges contained in the sub-path impacted by the move has to be updated. For instance, consider a demand d with SR path $[s, a, t]$. If we want to insert a new segment m between a and t we first remove the flow between segments a and t. This corresponds to perform the following operations for each edge $\texttt{load}(e) = \texttt{load}(e) - S_{a,t}(e) \cdot bw(d)$. Then, we insert the flow between segments a and m, and the flow between segments m and t. This corresponds to perform the following operations for each edge $\texttt{load}(e) = \texttt{load}(e) + S_{a,t}(e) \cdot bw(d)$. We can efficiently perform those steps using both preprocessed data structures from the previous section by iterating exactly on the impacted non-empty edges.

Once a move has been applied and its impact on the solution evaluated, we need to undo the move and restore the state of the solution in order to evaluate the next move. While this operation can easily be performed by applying the opposite of the move, we follow a more time-efficient approach. Before applying any move, we indeed save the set of edges impacted by the move, denoted by Δ, as well as their corresponding utilizations (values in `load`). This way, we can easily restore the state of the current solution by iterating on the edge in Δ and reassigning the load of each edge to its saved value.

Maintaining Δ has many advantages over the "opposite move" approach. First it is faster than applying the opposite move. Second, it reduces the potential floating point errors due to additions and subtractions. Third, it allows us to efficiently evaluate a move only on the part of the network actually impacted by the move. We adopt the same approach to update other data structures, like the cumulative tree presented below.

D. Prefiltering moves

When exploring neighborhoods, we observe that many moves do not even decrease the current maximum edge. Checking how a move affects a particular edge can be done in constant time, provided we can compute any $S_{i,j}(e)$ in constant time (see § IV-B). For instance, consider again an SR path of $[s, a, t]$, for a demand d. If we insert segment m between a and t, the difference in load for each edge e will be:

$$(S_{a,m}(e) + S_{m,t}(e) - S_{a,t}(e)) \cdot bw(d).$$

We implemented this kind of "guard" on the `insert` and `replace` moves by simply skipping moves that do not decrease the maximum edge utilization.

E. Edge and Demand Selection

Efficiently selecting the next edge and the next demand to build our neighborhood is a critical operation that threatens to drastically reduce the efficiency of our algorithm. Selecting edges and demands as explained in §III-B corresponds to the roulette-wheel problem: given a list of n weighted items, randomly select an item in the list such that the distribution of the randomly selected items matches the distribution of weights. Particularly, we need a fast data structure that enables such selection, but also allows us to insert, remove and change the weight of any item.

We propose a solution based on a *cumulative tree*. A cumulative tree is a complete binary tree[3] that stores weighted items in its leaves. The weight of an internal node is the sum of the weight of its children. Fig. 5 illustrates a cumulative tree with 4 items. By relying on cumulative trees, we can select a random item in $\mathcal{O}(\log n)$ with a single randomly generated number. Also, we can remove and insert items with a complexity of $\mathcal{O}(\log n)$ in time and $\mathcal{O}(n)$ in space.

We now detail how to perform the previous required operations on this structure:

Random selection. The first step of random selection is to generate a random number $r \in [0, W[$ where W is the sum of the weighted items in the tree — i.e. the weight of the root node. We then perform a binary search as follows: if r is lower than the weight of the left child then recurse on the left child; otherwise, subtract the weight of the left child from r and recurse on the right child. Let us illustrate this procedure on the tree of Fig. 5 with random number $r = 4$. Since the weight of the left child of the root is $5 \geq 4$, we go left. Then, since the weight of the left child is $2 < 4$, we subtract 2 from 4 and go right which is the leaf we are looking for.

Weight change. To change the weight of an item, we just need to change the weight of the corresponding leaf and then propagate this change to the root by recomputing the weight of all its ancestors. Fig. 6 illustrates this process by changing the weight of the third leaf of the tree in Fig. 5.

Insertion. We can insert a new item by creating a new leaf at the end of the current level of our complete binary tree. If the current level is already full, we simply create a new level and insert or item as the first leaf of this level. As before, we then need to propagate this change to the root node.

Removal. We must ensure that the tree is still complete after the removal. To achieve that, we first move the last leaf of

[3]This assumption simplifies the following operations and allows us to efficiently implement our tree in an array (see binary heap in [22])

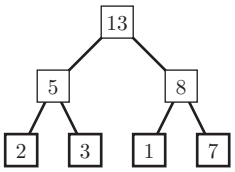


Fig. 5: A cumulative tree with 4 items.

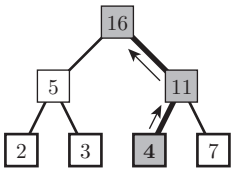


Fig. 6: Weight changes are propagated to the root node.

the tree to the position of the removed leaf (see Fig. 7 a). We then propagate this change starting from the new and the old positions of the last leaf (see Fig. 7 b).

The cumulative tree that maintains our edge selection has a fixed size of $|\mathbf{E}|$ items and only has to handle weight change operations. Demand selection is more complex since it requires each edge to maintain its own cumulative tree of demands which are subject to both insert and remove operations. The space complexity of demand selection is thus $\mathcal{O}(|\mathbf{D}||\mathbf{E}|)$ — though it is much smaller in practice.

All the previous operations are built on the assumption that we can easily access the leaf associated to an item. This is not a problem in the context of edge selection since the mapping between edges and leaves is static. We however need to dynamically maintain the mapping between demands and leaves due to insert and remove operations. We solve this problem by maintaining the mapping with a hashtable. Note that this does not change the complexity of our operations.

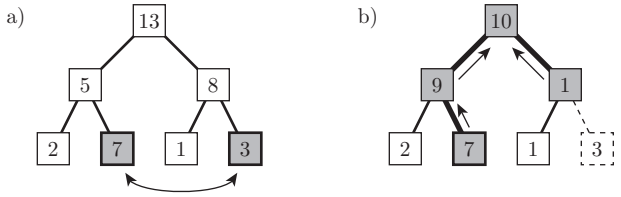


Fig. 7: Item can be removed in $\mathcal{O}(\log n)$.

V. Evaluation

We now report on the extensive evaluation of the proposed LS algorithm, and its comparison with alternative techniques.

We used 260 topologies from the Topology Zoo [15] with sizes ranging from 4 to 197. Demand matrices (5 per topology) were generated using a gravity model [16]. To normalize those matrices, we divide all demands by a constant to get a multi-commodity flow [17] lower bound (MCF) value of 0.9. However, the MCF is *not* the optimal value for segment routing. Since SR cannot split traffic with arbitrary ratio, we may not able to reach the MCF (especially in small networks). We thus filtered out instances that cannot be put below 1.0 of max utilization in 60 seconds by any of the evaluated solvers. This leaves 233 topologies out of 260.

The LS algorithm was configured to only generate path with at most one detour (3 segments). At each iteration, it selects one of the most loaded edges ($\alpha = \infty$). Demands are randomly selected such that their distribution matches the distribution of their bandwidth ($\beta = 1$).

All our experiments were run on a 40-core 3.10GHz computer with 128 GB of RAM with a Java 1.8 JVM, every process was limited to 16 GB of memory and 4 concurrent threads (only the MILP solver is multithreaded).

A. Time-Constrained TE

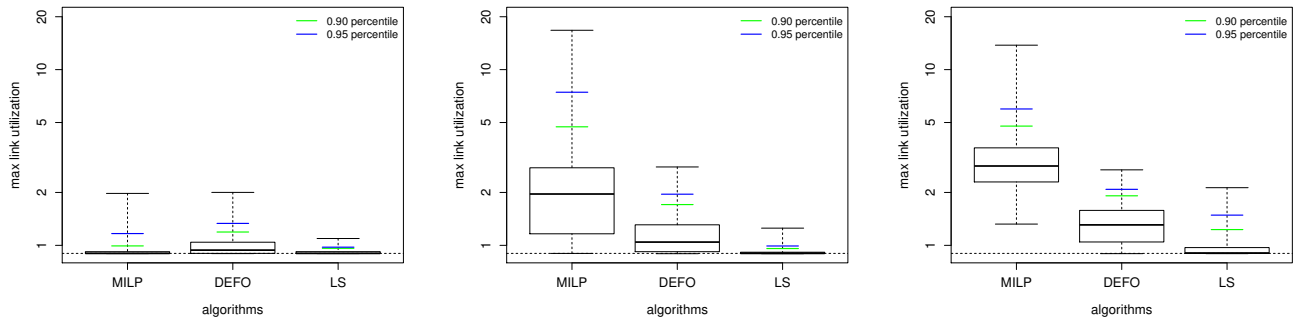
We compare our LS algorithm with the two state-of-the-art alternatives described in §II: We refer to them as DEFO (for the constraint programming solver described in [14]) and 2-SR MILP (for the integer linear programming approach with even traffic splitting, inspired by [9]). We run the three algorithms to solve a time-constrained min-max utilization scenario with a timeout of 1 second. Note that we implemented the algorithms such that they return the current paths if they do not find an optimized solution in the given timeout. In the following, we generically refer to the three compared algorithms with the name of solvers.

Our algorithm computes in 1 second better paths than its alternatives. Experimental results are summarized in Fig. 8. We observed that all solvers tend to find worse solutions when the topology size increases; hence, the figure contains three plots, for small, medium and large topologies respectively. Globally, they highlight that all the three solvers find good solutions for most small instances, except for some where the unsplittable flow assumption still make the min-max problem hard to solve. On medium-sized topologies, 2-SR MILP sometimes returns solutions worse than initial configurations (with no segment beyond source and destination for any demand). DEFO improves solution quality with respect to 2-SR MILP, but it still exceed link capacities in more than 50% of the cases. In contrast, our LS algorithm manages to compute much better SR paths, implying a maximum link utilization smaller than 1 in almost 95% of our experiments. On large topologies, 2-SR MILP is never able to remove congestion, and DEFO does that in a handful of cases. Again, our LS algorithm still removes congestion in most of the experiments. Nevertheless, a closer analysis of our results also shows that the biggest topologies in our dataset represent a hard challenge even for our LS approach, which could not lower the maximum link utilization below 1 (for this extreme traffic engineering scenario, in 1 second) on topologies with more than 100 nodes.

B. Fast Reaction TE for Congestion Removal

In a second set of experiments, we measure the response time of our solvers for significant traffic changes. Namely, we take one solver and one topology at the time. We initially feed the solver with the topology and one of the 5 demand matrices generated for that topology. Then, we iterate over the 5 matrices and ask the solver to bring the maximum link utilization below 1 at every matrix change.

Our algorithm achieves sub-second congestion removal even in large topologies. As an illustration of this set of experiments, Fig. 9 plots the evolution of the maximum link



(a) Small topologies (<20 nodes). (b) Medium topologies (20<nodes<40). (c) Large topologies (≥40 nodes).

Fig. 8: Maximum link utilization achieved in 1 second by our LS algorithm, DEFO and 2-SR. The y-axis is in logarithmic scale. Top and bottom of the boxes represents 25-th and 75-th percentiles, the thick line within the boxes represents the median, and whiskers map to the minimum and maximum values. The dashed horizontal line is the lower bound (0.9 for all topologies).

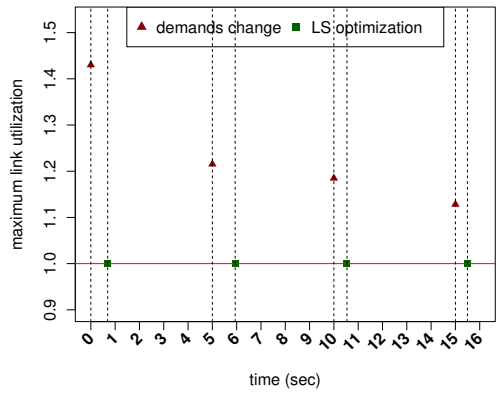


Fig. 9: Even for the largest topology in our dataset (UsCarrier with 156 nodes and 24,806 demands), our LS algorithm provided sub-second reaction to significant demand changes.

utilization over time when we apply our LS algorithm to UsCarrier, the biggest topology in our dataset. Our solver finds SR-paths under 1.0 for the first traffic matrix. Then, at time t=0, we change the demand matrix while keeping the computed paths: This brings a maximum link utilization to more than 1.4 as highlighted by the first "demands change" point in Fig. 9. We then ask the solver for new paths that deal with the change and remove congestion. After less than 1 second, our LS algorithm returns paths that induce a maximum link utilization of 1, as reported the first "LS optimization" point in the plot. We then change the demand matrix again (at time t=5), display the new maximum link utilization, and report on the second path re-optimization (around t=6). We iterate this process for all the remaining demand matrices. For all simulated demand changes, the reaction time of our LS algorithm is under 1 second.

We now compare results of our LS algorithms with its alternatives across all the topologies in our dataset. To encompass a simple, we also evaluate the SP heuristic, which is similar to the constrained shortest-path first one used for MPLS routing [23]. SP computes paths considering one demand at the time, from the largest to the smallest. In particular, it assigns each demand to the path that leads to the smallest increase of the maximum utilization, using at most one additional segment.

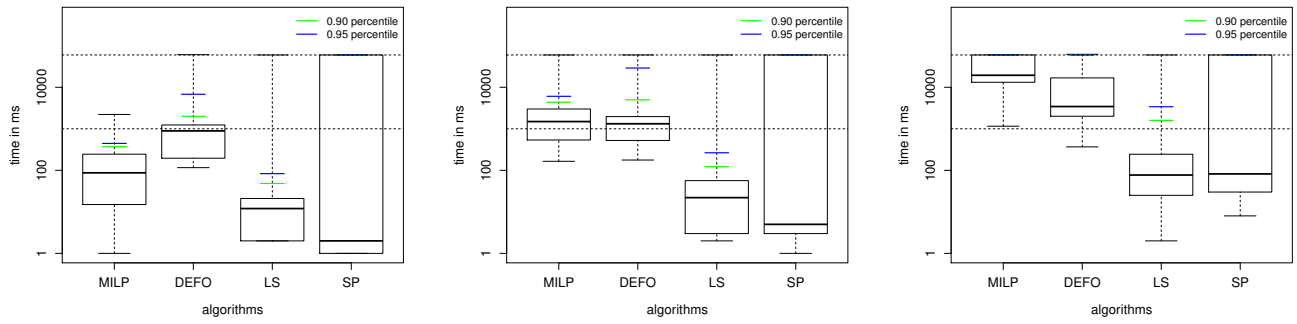
Our algorithm systematically removes congestion in far less time than its alternatives. Experimental results are summarized in Fig. 10. Once again, bigger topologies make the TE exercise harder. For all solvers, most ($> 95\%$) small and medium topologies are solvable under 60s, but only LS reaches the congestion removal objective in less 1 second, systematically across our experiments. For large topologies, 2-SR takes more than 60 seconds in more than 25% of the experiments, while DEFO removes congestion in less than 60 seconds in most of the cases. In contrast, our LS algorithm handles almost 90% of the experiments in 1 second – only cases on our biggest topologies forces it to take more time.

SP takes less time than other solvers on at least half of the instances. However, it is much less robust, returning paths that break the 1 max utilization limit in many cases. In small and medium topologies, SP computation time can make it a worthy starting point if having a high number of paths with an intermediate segment is not an issue. This possibility becomes less and less interesting as the topology grows, as shown by results on larger topologies.

C. Fast Reaction TE with Optimized Link Utilization

We finally track the cases that our LS algorithm can handle when we require lower and lower link utilization. Namely, we repeat the experiments above, but setting a target maximum link utilization progressively closer to the MCF of 0.9.

In most experiments, our algorithm gets close to the link-utilization lower bound in milliseconds. Table II reports on both the computation time and the percentage of rerouted demands. Unsurprisingly, tighter constraints on link utilization makes our algorithm run for more time and reroute more demands. However, for most experiments, our LS solver achieves a maximum link utilization of 0.92 in less than 1 second, rerouting 5% of the demands on average.



(a) Small topologies (<20 nodes). (b) Medium topologies (20<nodes<40). (c) Large topologies (≥40 nodes).

Fig. 10: Time taken by the evaluated algorithms to avoid congestion. We set a timeout of 60s and add a point at 60s when an algorithm does not find any solution before this timeout. The y-axis is in logarithmic scale. Top and bottom of the boxes represents 25-th and 75-th percentiles, the thick line within the boxes represents the median, and whiskers map to the minimum and maximum values. The top dashed horizontal line is the timeout of 60s, and the middle dashed line is our target of 1 second.

Max link utilization	Execution time					Rerouted demands
	50ms	100ms	200ms	500ms	1sec	
0.92	61.8%	68.3%	72.3%	76.0%	77.8%	5.0%
0.94	68.2%	75.1%	78.9%	82.1%	83.8%	4.1%
0.96	74.4%	81.8%	85.5%	88.6%	90.8%	3.3%
0.98	78.4%	85.5%	88.5%	90.8%	92.7%	2.7%
1.0	82.0%	88.1%	91.5%	93.9%	96.0%	2.2%

TABLE II: Percentage of experiments where our LS algorithm matches stringent link-utilization and time constraints. The last column reports the percentage of demands that have to be rerouted (on average, for paths computed in 1 second).

VI. Conclusions

Segment Routing is becoming a popular technology for traffic engineering, with applications ranging from load balancing [9] to link-failure recovery [8] and complex path requirements [13]. Previous approaches can be used to address *expected* network dynamics, but can be hardly adopted to deal with *unexpected* events, like arbitrary traffic-volume modifications due to sudden flash crowds. To support quick answers to unexpected events, we propose an approach based on Local Search (LS). With respect to previously-proposed approaches relying on integer linear programs (MILP) or constraint programming (CP), our LS algorithm sacrifices completeness (of search space exploration) to guarantee that it can always return a solution independently of its execution time — the returned solution strictly improves with execution time. We *design* our algorithm for quick computation of a limited set of quickly-implementable path changes that re-optimize per-link utilization upon unexpected events. Our evaluation on (real) networks shows that our algorithm outperforms existing traffic-engineering ones, at short time scales, across all our experiments. Also, it is the only one that systematically react to (even dramatic) traffic changes in less than one second.

Acknowledgements

This work has been partially supported by the ARC grant 13/18- 054 from Communauté française de Belgique.

References

[1] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *ACM CCR*, vol. 38, no. 2, pp. 69–74, 2008.
[2] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM*, 2013.
[3] C.-Y. Hong *et al.*, "Achieving High Utilization with Software-driven WAN," in *SIGCOMM*, 2013.
[4] P. Wendell and M. J. Freedman, "Going viral: Flash crowds in an open cdn," in *IMC*, 2011.
[5] A. Elwalid *et al.*, "MATE: multipath adaptive traffic engineering," *Computer Networks*, vol. 40, no. 6, pp. 695–709, 2002.
[6] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *INFOCOM*, 2000.
[7] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *HotSDN*, 2013.
[8] F. Hao, M. Kodialam, and T. V. Lakshman, "Optimizing restoration with segment routing," in *INFOCOM*, 2016.
[9] R. Bhatia, F. Hao, M. Kodialam, and T. V. Lakshman, "Optimized network traffic engineering using segment routing," in *INFOCOM*, 2015.
[10] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and Precise Triggers in Data Centers," in *SIGCOMM*, 2016.
[11] O. Tilmans, T. Buhler, S. Vissicchio, and L. Vanbever, "Mille-Feuille: Putting ISP traffic under the scalpel," in *Hotnets*, 2016.
[12] C. Filsfils *et al.*, "Segment Routing Architecture," Internet draft, 2014.
[13] R. Hartert *et al.*, "A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks," in *SIGCOMM*, 2015.
[14] ——, "Solving segment routing problems with hybrid constraint programming techniques," in *CP*, 2015.
[15] S. Knight *et al.*, "The internet topology zoo," *IEEE JSAC*, vol. 29, no. 9, pp. 1765–1775, 2011.
[16] M. Roughan, "Simplifying the synthesis of internet traffic matrices," *ACM CCR*, vol. 35, no. 5, pp. 93–96, 2005.
[17] F. Shahrokhi and D. W. Matula, "The maximum concurrent flow problem," *Journal of the ACM*, vol. 37, no. 2, pp. 318–334, 1990.
[18] I. Gurobi Optimization, "Gurobi optimizer reference manual," 2015. [Online]. Available: http://www.gurobi.com
[19] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by simmulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.
[20] F. Glover, "Tabu search: a tutorial," *Interfaces*, vol. 20, no. 4, pp. 74–94, 1990.
[21] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.
[22] R. Sedgewick, *Algorithms*. Pearson Education India, 1988.
[23] B. S. Davie and Y. Rekhter, *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc., 2000.