Verifying maximum link loads in a changing world

Tibor Schneider
ETH Zürich

Stefano Vissicchio
University College London

Laurent Vanbever
ETH Zürich

Abstract

To meet ever more stringent requirements, network operators often need to reason about worst-case link loads. Doing so involves analyzing traffic forwarding after failures and BGP route changes. State-of-the-art systems identify failure scenarios causing congestion, but they ignore route changes.

We present Velo, the first verification system that efficiently finds maximum link loads under failures *and* route changes. The key building block of Velo is its ability to massively reduce the gigantic space of possible route changes thanks to (i) a router-based abstraction for route changes, (ii) a theoretical characterization of scenarios leading to worst-case link loads, and (iii) an approximation of input traffic matrices. We fully implement and extensively evaluate Velo. Velo takes only a few minutes to accurately compute all worst-case link loads in large ISP networks. It thus provides operators with critical support to robustify network configurations, improve network management and take business decisions.

1 Introduction

Link loads—how much traffic crosses each link—are a key indicator of network performance. High link loads, in particular, increase the likelihood of congestion, packet drops, and inflated delays. To meet stringent service-level agreements, operators often need to reason about the worst-case load that every link can realistically experience during network operation – e.g., to check that all loads are below a safety threshold, and adapt routing configurations if they do not.

Identifying worst-case link loads is hard, though. Operators often have tools to measure traffic and model traffic patterns, even long term [26, 35]. However, determining worst-case loads requires to scrutinize traffic forwarding for a *huge range of possible events*. During network operation, links and nodes fail, and BGP routes for external destinations appear, disappear and change. Even a single failure or 3-4 route changes at specific border routers can overload links, as our case study on a real ISP network shows (§7).

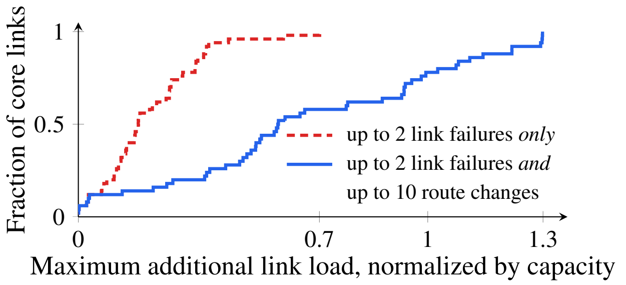


Figure 1: Additional link loads can double due to route changes compared with link failures only. This plot shows the additional traffic on all the core links of an ISP network.

Reasoning about link loads is beyond the capabilities of existing network verifiers. Most of them [1, 4, 14, 39, 40, 42, 45, 50] do not support *performance* requirements such as maximum link loads, but restrict to *functional* requirements, such as the absence of blackholes and forwarding loops.

A couple of recent contributions [7, 27, 44] focus on assessing properties on link loads. Yet, they only consider network failures and assume fixed external routes, fitting controlled network environments such as data centers. For most Internet-connected networks, *both* failures and route changes must be considered, *jointly*, to provide guarantees on link loads. As an illustration, Figure 1 depicts the additional load on the core links of a real ISP network, when simulating all one and two link failures. In the presence of up to ten route changes (solid curve), most links reach higher load than without route changes (dashed curve), and their load can be twice as big. In other words, ignoring route changes leads to vastly underestimating maximum link loads.

This paper presents Velo, the first verification system that efficiently computes the individual worst-case loads of *all* links for arbitrary link failures and external route changes. Velo does not depend on how paths are computed, and hence it works in both traditional (e.g., IGP/BGP-based) and SDN networks; it also supports typical traffic engineering technologies such as ECMP and tunnelling (e.g., MPLS).

In each run, Velo takes as input: router configurations, BGP routes, and a traffic matrix. The traffic matrix captures the specification (such as the worst-case pattern seen in the past, or the expected future traffic) for that run. Different traffic scenarios can be verified in separate runs. As in [44], operators can also input the maximum traffic volume additional to the given matrix, e.g., to model uncertainty. Further, Velo can limit its computations to user-defined subsets of failures and route changes that model realistic scenarios. For example, operators can specify a maximum number of simultaneous route changes, and define per-network and per-router stable destinations, for which route changes are not allowed.

As such, Velo supports many practical use cases including: *(i)* verification of configurations before deployment; *(ii)* adaptation of BGP preferences to make the worst-case routing states less likely; *(iii)* fine-tuning of network monitors to raise alerts only when there is a significant risk of congestion; *(iv)* support for business decisions and routing policies. Our case study (§7) exemplifies some of these use cases.

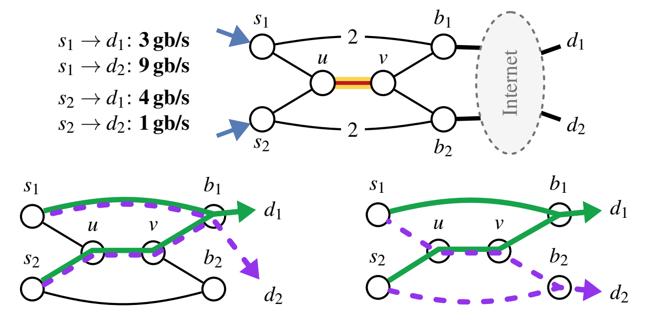
Challenge It is infeasible to iterate over all possible route changes and failure scenarios. For each destination, external networks may advertise new routes, modify existing ones, or completely withdraw them. New and modified routes have attributes (e.g., AS path or BGP communities) with a very large set of possible values. When factoring in possible failures, not only does the search space grow, but it also becomes more challenging to navigate. Indeed, failures generally affect paths to multiple destinations, hence destinations cannot be treated as independent from each other across failures. This also prevents us from extending prior verification work [4, 15, 45, 50].

Solution We greatly reduce this search space in three steps.

First, we provide a compact abstraction for capturing route changes. We do not explicitly model possible external routes, e.g., symbolically as in [4]. Instead, we model possible routes for a destination as *sets of egress routers*. Each set captures the many equivalent routes inducing the same link loads.

Second, we design Velo to explore a *minimal* set of states guaranteeing correctness. We prove that for common intradomain routing schemes (e.g., shortest-path routing), each link is maximally loaded when only one egress router is used per destination. This enables Velo to compute all the worst-case link loads in *polynomial* time. In the presence of paths deviating from the above schemes (e.g., for traffic engineering), Velo extends its search to *minimal* sets of egress routers, ensuring limited impact on Velo's efficiency.

Third, we propose an efficient technique to approximate the input traffic matrix by combining destinations with similar traffic patterns, while also providing strong accuracy guarantees. We formalize the traffic matrix approximation as a clustering problem, and prove that the clustering error bounds the approximation error of the worst-case link load.


$s_1 \rightarrow d_1$: **3 gb/s**
$s_1 \rightarrow d_2$: **9 gb/s**
$s_2 \rightarrow d_1$: **4 gb/s**
$s_2 \rightarrow d_2$: **1 gb/s**
s_1
s_2
u
v
2
2
b_1
b_2
Internet
d_1
d_2

(a) Both destinations d_1 and d_2 are advertised only to b_1.

(b) Destination d_1 is advertised to b_1 and d_2 is advertised to b_2.

Figure 2: Network that routes traffic along the shortest paths. Subfigures (a) and (b) show the forwarding paths for two possible routing advertisements of destinations d_1 and d_2. All link weights are 1, except those from s_i to b_i are 2.

Results We implement a prototype of Velo and make it publicly available[1]. Our evaluation on real network topologies shows that Velo computes the worst-case load of every link in large networks and the entire BGP routing table, *within minutes*, even considering up to two simultaneous link failures. Velo's computed maximum loads are also highly accurate, within 1% of the actual ones, and this across a wide range of traffic matrices, from heavy-tailed to almost-uniform.

2 Overview

Velo finds the worst-case load for each link by exploring the input space of failures and route changes. Consider the example in Figure 2. Routers s_1 and s_2 receive traffic for two remote destinations d_1 and d_2. Depending on the received BGP routes, d_1 and d_2 can be reached via either b_1 alone, b_2 alone, or both simultaneously. Both s_1 and s_2 send packets to the closest egress among b_1 and b_2 that can reach the destination (i.e., implementing hot-potato routing).

Let's initially focus on the link (u,v), assuming no failures. The link load depends on the BGP routes received by b_1 and b_2. For example, if only b_1 receives BGP routes for d_1 and d_2, packets from s_2 crosses (u,v) but those from s_1 do not, resulting in a total load of 5 gb/s—see Figure 2a.

To compute the worst-case link load, we conceptually need to check all the combinations of each destination being reachable from b_1, b_2, or both. For example, if b_2 receives the same BGP route as b_1 for d_1 and still no route for d_2, the load of (u,v) decreases, because the traffic from s_2 to d_1 is moved to the path (s_2,b_2). Figure 2b shows the worst-case scenario: when d_1 is only reachable from b_1 and d_2 only from b_2, then 13 gb/s are sent over (u,v) since both demands $s_1 \rightarrow d_2$ and $s_2 \rightarrow d_1$ cross that link.

[1]Available at https://github.com/nsg-ethz/velo (GPLv3 license)

Consider now any other link, for example, (s_2, b_2). Clearly, Figure 2b is not the worst-case scenario for (s_2, b_2); its load is higher if s_2 uses b_2 as next hop for both d_1 and d_2.

Real networks are much bigger than Figure 2, and typically have many more border routers and millions of destinations, making the problem challenging to solve. For example, iterating over all the possible BGP routes received by any border router for each destination does not scale. Worse yet, every possible failure affects the network topology, and so requires to re-assess the impact of each possible route change.

2.1 Problem Statement

We now describe Velo's problem and provide the intuition of how Velo manages to accurately and efficiently solve it. We refer to any router that sends traffic outside the network as *egress router*. For example, in Figure 2b, b_1 is the only egress router for d_1, and b_2 is the only egress router for d_2. An *egress change* occurs when the set of egress routers for one destination changes, such as between Figure 2a and Figure 2b.

Problem: Given *(i)* the network configuration, *(ii)* current BGP routes, *(iii)* traffic per destination, *(iv)* constraints on route changes, and *(v)* a space of failure scenarios, find the worst-case load for each link in the network, along with the egress changes and the failure scenario causing such loads.

We provide more details on Velo's inputs in the following.

Network configuration We define a network configuration as the network topology and the collection of all router configurations. We support a wide range of networking paradigms and protocols and present an abstraction in §3. Operators can provide either the *current* topology and router configurations as input, or alternative ones (e.g., for what-if analyses).

BGP routes Velo takes the current BGP routes as input. We define a BGP route as a BGP advertisement from an external network concerning a single destination and including its BGP attributes. Routes generally change over time.

Yet, not all route changes are equally likely, and some are not worth considering in practice. For example, at operational timescales, receiving new routes for a few destinations is more likely than the same occurring for all Internet prefixes.

For this reason, Velo allows to limit the set of route changes verified in any Velo's run. Operators can specify the maximum number k of egress changes, where the *number of egress changes* is the number of destinations featuring one or more egress changes. Operators can also provide constraints on specific destinations (e.g., high-volume IP prefixes) for which routes can or cannot change, and on the possible egress routers per destination (e.g., enforcing that BGP customers always advertise only their own prefixes).

Failure scenarios Contrary to route changes, network failures typically affect many forwarding paths for many (if not all) destinations. Velo takes as input a description of the space of failure scenarios to explore. This can be the set of up to l simultaneous link or node failures [42, 44], and can include Shared Risk Link Group (SRLG) information [53]. The only constraint here is that for each failure scenario, Velo can derive a new topology from the original one.

Traffic In each run, Velo takes a traffic matrix as input. Previous work often defines a traffic matrix as the traffic volumes per pair of ingress and egress routers, describing where traffic enters and exits the network. We instead define a traffic matrix as traffic volumes for each pair of ingress router and destination prefix. We do so because we are interested in assessing the impact of BGP routes which are per-destination.

While Velo accepts a single matrix as input, operators can specify the current traffic matrix, or, for more conservative analyses, they can provide Velo with a matrix derived from one or multiple congestion events recorded in the past. They can also run Velo multiple times on different matrices.

Velo models uncertainty on inter-domain traffic in the same way as QARC [44]. It indeed allows operators to specify the total amount of traffic, across all the destinations, that may need to be forwarded in addition to the input traffic matrix.

2.2 Velo in a nutshell

Verifying link loads requires exploring a gigantic search space. Besides the sheer number of routes and route attributes, link failures prevent destinations from being verified independently (i.e., one at the time). Velo addresses these challenges by massively reducing both the search space and the number of destinations. We now provide the intuition of how it does so, referring to the following sections for details.

Reducing the search space In §4, we prove that the worst-case link loads occur when for each destination, all the network routers select the same BGP route. Indeed, the worst-case load for (u, v) in Figure 2 occurs when all routers select b_1's route for d_1, and b_2's route for d_2. This key insight enables us to explore, for each failure scenario, only $|P| \cdot |N_B|$ different states, where P is the set of destinations and N_B is the set of egress routers—a polynomial search space instead of an exponential one.

The above property holds for all strictly isotone routing protocols (such as shortest path routing) that usually govern intra-domain routing. We further extend our technique to support exceptions to strict isotonicity, such as MPLS tunnels for traffic engineering, by computing the minimal set of additional states to explore in these cases (§4.1.2).

Velo iterates over all the input failure scenarios, one by one. We experimentally show (§6) that Velo is anyway more efficient than approaches attempting to prune failure scenarios, like QARC, even when considering combinations of up to 3 or 4 simultaneous failures. Intuitively, this happens because our search space reduction enables Velo to analyze any failure scenario *very* quickly.

Reducing the number of destinations In §5, we show that we can combine destinations with *similar* traffic distributions across the same ingress routers, and obtain an approximated traffic matrix that contains much fewer destinations. This approximation significantly improves Velo's running time at the expense of potentially degrading its accuracy. Yet, we present a modified k-means clustering algorithm that computes an approximated traffic matrix with provable bounds on the accuracy loss. In §6.2, we experimentally show that the approximation is highly accurate for both realistic (heavy-tailed) and unrealistic (almost-uniform) traffic matrices. We show that this clustering technique is more efficient and general than a simple approach based on heavy-hitters, as it achieves tighter error bounds by considering fewer destinations for all the traffic matrices used in our experiments.

To illustrate our approximation, consider a third destination d_3 in Figure 2 with traffic $s_1 \to d_3$ of 0.4 gb/s and $s_2 \to d_3$ of 0.3 gb/s. d_3 is also reachable via b_1, b_2, or both. The traffic for d_1 (3 gb/s from s_1 and 4 gb/s from s_2) is ten times larger than for d_3, but their distributions are similar. Considering d_1 and d_3 jointly would yield a maximum load on link (u,v) of 13.3 gb/s instead of 13.4 gb/s when analyzing d_1 and d_3 individually. Velo accurately bounds the error to 0.1 gb/s.

Dealing with traffic variability Velo's input can include an amount y of traffic additional to the input traffic matrix. There are infinite traffic matrices compatible with any given y; obviously, we cannot analyze all of them. Two key observations instead inspire Velo's approach. First, allowing y additional traffic increases the worst-case load of any link by up to y compared to its maximum load for the input traffic matrix. Second, for any link, its maximum load increases exactly by y whenever a router r sends *all* its traffic to a destination d across that link: in this case, indeed, increasing the traffic from r to d by y increases the load on that link by y.

Based on these observations, Velo accounts for y additional traffic by first computing the maximum link loads for the input traffic matrix, and then increasing the maximum link loads by y. This approach may overestimate the maximum load of a link if no router forwards all its traffic for any single destination over that link in the worst-case scenario for that destination. This tends to occur rarely in practice (less than 0.1% in our experiments in §6), implying that Velo is fully accurate in the vast majority of scenarios.

3 Model and Notation

We model the network as a graph $G = (N, E)$ with routers N connected by edges E. Border routers $N_B \subset N$ are connected to external networks that advertise BGP routes to destinations P, that is, remote IP prefixes populating the routing tables of the routers. For each destination, the network selects a set of *preferred routes*, corresponding to a set $B \subseteq N_B$ of border routers that will be used as *egress routers* (i.e., to forward traffic towards that destination). This selection can be done by a central controller [10] or in a distributed fashion using iBGP [37]. In the case of iBGP, we assume that *(i)* BGP attributes that affect routes' preference are not modified on any iBGP session, and *(ii)* all routers eventually learn their preferred routes. Several techniques are available to avoid iBGP visibility problems [36, 47, 48] or to prove the absence thereof [4, 16, 45, 49].

By focusing on egress routers per destination, we hide most of BGP's low-level details (e.g., its decision process). For any given destination d, we must only consider *(i)* the border routers that *can* receive a route for d, and *(ii)* the possible subsets of border routers that can receive *preferred* routes for d. For the former, we check whether configured eBGP ingress route maps discard all routes for d. For the latter, we analyze the attributes modified by ingress route maps, mainly the Local Preference, that affect the BGP decision process. Other attributes like the AS path can take arbitrary values, including those that result in a tie during route selection.

We denote the input traffic matrix as $\mathbf{M}$. Each element $\mathbf{M}_{s,d}$ is the ingress traffic that enters the network at router $s \in N$ with destination $d \in P$. We define a column $\mathbf{M}_d$ of $\mathbf{M}$ as the total traffic for destination d, as sourced from all ingress routers in N. The network sends traffic $\mathbf{M}_d$ for d over the computed paths for egress routers in B, resulting in load $load(e, \mathbf{M}_d, B)$ on any link $e \in E$.

Intra-domain routing Forwarding paths are computed from each router to its selected egress router. Many intra-domain routing approaches exist. Some rely on a central controller [18, 20], while others use distributed Interior Gateway Protocols (IGPs) [33, 34]. Static routes, source routing or tunnelling can also be used to forward traffic [26].

We model the intra-domain path computation using routing algebra [41], consisting of weights W, a binary operation $\oplus$ and an order relation $\succcurlyeq$. For example, the commonly used shortest-path-based IGPs [33, 34] are modeled with weights corresponding to per-link IGP costs, a binary operation that adds costs, and a binary relation that selects the shorter path.

We write link weights as $w : E \mapsto W$. We denote the optimal s-t path in the routing algebra as $p^*_{s,t} = (s, n_1, \dots, n_i, t)$ with path weight $w_{s,t} = w(s, n_1) \oplus \dots \oplus w(n_i, t)$. We assume that traffic is split equally across all next-hops of each router, i.e., the first hops of its optimal paths. This is consistent with traffic distribution in the presence of ECMP [26].

Link weights W are *strictly isotone* if $a \succ b$ implies both $a \oplus c \succ b \oplus c$ and $c \oplus a \succ c \oplus b$ for any $a, b, c \in W$. This property is both necessary and sufficient to ensure that any sub-path of an optimal path is itself optimal, allowing the paths to be computed using a generalized Dijkstra algorithm (if they exist) [41]. Note that strict isotonicity is common in intra-domain routing regardless of the forwarding paradigm (e.g., hop-by-hop vs. source routing vs. SDN).

Strictly isotone weights offer limited flexibility to achieve traffic engineering [13], so operators sometimes configure *exception paths* (e.g., MPLS RSVP-TE tunnels [3]) that carry packets over manually defined sequences of edges. We denote the set of exception paths as ρ that take precedence over shortest paths.

4 Finding Maximum Link Loads

We achieve scalability thanks to a highly efficient algorithm for finding the worst-case link load in any input topology, allowing us to iterate over a large number of failure scenarios. For any given topology, the worst-case load of a link is the sum of its worst-case loads for each destination considered independently: this is because BGP routes are specified per destination, and so are forwarding paths. To find the worst-case link loads in each topology, we thus search for the worst-case egress routers per destination, one destination at a time. Hereafter, we explain how Velo performs such a search for a given topology and a single destination.

4.1 Per-Destination Worst-Case Link Loads

The basic building block of our approach is to efficiently find the set of egress routers for a single destination that maximizes the load of a specific link. Reasoning on the worst-case egress routers per destination compared to the huge and possibly infinite space or BGP routes and their attributes is already a massive optimization. Yet, naively exploring the space of egress routers still requires exploring $O(2^{|N_B|})$ states.

In the following, we explain how we reduce our search space to $O(|N_B|)$ states whenever internal paths are computed from strictly isotone weights (§4.1.1), and slightly more states in the presence of a few exception paths (§4.1.2).

4.1.1 Strictly Isotonic Routing

Let $G = (N, E)$ be a network routing traffic matrix $\mathbf{M}$ according to protocol $(W, \oplus, \succcurlyeq)$ where W is strictly isotone. We focus on one destination d and the maximal load on link e.

Theorem 1 *For any non-empty set of egress routers $B \subseteq N_B$ and any link $e \in E$, there exists a single egress $b \in B$ that, if chosen as the unique egress for destination d, causes at least the same load on e as the set B. Formally:*

$$\forall e \in E : \exists b \in B \text{ s.t. } load(e, \mathbf{M}_d, B) \leq load(e, \mathbf{M}_d, \{b\})$$

Algorithm 1: Find the worst-case link load for all links simultaneously if all traffic is forwarded along optimal paths.

```
Data: Graph $G = (N, E)$, border routers $N_B \subseteq N$, link
      weights $w : E \mapsto \mathbb{R}$, traffic $\mathbf{M}_d$
Result: maximum link loads $y[e]$ for all links $e \in E$
$y[u, v] \leftarrow 0 \; \forall (u, v) \in E$
for $b \in N_B$ do
    $l[s] \leftarrow \mathbf{M}_{s,d} \; \forall s \in N$
    $G_{dag} \leftarrow ForwardingDAG(G, w, root = b)$
    for $u \in TopoSort(G_{dag})$ do
        $d_{out} \leftarrow |out(G_{dag}, u)|$
        for $v \in out(G_{dag}, u)$ do
            $l[v] \leftarrow l[v] + l[u]/d_{out}$
            $y[u, v] \leftarrow \max(y[u, v], l[u]/d_{out})$
```

Intuitively, Theorem 1 holds by the main property of strict isotonic routing protocols; the sub-path of any optimal path is also optimal. All traffic crossing the edge $e = (u, v)$ must follow a path that is also optimal for v, allowing us to reduce the set of egresses to the one selected by v. In case a router splits its traffic equally among multiple paths, we show that the fraction of traffic it sends across the edge e will not decrease.

Proof of Theorem 1. Consider any edge $e = (u, v)$, and let $N_e \subseteq N$ be the set of all nodes that forward traffic along e. Further, let b be an egress router preferred by v, and pick any router $n \in N_e$.

First, observe that the weight of n's optimal path towards any egress router in B, namely $w_{n,B}$, is equal to the weight of its path to b via e, i.e., $w_{n,b} = w_{n,u} \oplus w(e) \oplus w_{v,b}$. Hence, $w_{n,B} = w_{n,b}$ due to the strict isotonicity.

We now show that the total amount of traffic n sends across edge e cannot decrease when only egress b is available. We show this by comparing n's next-hops for egresses B, i.e., $nh_{n,B}$, with those for the single egress b, i.e., $nh_{n,b}$. First, we argue that $nh_{n,b} \subseteq nh_{n,B}$, and thus, the traffic n sends to any next-hop in $nh_{n,b}$ does not decrease. This follows from the fact that any optimal path of n for b is also optimal for B.

Second, we show that $nh_{n,b} \cap N_e = nh_{n,B} \cap N_e$, proving that the fraction of traffic n sends across the edge e cannot decrease. Consider any next-hop $x \in nh_{n,B} \cap N_e$: n can still reach the destination d via egress b on an optimal path via x. This is because $x \in N_e$, and thus, $w_{x,B} = w_{x,b}$. □

Theorem 1 implies that the worst-case load of any link for any destination is obtained when the entire network chooses a single egress router. More formally,

$$\max_b load(n, \mathbf{M}_d, \{b\}) = \max_B load(n, \mathbf{M}_d, B).$$

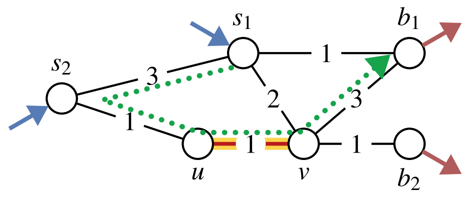


Figure 3: Example where considering less combinations of egress routers than Velo in the presence of exception paths leads to underestimating maximum link loads.

This observation directly enables us to reduce the number of explored states to $O(|N_B|)$. Indeed, Velo iterates over the border routers in N_B. For each border router b, Velo creates the forwarding graph rooted at b, which is a directed acyclic graph (DAG). It then computes the corresponding link loads by traversing the DAG in topological order and pushing traffic from each node to its outgoing edges. At the end of the iteration, it reports the worst-case scenario for each link.

Algorithm 1 shows our pseudo-code. Note, that the second-to-last line ensures traffic is split equally among next-hops, i.e., implementing ECMP. Its computational complexity is $O(|N|^2 \log|N| + |N||E||P|)$, because the two most expensive operations are (i) computing $|N_B|$ forwarding DAGs with Dijkstra's algorithm [41], and (ii) traversing a DAG in topological order for each destination prefix $d \in P$.

4.1.2 Exception Paths for Traffic Engineering

Theorem 1 does not hold in the presence of exception paths. Take the example depicted in Figure 3, with two egresses b_1 and b_2, and two ingress routers s_1 and s_2. Shortest-path routing is always used, except s_1 forwards traffic for b_1 on the path (s_1, s_2, u, v, b_1), drawn as a dotted line. The load on link (u, v) is maximized when both b_1 and b_2 are egress routers. Indeed, shortest paths are such that s_2 bypasses (u, v) if b_1 is the only egress, and s_1 bypasses (u, v) if b_2 is the only egress.

To deal with exception paths for destination d, we explore all single egress routers plus all the combinations of the border routers terminating any exception paths configured for d. The example in Figure 3 demonstrates that considering less egress routers can lead to incorrect results. We now show that considering only these combinations is sufficient to find the worst-case link loads. Let ρ_d be the exception paths configured for d, and let $T = \{b \mid (s, \cdots, b) \in \rho_d\}$ be the set of egress routers ending any path in ρ_d.

Theorem 2 *For any non-empty set of egress routers $B \subseteq N$ and any link $e \in E$, there exists a subset $B' = \{b\} \cup T'$, where $T' = T \cap B$, that causes at least the same load on e than the set B. Formally, for every link $e \in E$,*

$$\exists b \in B \text{ s.t. } load(e, \mathbf{M}_d, B) \leq load(e, \mathbf{M}_d, \{b\} \cup T \cap B)$$

Intuitively, Theorem 2 holds as nodes either forward via optimal paths, in which case Theorem 1 applies, or via exceptional path that are still available with the reduced subset of egress routers. We provide a proof in Appendix A.1.

Theorem 2 allows us to find the worst-case link loads by iterating over all subsets $T' \subseteq T$ and one other egress router in $N_B \setminus T$. We filter out combinations of egresses that cannot receive equally preferred routes due to BGP policies.

For each set B of egress routers computed as above, we construct the corresponding forwarding DAG by combining the forwarding DAGs rooted at all egresses in B. We then find the load resulting from traffic routed along exception paths, and push the remaining traffic through the DAG as in Algorithm 1. We provide a pseudo-code in Appendix A.2.

Computing link loads in the presence of exception paths has complexity $O(|N|^2 \log|N| + 2^{|T|}|N||E||P|)$, similarly to the complexity of Algorithm 1 but for the larger search space.

4.2 Restricting the Number of Route Changes

So far, we have discussed finding the worst-case link loads considering any set of route changes. However, part of Velo's input is the maximum number k of egress changes. Hence, for each link, we must find the k destinations that cause the highest increase of its load. To that end, we maintain for each link a heap of size k with the largest difference between the worst-case and current-state link loads. Maintaining the heap has a time complexity of $O(|E| \cdot |P| \cdot \log k)$ which is negligible with respect to the total time of the algorithm.

5 Approximating the Traffic Matrix

The algorithms presented in §4 scale linearly in the number of destinations. It is not rare that real routers have order of a million entries [19, 26], so finding the worst-case link load for so many destinations would be a limiting factor.

We greatly reduce the number of destinations by combining those with similar traffic patterns. Specifically, we approximate the input traffic matrix with a smaller one, leveraging the low effective rank that traffic matrices typically exhibit [6, 31].

We formulate this traffic matrix approximation problem as a clustering problem, and find that the clustering error bounds the *approximation error* δ; by how much the link loads differ when computed on the original and approximated traffic matrices. We envision that our approximation technique can be useful within other networking problems for which the size of traffic matrices limits scalability.

Approximation problem Given a traffic matrix $\mathbf{M} : |N| \times |P|$, we aim to find a clustering C of the destinations that reduces the size of $\mathbf{M}$ while also bounding the error induced by that reduction. More formally, we aim to compute a partition C of P, an approximate traffic matrix $\mathbf{A}$ of size $|N| \times |C|$, and an *error bound* ε for the approximation error δ, such that

$$\delta = \max_{e \in E} |max\,load(e, \mathbf{M}) - max\,load(e, \mathbf{A})| \leq \varepsilon, \quad (1)$$

where $max\,load(e, \mathbf{M})$ is the maximum load of link e for $\mathbf{M}$.

We take the number of clusters $|C|$ as input. Increasing $|C|$ reduce both ε and δ at the cost of running time. Operators can tune $|C|$ to find a balance between accuracy and running time.

We must however be careful to cluster together only destinations for which the routing and forwarding states are equivalent. For example, we must not aggregate destinations with different intra-domain paths for the same egress point. To this end, we provide a custom definition of *Equivalence Class*, and only cluster destinations in the same equivalence class. We distribute the available clusters $|C|$ across all equivalence classes proportional to the amount of traffic they carry. Specifically, any pair of destinations in an equivalence class must satisfy the following two conditions:

1. The same egress routers are used for both destinations in the current routing state.
2. The forwarding paths for both destinations are always equal for the same set of preferred egress routers.

In the following, we detail our clustering algorithm for destinations in the same equivalence class.

Normalized clustering algorithm We cluster together destinations with similar traffic distribution across ingress routers, i.e., if similar fractions of traffic for two destinations enter from the same routers. Such destinations *likely* produce worst-case link loads for the same routing inputs.

We define the traffic distribution $\mathbf{M}_d/|\mathbf{M}_d|$ of destination d as the fraction of traffic from each ingress compared to d's total traffic load $|\mathbf{M}_d|$. We then use the L1 norm [24] to measure the similarity of two distributions, as the L1 norm quantifies their absolute difference. We run an *adaptation* of the k-means clustering algorithm on the traffic distributions: instead of minimizing the L1 distances within a cluster c_i, we weighten the distance between its cluster centroid $\mathbf{A}_i$ and a destination d according to d's total traffic. Thus, the computed clusters accurately resemble high-traffic destinations. Cluster centroids $\mathbf{A}_i$ and the clustering error ε are defined as follows.

$$\mathbf{A}_i = \sum_{d \in c_i} \mathbf{M}_d \qquad \varepsilon = \frac{1}{2} \sum_{c_i \in C} \sum_{d \in c_i} |\mathbf{M}_d| \cdot \left| \frac{\mathbf{M}_d}{|\mathbf{M}_d|} - \frac{\mathbf{A}_i}{|\mathbf{A}_i|} \right|. \quad (2)$$

Finally, Velo constructs the approximate traffic matrix $\mathbf{A} = [\mathbf{A}_1, \mathbf{A}_2, \cdots]$ directly from the cluster centroids.

Our clustering approach guarantees that the approximation error δ is bounded by the clustering error ε. Intuitively, this is because *(i)* ε is the amount of traffic in $\mathbf{M}$ that the approximation $\mathbf{A}$ does not account for, and *(ii)* ε is also the amount of additional traffic in $\mathbf{A}$ that is not part of the original matrix $\mathbf{M}$. Consequently, $max\,load(e, \mathbf{A})$ can differ from $max\,load(e, \mathbf{M})$ by at most ε. We provide a formal proof in Appendix B.

6 Evaluation

We now evaluate Velo's performance in terms of scalability (running time) and accuracy. Its correctness is instead guaranteed by our theorems in §4 and §5.

In §6.1, we discuss aspects that influence Velo's running time. We show that Velo can find the maximum loads for all the links in large networks (with close to 2 000 links) within a few hours, considering up to two simultaneous failures and up to ten changes of any external routes. As baseline, we also compare Velo against its closest related work, QARC [44], in scenarios where external routes do not change. We find that Velo runs much faster than QARC for scenarios considering up to two simultaneous failures (the most practical ones).

In §6.2, we evaluate the impact of the traffic matrix on Velo's accuracy. We show that Velo's approximation error is less than 1% even for unrealistic traffic matrices that are hard to approximate.

Implementation We implement a prototype of Velo in $\approx$ 7 000 lines of Rust,[2] including an optimized version of both Algorithms 1 and 2, and our modified k-means clustering algorithm. We run the evaluation on a server with 96 CPU threads and 384 GB of memory.

Networks We study the 75 largest networks from Topology-Zoo [23], ranging from 40 to 754 nodes with 80 to 1 790 links. For each network, we study the effect of increasing the number of border routers by randomly connecting external peers that can advertise a route for any destination to internal routers. We assign random IGP weights (between 10 and 100) to each link. To study the impact of non-shortest paths routing, we setup exception paths (i.e., traffic-engineered paths that do not follow the shortest paths) for given destinations by computing alternative paths using permuted link weights.

Traffic matrices We obtain real traffic matrices from a research network. We also generate synthetic traffic matrices following the gravity model [9,31]. This enables us to control and vary traffic characteristics to evaluate Velo's sensitivity to different traffic patterns.

6.1 Running Time

The following factors affect Velo's running time: network size, number of simultaneous failures l, maximum number of external route changes k, number of clusters $|C|$, number of border routers $|N_B|$, and number of exception paths $|\rho|$. The specification of traffic additional to the traffic matrix instead has no noticeable impact, consistently with the inexpensive approach to deal with it in Velo (see §2).

[2]Available at `https://github.com/nsg-ethz/velo` (GPLv3 license).

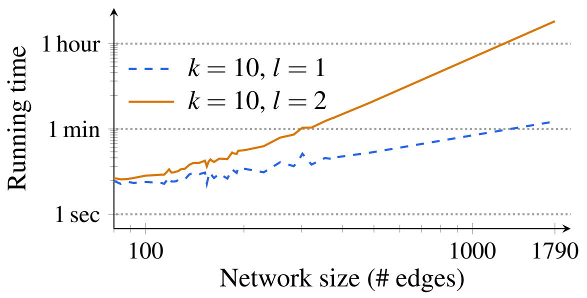


Figure 4: Velo can scale to large networks of over 1 000 links while exploring up to two simultaneous link failures. This log-log plot shows the running time of Velo for the 75 largest topologies in TopologyZoo with up to one and two simultaneous link failures l running on 96 parallel threads.

We find that the network size and the number of failures have the biggest impact on Velo's running time. We therefore divide our experiments in two sets: a fist set considering only these two factors, and another set considering the others.

Following these experiments, we compare Velo's running time against QARC [44], a state-of-the-art approach to find link load violations under varying scenarios, albeit assuming fixed external routes.

Network size and link failures In the first set of experiments, we analyze the impact of network size and maximum number of failures on Velo's efficiency across all the topologies in our dataset. We set a relatively low number of simultaneous failures ($l \leq 2$) and route changes ($k = 10$), as it is unlikely for many more links to fail and many more external routes to change at the same time [42]. We also set $|C|$ to 300, which empirically provides a good accuracy-performance tradeoff (see Appendix D.3). We finally configure 30 border routers per topology, and no exception path.

Figure 4 shows a log-log plot of Velo's running time. Varying the network size from 100 to 1 790 increases the running time by about four orders of magnitude. A similar effect is caused by considering all combinations of two link failures ($l = 2$) rather than single link failures ($l = 1$).

Yet, our results show that Velo is practical for all the TopologyZoo networks. For the largest one, Velo finds the worst-case load for all its 1 790 links within one minute for single link failures, and within three hours for two link failures. For all the other networks, Velo finds all the worst-case loads for $l \leq 2$ within two minutes.

We further break down Velo's efficiency according to its two main phases: clustering and analysis. The clustering phase dominates the running time for small networks (up to 200 edges), while the analysis phase is the dominant one for bigger networks. The asymptotic growth of the analysis time, and hence of the overall running time, is roughly cubic for $l = 2$. Full results are shown in Appendix D.2.

Finally, we approximate the running time improvement gained from reducing the search space of route changes. With 30 border routers, the naive approach requires exploring 2^{30} states—seven orders of magnitudes bigger than Velo's search space. Analyzing a network of only 80 links without the state reduction would take roughly one year for $l = 2$.

Other factors To measure the impact of the other factors, we focus on Cogent, one of the largest networks in Topology Zoo, and vary each factor, one at a time.

The number of route changes k has a small effect on Velo's efficiency. This is because for each link, we always need to compute the worst-case routing input for *all* destinations, before picking the k destinations causing the highest load on that link. Increasing k from 10 to 100 still increases the running by 1.3× but only because the algorithm must maintain larger heaps. We however note that allowing all routing inputs to change (i.e., $k = |P|$) *reduces* the running time by 2.9×, as Velo needs to maintain less state.

Increasing the number of clusters $|C|$ affects the running time, as Algorithms 1 or 2 are called for each cluster. Increasing $|C|$ from 300 to 600 or 1 000 increases the running time by 1.3× or 1.6×, respectively. Similarly, decreasing $|C|$ to 100 improves Velo's performance 1.2×. As a reference, deactivating clustering would take Velo roughly 1 000× longer to complete, assuming around 1 million destinations.

The number of border routers $|N_B|$ affects the number of states to be explored. Yet, increasing $|N_B|$ from 30 to 100 only increases Velo's running time by 1.7×.

Finally, adding exception paths forces Velo to explore additional states. Indeed, Velo must explore all combinations of distinct egress routers of these paths. Such paths are usually configured manually, and they are often used only for traffic-heavy destinations. A total of 30 or 50 exception paths increases the running time by respectively 1.4× or 3.2×.

Comparison to QARC QARC does not support changes in routing inputs. We thus compare Velo against QARC when both systems are asked to analyze link loads under l link failures. We rerun the same experiments as in [44] with the author's implementation. These experiments involves the specification of traffic between all pairs of routers, and 10% additional traffic. For Velo, we extract the traffic matrix by attaching one destination per egress router, and set $k = 0$: this effectively disables Velo's clustering algorithm, since no destinations fall in the same equivalence class (see §5).

The results for up to two simultaneous link failures are shown in Figure 5. Velo is several orders of magnitude faster in computing maximum link loads than QARC, achieving sub-second computation on all topologies. We believe that such a performance gap is mainly due to the different approaches at the core of the two systems: QARC solves an Integer Linear Program, while Velo relies on Dijkstra's algorithm.

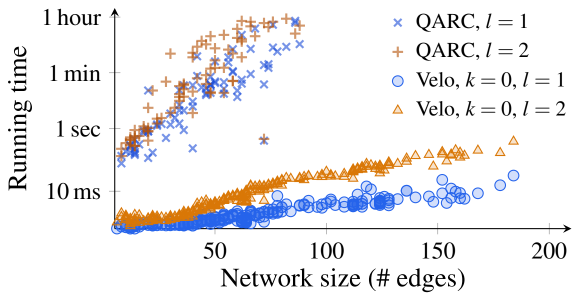


Figure 5: For small $l < 3$, Velo outperforms QARC when comparing only the current state, i.e., with $k = 0$. This figure compares the running time of Velo and QARC [44] to find link load violations for networks from TopologyZoo.

Velo is 10-100× faster than QARC even for $l = 3$ and $l = 4$ (see Figure 7). However, the performance gap between the two systems seems to progressively decrease when increasing l. We expect QARC to outperform Velo for large values of l.

In fairness, QARC guarantees exact computation of link loads, while Velo theoretically doesn't. However, in the absence of clustering, Velo is fully accurate, except possibly overapproximating additional traffic (see §2.2). This is a rare case that does not occur in these scenarios—Velo has *no approximation error* in any of these experiments.

Takeaways Velo finds the worst-case loads of all links within minutes for the most likely and practical scenarios with up to two link failures and a limited number of route changes. Because it is mostly polynomial in the network size, Velo scales to large networks with close to 2 000 links.

6.2 Accuracy

Velo's accuracy depends on the input traffic matrix, which indeed impacts the effectiveness of our clustering. We evaluate Velo's accuracy on both real inter-domain traffic data and synthetic traffic matrices, using the latter ones to measure the effects of traffic characteristics on our clustering algorithm.

We measure both the error bound ε as defined in Equation (2) and the approximation error δ from Equation (1), i.e., maximum difference in worst-case link loads between the approximate and the original traffic matrix.

We further compare our clustering algorithm with the simpler approach of picking the destinations carrying the most traffic, in the following called *Top-X*. We determine how many such destinations are needed to provide the same guarantees as Velo. That is, we find the smallest X such that the amount of traffic disregarded by Top-X equals our error bound ε. We report the ratio between this smallest X and $|C|$: such ratio relates directly to Velo's running time, given that its efficiency scales linearly with the analyzed destinations.

We represent real and synthetic traffic matrices according to the gravity model, a common technique in network traffic synthesis [38]. In the gravity model, a traffic matrix is parametrized by three distributions:

$$\mathbf{M}_{s,d} \propto repulsion(s) \cdot attraction(d) \cdot friction(s,d).$$

The *attraction* describes the distribution of the traffic towards the destinations: a heavy-tailed attraction results in a few destinations attracting most traffic. The *repulsion* describes the distribution of traffic originating from each ingress, symmetrically to the attraction. The *friction* describes traffic similarity across source-destination pairs: a heavily tailed friction yields a sparse matrix, where many entries are zero.

Related work has shown that all the three distributions are heavy tailed in practice [31]. Our real measurements confirm this: traffic per destination (i.e., the attraction), traffic per source (i.e., the repulsion), and traffic similarity across destinations (i.e., friction) are similar to LogNormal distributions. We thus characterize traffic matrices according to the σ parameter of attraction, repulsion and friction: σ indeed measures the skeweness of LogNormal distributions. We provide more details in Appendix C.1.

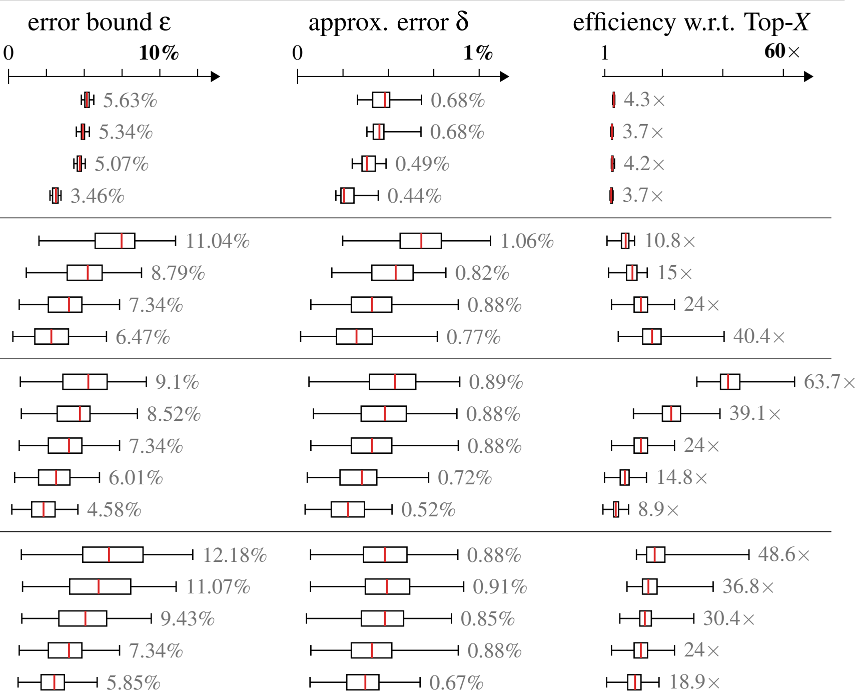
In all experiments, we pick $|C| = 300$ clusters, as we find this to be a good compromise between accuracy and running time, but we evaluate different values of $|C|$ in Appendix D.3.

Real traffic data We obtain Netflow data for four 5-minute slices from the Swiss research network Switch. These measurements include byte counters per ingress link and per destination IP, aggregated by /24 IPv4 or /48 IPv6 prefixes. The data is extremely skewed; 95% of all traffic is towards 0.23% of all destinations. We stress that this does not seem to be common in the Internet [11, 31]. For example, the top 15% destinations are reported to account for 95% of the traffic in the real traffic matrices studied in [31]. This difference might be an artifact of the destination granularity (i.e., we cannot reconstruct traffic for prefixes less specific than /24), or due to the specificities of the research network originating our data.

Both addresses and ingress links are anonymized, forcing us to randomly assign links to routers. We analyze 100 such random assignments to confirm that this has little impact on the results. We refer to Appendix C.2 for more details.

The first four rows in Table 1 show the results for all such traffic matrices. Velo achieves an error bound ε of around 5% and an approximation error δ of around 0.5%. Appendix D.4 expands on the relationship between ε and δ. Further, Velo is ≈ 3.4× more efficient that top-X despite the extreme skeweness of the data; a property that favors Top-X.

Synthetic data Based on our real traffic matrices, we synthesize matrices with similar characteristics using the gravity model with 30 equivalence classes, cf. §5. Starting from the average parameters of the real matrices, we evaluate how



real	repulsion	attraction	friction	error bound ε	approx. error δ	efficiency w.r.t. Top-X
yes	**2.79**	**2.99**	**3.99**	5.63%	0.68%	4.3×
yes	**2.82**	**2.93**	**3.96**	5.34%	0.68%	3.7×
yes	**2.79**	**3.05**	**4.03**	5.07%	0.49%	4.2×
yes	**3.16**	**2.91**	**4.21**	3.46%	0.44%	3.7×
no	**2.03**	3.01	4.02	11.04%	1.06%	10.8×
no	**2.53**	3.01	4.03	8.79%	0.82%	15×
no	**3.00**	3.00	4.01	7.34%	0.88%	24×
no	**3.47**	2.99	4.01	6.47%	0.77%	40.4×
no	2.97	**1.98**	4.02	9.1%	0.89%	63.7×
no	2.98	**2.50**	4.01	8.52%	0.88%	39.1×
no	3.00	**3.00**	4.01	7.34%	0.88%	24×
no	3.02	**3.51**	4.01	6.01%	0.72%	14.8×
no	3.04	**4.01**	4.01	4.58%	0.52%	8.9×
no	2.95	3.00	**2.51**	12.18%	0.88%	48.6×
no	2.96	3.01	**3.01**	11.07%	0.91%	36.8×
no	2.97	3.00	**3.50**	9.43%	0.85%	30.4×
no	3.00	3.00	**4.01**	7.34%	0.88%	24×
no	3.01	3.00	**4.51**	5.85%	0.67%	18.9×

Table 1: Velo's accuracy is affected by the tail distribution over per-destination traffic, i.e., the attraction, and the variation between traffic distributions, i.e., the friction. This table shows both the error bound ε and the approximation error δ, the latter on a smaller scale. The box represents the 25 and 75 percentiles, with the mean indicated by the red line. The whiskers show the minimum and maximum values. The number right to the upper whisker show the maximum number.

changing the skewness in all three dimensions affects Velo's accuracy, one at a time. We do so for six large topologies from TopologyZoo (with 140–200 nodes). For each topology and set of parameters, we generate 20 random traffic matrices.

Table 1 demonstrates that clustering is an excellent fit for approximating traffic matrices. The error bound ε is mostly below 10%, and the actual approximation error δ is always $<1\%$. Clustering is also 5x-50x more efficient than Top-X.

As expected, decreasing traffic skeweness (for any parameter) increases both ε and δ, because it causes traffic to be more evenly spread across more sources or destinations, making it harder to capture all traffic with a fixed amount of clusters. Yet, $|C| = 300$ clusters accurately approximate unrealistic traffic matrices where traffic distributions are not very skewed at all. In contrast, Top-X needs a lot more destinations to achieve the same error bound, especially if the attraction has a less significant tail. We conclude that Velo's clustering is more effective and more general than Top-X.

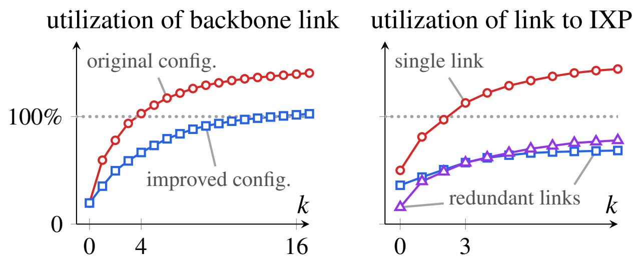
The number of clusters $|C|$ also affects the accuracy of Velo, reducing the error when increasing $|C|$. Doing so, however, also increases the running time, as discussed in §6.1. We find that Velo with $|C| = 300$ is sufficiently accurate for all matrices we evaluated, and increasing it to 1000 clusters only yields minor gains in accuracy (reducing the error bound from ≈3.5% to ≈1.8%). Appendix D.3 shows more results.

Traffic Variability We finally evaluate Velo's accuracy when traffic volumes additional to the traffic matrix is given as input (see §2). Across all the experiments described in Figure 4, Velo computes the correct maximum link loads in over 99.9% of all links, for any value of additional traffic.

Takeaways Velo is very accurate in approximating the worst-case link loads, even when using only $|C| = 300$ clusters. In all our experiments, Velo's approximation error is below 1% of the total traffic volume. Further, Velo guarantees error bounds below 12% even for unrealistic traffic matrices that are challenging to approximate.

7 Case Study

We experiment with an ISP that provides connectivity to around 100 universities and research institutions. The ISP manages 126 routers connected by 376 links. Ten routers are connected to either providers or IXPs. We generate a traffic matrix with 100 000 prefixes according to the gravity model [32] to match reported characteristics [31]. Customers only announce their respective prefixes, while all the other prefixes are reachable through neighboring providers or IXPs. In the initial network state, all links are utilized below 50%—the ISP upgrades links that maintain load above that threshold.



(a) Configuration fine-tuning (b) Network upgrade

Figure 6: Our case study demonstrates that Velo helps improving network robustness.

We run Velo on the relevant parts of the router configurations, the generated traffic matrix, and an increasing k. Within a few seconds, Velo finds that the current configuration and topology is worryingly fragile to specific egress changes: two links would exceed their capacities if egresses of only four destinations change. We now show three ways in which Velo's insight can be leveraged by operators.

Robustifying configurations One of the two fragile links is part of the ISP backbone: this link would be congested by specific egress changes for four (large) destinations. We use this insight to assess an alternative configuration with three additional MPLS paths to effectively shift traffic to underutilized links. The alternative configuration would prevent congestion on any link for up to 16 arbitrary egress changes – a much less likely scenario than 4 egress changes. Figure 6a compares the robustness of the initial and alternative configurations.

Aiding strategic decisions The second fragile link connects the ISP to an IXP. The link would be congested if traffic to just three specific destinations is sent to that IXP. Operators may think about upgrading that link; however, Velo's output highlights that other links within the ISP would also need to be upgraded to sustain the additional traffic. A better alternative is to add a link from the IXP to another, geographically close backbone router. Re-running Velo on the alternative topology reveals that $\approx 1\,000$ prefixes must be advertised at that IXP to congest either of the two links – see Figure 6b.

Velo can also guide operators' peering decisions, predicting possible configuration and infrastructural changes required by new BGP peerings.

Improving network management Velo identifies 64 pairs of prefixes and egress routers; link loads significantly increase if any such prefix is routed via the corresponding egress. This information is highly valuable for network management and operation. For example, it enables monitoring systems to focus only on those 64 pairs of prefixes and egress routers, alerting operators only for the small subset of critical events.

Velo's output can also improve automated management systems. For example, it enables traffic engineering systems (e.g., [8, 18, 20]) to optimize paths *before* link load requirements are violated. Consider the above case of the backbone link congested after four BGP egress changes. When any three egress changes are observed out of these four, an emergency recomputation of internal paths can be triggered immediately, *preventing* congestion without waiting for data-plane measurements typically collected every few minutes [20].

8 Related Work

Velo focuses on verifying link load properties, e.g., to help improving network robustness to external events. As such, it is related to prior work in network verification and routing.

Network verification Many verification systems have recently been proposed. Among them, data-plane ones [12, 21, 22, 43] target properties based on packet headers for fixed forwarding states, basically ignoring traffic and routing.

Control-plane verification systems [1, 4, 5, 14, 15, 45, 46, 54] are more related to Velo, as they admit different network settings (e.g., failures). Almost all of them however focus on properties, such as forwarding paths and routes' propagation, that cannot be directly translated into link loads. Prior works cannot be easily extended to verify link load properties, because they verify one destination at the time. As failures break the independence of destinations, thousands of destinations must be considered jointly for link load properties. Doing so with prior approaches would simply not scale: for techniques like [15], it would inflate SMT formulas by $|P|\times$, where P is the number of destinations, hence hundreds of thousands.

The few existing checkers of link load properties *do not support BGP route changes*. Jingubang [26] is a flow-level simulation tool that computes link loads for specific failure scenarios and traffic matrices. YU [27] extends the above simulator to deal with k arbitrary link failures: it still works at the per-flow level. NetDice [42] implements probabilistic verification of properties including link load ones; however, for link load properties, NetDice's approach does not scale beyond a dozen ingress-destination pairs [42]. The framework presented in [7] allows to check link utilization under arbitrary failures for network designs where configurations (e.g., routing) can be changed. Our closest related work is QARC [44], that verifies link load *violations* (without computing all link loads) upon link failures. We experimentally compare Velo against QARC in §6.1.

Routing systems Optimizing link loads has been the focus of a large body of work [2, 30, 51], especially in the area of traffic engineering (TE). TE approaches typically optimize paths based on the *current* traffic and BGP routes or, at best, variations of them [17, 25, 35]. Some of these approaches [7, 28, 52] rely on robust optimization to *approximate* the impact of failures. Future work may try robust optimization techniques for link-load verification, although it's unclear how these techniques would guarantee accuracy and scalability when analyzing both failures and route changes.

Real-world TE systems [18, 20] update forwarding upon detecting a failure, and react to BGP route changes by re-optimizing paths according to load measurements collected every few minutes. By anticipating the most impactful failures and route changes, Velo can be integrated into TE systems to prevent excessive link loads or react faster to them, see §7.

Information on current and future traffic is key for intra-domain traffic engineering. Recently, systems (e.g., [29]) have been proposed to predict changes in incoming traffic caused by specific BGP routes *announced to* external neighbors. We envision that such systems can be used by operators to compute traffic matrices they want to verify with Velo.

9 Discussion

In this section, we discuss some of the design choices we made in Velo alongside the generality of the approach.

Why *not* modeling congestion? For each link, Velo computes the load by simply summing the amount of traffic routed over it. In practice, though, no link load could exceed its capacity: traffic would be dropped instead. This also means that the amount of traffic forwarded through the network immediately decreases for the links downstream of the congested one. Hence, when any link is congested, the link loads reported by Velo do not reflect the actual traffic volumes crossing links.

Despite this simplification, Velo's approach is practical for at least two reasons. First, Velo correctly identifies *all* the links that cannot sustain the input traffic. Second, it correctly estimates the amount of exceeding traffic, and it therefore helps quantifying how much traffic needs to be rerouted, or by how much links' capacity should be upgraded.

What protocols and features can Velo model? Despite hiding most of the complexity of BGP and intra-domain routing protocols, Velo's model is powerful enough to capture common routing architectures, including iBGP and SDN (see §3). In fact, our model only imposes two constraints on intra-domain routing. First, all routers must be able to select any of the egress routers, which means that there must be no iBGP visibility problems. Several existing techniques avoid these problems [36, 47, 48]. Second, destinations must be independent, which requires the de-activation of intra-domain routing features that tie different destinations together, such as route aggregation or conditional advertisements. The use of these features is discouraged in iBGP [37]. Additionally, we think that Velo can be extended to jointly reason about egresses of dependent destinations, if needed.

Data-plane filters like ACLs do not influence routing, and thus, Theorem 1 still applies. In their presence, Algorithms 1 and 2 can be easily extended to apply such filters by dropping (some) traffic instead of propagating it. Doing so would ensure that the worst-case link loads computed by Velo would be consistent with both routing inputs and data-plane filters.

What route changes does Velo support? Velo computes the worst-case link loads for route changes affecting subsets of destinations. In §3, we have defined destinations as the IP prefixes in the routing tables. This allows Velo to cover BGP route changes for any of these IP prefixes.

In reality, the set of IP prefixes learned via BGP can change over time. For example, at any time, the remote network owning an IP prefix, say 1.0.0.0/8, can start announcing a more specific prefix, such as 1.0.0.0/16. If this happens, BGP propagates two distinct and independent routes – e.g., one for 1.0.0.0/8 and the other for 1.0.0.0/16.

Since Velo learns the destinations from the input traffic matrix, it supports some analyses of route changes affecting the set of known destinations, provided that enough information is specified by operators. For instance, Velo can handle the previous example if the input matrix includes traffic volumes for 1.0.0.0/16 and (separately) for the other subnets in 1.0.0.0/8. This information is anyway necessary to compute link loads with distinct routes for 1.0.0.0/16 and 1.0.0.0/8.

In contrast, Velo does not currently support systematic explorations of route changes affecting the set of destinations, e.g., announcements of *any* possible sub-prefix of a current destination. We leave this direction for future work.

10 Conclusions

Velo demonstrates the feasibility of verifying worst-case link loads in large networks, despite the need to navigate a gigantic space of possible failures and route changes. Our evaluation shows that Velo is both scalable and accurate across a wide range of topologies, settings, constraints and traffic matrices.

Velo focuses on maximum link loads due to their straightforward practical relevance in assessing network performance. As our case study demonstrates, Velo readily supports operators in complex tasks ranging from improving configurations to network design and aiding business decisions.

We also see Velo as a stepping stone towards the broader goal of verifying network performance. We believe that exploring other performance aspects, such as delay or bandwidth guarantees, is an interesting avenue for future research.

Acknowledgments The research leading to these results was supported by an ERC Starting Grant (SyNET) 851809. We thank João Luís Sobrinho, our shepherd Siva Kakarla and all anonymous reviewers for their valuable feedback. We also thank Simon Leinen and Switch for providing traffic data.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, 2020.

[2] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71:1–30, 2014.

[3] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Dr. Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209, Internet Engineering Task Force, December 2001.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 476–489, 2018.

[6] Vineet Bharti, Pankaj Kankar, Lokesh Setia, Gonca Gürsun, Anukool Lakhina, and Mark Crovella. Inferring invisible traffic. In *Proceedings of the 6th International Conference*, Co-NEXT '10, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust Validation of Network Designs under Uncertain Demands and Failures. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 347–362, Boston, MA, March 2017. USENIX Association.

[8] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. Mate: Mpls adaptive traffic engineering. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1300–1309. IEEE, 2001.

[9] Wenjia Fang and Larry Peterson. Inter-as traffic patterns and their implications. In *Seamless Interconnection for Universal Services. Global Telecommunications Conference. GLOBECOM'99.(Cat. No. 99CH37042)*, volume 3, pages 1859–1868. IEEE, 1999.

[10] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. The case for separating routing from routers. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, page 5–12, New York, NY, USA, 2004. Association for Computing Machinery.

[11] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions On Networking*, 9(3):265–279, 2001.

[12] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, 2015.

[13] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *Proceedings IEEE INFOCOM 2000. conference on computer communications. Nineteenth annual joint conference of the IEEE computer and communications societies (Cat. No. 00CH37064)*, volume 2, pages 519–528. IEEE, 2000.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 300–313, 2016.

[15] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. Nv: An intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 958–973, 2020.

[16] Timothy G Griffin and Gordon Wilfong. On the correctness of ibgp configuration. *ACM SIGCOMM Computer Communication Review*, 32(4):17–29, 2002.

[17] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *Proceedings of the ACM SIGCOMM 2018 Conference*, page 88–102, 2018.

[18] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[19] Geoff Huston. Measuring BGP in 2023—have we reached peak ipv4? APNIC Blog, Jan 2024.

[20] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally Deployed Software Defined WAN. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013.

[21] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, 2012.

[22] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 49–54, 2012.

[23] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[24] Gottfried Köthe and Gottfried Köthe. *Topological vector spaces*. Springer, 1983.

[25] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, 2018.

[26] Ruihan Li, Fangdan Ye, Yifei Yuan, Ruizhen Yang, Bingchuan Tian, Tianchen Guo, Hao Wu, Xiaobo Zhu, Zhongyu Guan, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, , and Ennan Zhai. Reasoning about network traffic load property at production scale. In *21st USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[27] Ruihan Li, Yifei Yuan, Fangdan Ye, Mengqi Liu, Ruizhen Yang, Yang Yu, Tianchen Guo, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, and Ennan Zhai. A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures. In *Proceedings of the ACM SIGCOMM 2024 Conference*, page 228–243, New York, NY, USA, 2024. Association for Computing Machinery.

[28] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, page 527–538, New York, NY, USA, 2014. Association for Computing Machinery.

[29] Michael Markovitch, Sharad Agarwal, Rodrigo Fonseca, Ryan Beckett, Chuanji Zhang, Irena Atov, and Somesh Chaturmohta. TIPSY: predicting where traffic will ingress a WAN. In *Proceedings of the ACM SIGCOMM 2022 Conference*, page 233–249, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Alaitz Mendiola, Jasone Astorga, Eduardo Jacob, and Marivi Higuero. A survey on the contributions of software-defined networking to traffic engineering. *IEEE Communications Surveys & Tutorials*, 19(2):918–953, 2016.

[31] Jakub Mikians, Amogh Dhamdhere, Constantine Dovrolis, Pere Barlet-Ros, and Josep Solé-Pareta. Towards a statistical characterization of the interdomain traffic matrix. In *NETWORKING 2012: 11th International IFIP TC 6 Networking Conference, Prague, Czech Republic, May 21-25, 2012, Proceedings, Part II 11*, pages 111–123. Springer, 2012.

[32] Jakub Mikians, Nikolaos Laoutaris, Amogh Dhamdhere, and Pere Barlet-Ros. Itmgen—a first-principles approach to generating synthetic interdomain traffic matrices. In *2013 IEEE International Conference on Communications (ICC)*, pages 2507–2512. IEEE, 2013.

[33] John Moy. OSPF Version 2. RFC 2328, Internet Engineering Task Force, April 1998.

[34] Dave Oran. OSI IS-IS Intra-domain Routing Protocol. RFC 1142, Internet Engineering Task Force, February 1990.

[35] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. {DOTE}: Rethinking (predictive){WAN} traffic engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1557–1581, 2023.

[36] Robert Raszuk, Bruno Decraene, Christian Cassar, Erik Aman, and Kevin Wang. BGP Optimal Route Reflection (BGP ORR). RFC 9107, Internet Engineering Task Force, August 2021.

[37] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Internet Engineering Task Force, January 2006.

[38] Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35(5):93–96, 2005.

[39] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: Synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 33 – 49, New York, NY, 2021-08. Association for Computing Machinery.

[40] Tibor Schneider, Roland Schmid, Stefano Vissicchio, and Laurent Vanbever. Taming the transient while reconfiguring BGP. In *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.

[41] João L Sobrinho. Algebra and algorithms for qos path computation and hop-by-hop routing in the internet. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society*, volume 2, pages 727–735. IEEE, 2001.

[42] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Probabilistic verification of network configurations. In *Proceedings of the Annual ACM SIGCOMM Conference of the ACM Special Interest Group on Data Communication*, Online, 2020. Association for Computing Machinery.

[43] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 314–327, 2016.

[44] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. Detecting network load violations for distributed control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 974–988, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 94–107, 2023.

[46] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Kirigami, the verifiable art of network cutting. *IEEE/ACM Transactions on Networking*, 2024.

[47] Jim Uttaro, Pierre Francois, Keyur Patel, Jeffrey Haas, Adam Simpson, and Roberto Fragassi. Best Practices for Advertisement of Multiple Paths in IBGP. Internet-Draft draft-ietf-idr-add-paths-guidelines-08, Internet Engineering Task Force, April 2016. Work in Progress.

[48] Virginie Van den Schrieck, Pierre Francois, and Olivier Bonaventure. Bgp add-paths: the scaling/performance tradeoffs. *IEEE Journal on Selected Areas in Communications*, 28(8):1299–1307, 2010.

[49] Stefano Vissicchio, Luca Cittadini, Laurent Vanbever, and Olivier Bonaventure. iBGP deceptions: More sessions, fewer routes. In *Proceedings of the IEEE INFOCOM Conference*, 2012.

[50] Dan Wang, Peng Zhang, and Aaron Gember-Jacobson. Expresso: Comprehensively Reasoning About External Routes Using Symbolic Simulation. In *Proceedings of the ACM SIGCOMM 2024 Conference*, page 197–212, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Ning Wang, Kin Hon Ho, George Pavlou, and Michael Howarth. An overview of routing optimization for internet traffic engineering. *IEEE Communications Surveys & Tutorials*, 10(1):36–56, 2008.

[52] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. R3: resilient routing reconfiguration. In *Proceedings of the ACM SIGCOMM 2010 Conference*, page 291–302, New York, NY, USA, 2010. Association for Computing Machinery.

[53] Fatai Zhang, Oscar Gonzalez de Dios, Matt Hartley, Zafar Ali, and Cyril Margaria. RSVP-TE Extensions for Collecting Shared Risk Link Group (SRLG) Information. RFC 8001, Internet Engineering Task Force, January 2017.

[54] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic router execution. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 336–349, 2022.

A Exception Paths

In the following, we first proof Theorem 2 that pretains to we handle exception paths. We then present the algorithm that relies on Theorem 2 to find the worst-case link loads under optimal and exception paths.

A.1 Proof of Theorem 2

Proof. Consider edge $e = (u, v)$, and let $b \in B$ be an egress router preferred by v. Now, take any node $n \in N_e$ that forwards traffic along the edge e for destination d. We distinguish two cases:

1. Paths of n that traverse e are optimal (i.e., are not exception paths), and by Theorem 1, n will not decrease the traffic it sends along edge e.
2. n forwards traffic along exception paths in ρ_d. Then, n will use the same forwarding paths for egresses $\{b\} \cup T \cap B$ as all destinations of n's forwarding paths are within T and B. Thus, n will still forward the same amount of traffic over link e.

It follows that the traffic on e cannot decrease for egresses $\{b\} \cup T \cap B$, proving Theorem 2. □

A.2 Algorithm for Exception Paths

Algorithm 2 shows the algorithm for finding the worst-case loads for all network links in the presence of exception paths, i.e., paths that don't follow the optimal paths according to the strictly isotone routing protocol.

The function `selected_combinations` iterates over all combinations of egress routers within T and all individual routers in $N_B \setminus T$. It further, checks whether there exists a set of advertisements for all egresses to be selected simultaneously based on the `LocalPref` values.

Algorithm 2: Find the worst-case link load for all links for optimal or exception paths.

```
Data: Graph $G = (N, E)$, border routers $N_B \subseteq N$, link
      weights $w : E \mapsto \mathbb{R}$, traffic $\mathbf{M}_d$, paths $\rho_d$
Result: maximum link loads $y[e]$ for all links $e \in E$
$T \leftarrow \{b \mid (s, \cdots, b) \in \rho_d\}$
$y[u, v] \leftarrow 0 \; \forall (u, v) \in E$
$G_{dag}[b] \leftarrow ForwardingDAG(G, w, root = b) \; \forall b \in N_B$
for $B \in$ selected_combinations$(N_B, T)$ do
    $y'[u, v] \leftarrow 0 \; \forall (u, v) \in E$
    $l[s] \leftarrow \mathbf{M}_{s,d} \; \forall s \in N$
    $G_{fw} \leftarrow merge(G_{dag}[b] \; \forall b \in B)$
    for $path = (s, n_1, \cdots, n_i, b) \in \rho_d$ do
        if $s$ forwards to $b$ in $G_{fw}$ then
            $y'[u, v] \leftarrow y'[u, v] + l[s] \; \forall (u, v) \in path$
            $l[s] \leftarrow 0$
    for $u \in TopoSort(G_{dag})$ do
        $d_{out} \leftarrow |out(G_{dag}, u)|$
        for $v \in out(G_{dag}, u)$ do
            $l[v] \leftarrow l[v] + l[u]/d_{out}$
            $y'[u, v] \leftarrow y'[u, v] + l[u]/d_{out}$
            $y[u, v] \leftarrow \max(y[u, v], y'[u, v])$
```

B Clustering Error Bounds

In the following, we formally proof that the clustering error bounds the approximation error of the worst-case link load. To that end, we first introdoce two lemmas the pertain to linearity and the superposition of maximum link loads before proofing Theorem 3.

Lemma 1 *max load*$(e, \mathbf{M})$ *is linear in terms of* $\mathbf{M}$*, i.e.,*

$$\forall a \geq 0 : a \cdot max\,load(e, \mathbf{M}) = max\,load(e, a \cdot \mathbf{M})$$

Proof. Intuitively, Lemma 1 holds, because scaling the traffic matrix does not affect the worst-case routing state, and thus, the forwarding paths in the worst case and the link loads remain unchanged. More formally, the worst-case routing state is obtained using a set of inequalities over sums of traffic volumes (see Algorithm 1). Scaling each entry in the traffic matrix does not influence these inequalities. The link loads in that routing state are computed by adding individual entries in the matrix, which itself is a linear operation. Thus, *max load* is linear in terms of $\mathbf{M}$. □

Lemma 2 *max load*$(e, \mathbf{M})$ *is equal the superposition of the maximum link loads for all columns of* $\mathbf{M}$*. In other words,*

$$max\,load(e, \mathbf{M}) = \sum_d max\,load(e, \mathbf{M}_d)$$

Proof. Lemma 2 is intuitively true because the forwarding decision for one destination is independent of the routing state of another destination. In other words, we can tweak the worst-case routing input for destination d_1 independently of d_2 to obtain the maximum load on link e for destinations d_1 and d_2. Then, the load on link e is equal to the sum of all traffic for all individual destinations, including d_1 and d_2. □

Theorem 3 *The approximation error* δ *of computing the maximum link loads for the approximated traffic matrix* $\mathbf{A}$ *is bounded by* ε*. More formally,*

$$\delta = \max_{e \in E} |max\,load(e, \mathbf{M}) - max\,load(e, \mathbf{A})| \leq \varepsilon. \quad (3)$$

Proof. We prove Theorem 3 by considering each individual destination d and its contribution to both the clustering and the approximation error. To that end, we define $\overline{\mathbf{M}}_d$ as the approximation of d's traffic $\mathbf{M}_d$, and express both the approximation error δ and the clustering error ε in terms of $\overline{\mathbf{M}}$.

Since the approximated matrix $\mathbf{A}$ is of a different shape than $\mathbf{M}$, we first describe how to relate the contribution of any destination $d \in c_i$ to its cluster centroid $\mathbf{A}_i$. To that end, we define $\overline{\mathbf{M}}_d = |\mathbf{M}_d| \cdot \mathbf{A}_i / |\mathbf{A}_i|$ as the approximated traffic for destination d. Notice, that the clustering error ε, defined in Equation (2), can be rewritten as:

$$\varepsilon = \frac{1}{2} \sum_{c_i \in C} \sum_{d \in c_i} \left| \mathbf{M}_d - |\mathbf{M}_d| \frac{\mathbf{A}_i}{|\mathbf{A}_i|} \right| = \frac{1}{2} \sum_{d \in P} \left| \mathbf{M}_d - \overline{\mathbf{M}}_d \right|.$$

Further, the maximum link load $max\,load(e, \mathbf{A})$ can expressed in terms of $\overline{\mathbf{M}}$ by relying on the definition of $\mathbf{A}_i$ in Equation (2), and the linearity and decomposition of maximum link loads. We prove both properties in Appendix B.

$$\begin{aligned} max\,load(e, \mathbf{A}) &= \sum_{c_i \in C} max\,load(e, \mathbf{A}_i) \\ &= \sum_{c_i \in C} \sum_{d \in c_i} max\,load(e, \mathbf{A}_i) \cdot |\mathbf{M}_d| / |\mathbf{A}_i| \\ &= \sum_{d \in P} max\,load(e, \overline{\mathbf{M}}_d) \end{aligned}$$

Now, let's focus on the maximum link load for $\mathbf{M}_d$ and $\overline{\mathbf{M}}_d$ on link e. To that end, we consider traffic x_d^+ that is part in the approximation $\overline{\mathbf{M}}_d$ but not in the original $\mathbf{M}_d$, and traffic x_d^- of $\mathbf{M}_d$ that is not part in $\overline{\mathbf{M}}_d$:

$$x_d^+ = \sum_s \max(0, \overline{\mathbf{M}}_{s,d} - \mathbf{M}_{s,d}) \quad x_d^- = \sum_s \max(0, \mathbf{M}_{s,d} - \overline{\mathbf{M}}_{s,d})$$

Let's first focus on x_d^+, but the argument for x_d^- is symmetrical by swapping $\mathbf{M}$ and $\overline{\mathbf{M}}$. The difference of maximum link loads for $\overline{\mathbf{M}}_d$ and $\mathbf{M}_d$ is at most the additional traffic x_d^+ of $\overline{\mathbf{M}}_d$, i.e., $max\,load(e, \overline{\mathbf{M}}_d) - max\,load(e, \mathbf{M}_d) \leq x_d^+$. That is because $max\,load(e, \mathbf{M}_d)$ by definition is the maximum load achievable on link e, and adding traffic x_d^+ cannot increase that maximum by more than x_d^+. Therefore, $\delta \leq x_d^+$ and $\delta \leq x_d^-$.

Finally, we show that $\sum_d x_d^+ = \sum_d x_d^- = \varepsilon$. This is because, by definition of the L1 norm, $x_d^+ + x_d^- = \left| \overline{\mathbf{M}}_d - \mathbf{M}_d \right|$, and that the centroid $\mathbf{A}_i = \sum_{d \in c_i} \overline{\mathbf{M}}_d$ contains equal positive and negative errors, i.e., $x_d^+ = x_d^-$. This proves Theorem 3. □

C Methodology for Table 1

The following elaborates on the Methodology for Table 1. Specifically, we detail how we find the attraction, repulsion, and friction parameters from a (real or synthetic) traffic matrix, and our processing that was required to obtain the real traffic matrices from aggregated netflow statistics.

C.1 Characterizing a Traffic Matrix

We characterize a traffic matrix according to the three distributions of the gravity model. Specifically, we assume the matrix was generatred from a gravity model, with repulsion, attraction, and friction sampled independently from LogNormal distributions. We then obtain the most likely σ parameter.

To characterize the attraction, we compute the total traffic to each destionation d, normalized to have a mean of 1:

$$\forall d \in P: \; \overline{\mathbf{M}}_d = \frac{|P|}{|\mathbf{M}|} \sum_s \mathbf{M}_{s,d}.$$

We then assume $\{\overline{\mathbf{M}}_d \mid d \in P\} \sim LogNormal(\sigma, \mu = 0)$ to be sampled from a LogNormal distribution with $\mu = 1$ (the expected value in log-space) and find the most likely σ parameter. Similarly, we repeat the same process for the repulsion, but with traffic from each ingress router:

$$\forall s \in N: \; \overline{\mathbf{M}}_s = \frac{|N|}{|\mathbf{M}|} \sum_d \mathbf{M}_{s,d}.$$

For the friction, we use the traffic from each source and to each destination to find the variance for each element in the matrix. In other words, we find find the most likely σ assuming $\{\overline{f}_{s,d} \mid (s,d) \in N \times P\} \sim LogNormal(\sigma, \mu = 0)$ with

$$f_{s,d} = \frac{\mathbf{M}_{s,d}}{\overline{\mathbf{M}}_s \cdot \overline{\mathbf{M}}_d}, \quad \overline{f}_{s,d} = \frac{|N| \cdot |P|}{\sum_{s,d} f_{s,d}} f_{s,d}.$$

By dividing each element $\mathbf{M}_{s,d}$ by the total traffic from source s to destination d, we are left with the variation factor *friction*(s,d) from the gravity model.

C.2 Real Traffic Matrices

The ISP samples traffic on all links towards external peers and providers, but not on links to their customers. The collected netflow data consists of byte counters aggregated by souce and destination IP addresses (either /24 or /48), and the specific egress link on which the packet was sampled, including its direction (whether it leaves or enters the ISP).

To construct the inter-domain traffic matrix (i.e., byte counters per ingress router and destination prefix), we must associate destination addresses of ingress traffic (and source addresses of egress traffic) to the customer associated with that prefix. Together with the operators, we constructed a mapping from prefixes to customers. Unfortunately, this mapping is neither complete nor unique. Some prefixes are related to multiple customers that did not obtain full /24 or /48 prefixes. Further, some prefixes could not be assigned to customers due to time constraints. This affects around 30% of the total traffic volume[3].

[3]We plan to improve this mapping for the final version.

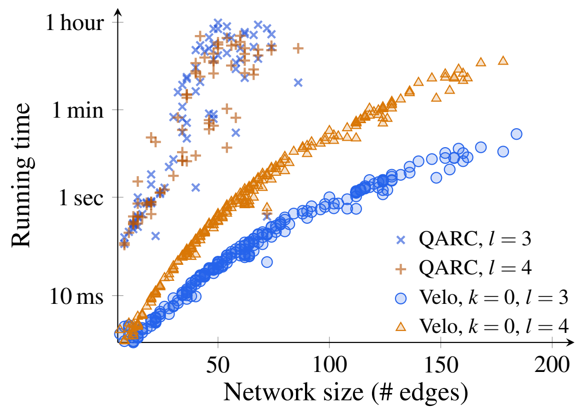


Figure 7: For $l = 3$ and $l = 4$, Velo still outperforms QARC, yet the performance gap decreases.

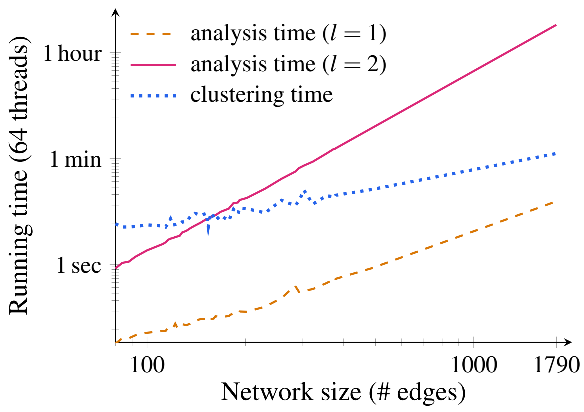


Figure 8: The running time of Velo in terms of the time to perform the clustering and the time to find the worst-case link failures.

The data further allows us to reconstruct the routing state (i.e., the selected egress routers) at time of measurement. For ingress traffic, the traffic is always routed towards the respective customer. For egress traffic, the egress interface on which the traffic is observed is the selected egress point.

D Supplementary evaluation

We provide additional measurements to support §6.

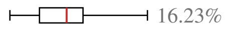
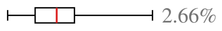
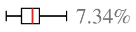
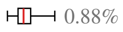
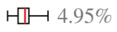
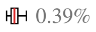
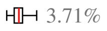
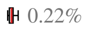
D.1 Comparison with QARC

Figure 7 compares the running time of Velo with QARC for finding worst-case link loads under three and four simultaneous link failures. In 15% of these scenarios, QARC timeouts after running for one hour, while Velo can find the worst-case within 20 minutes for networks up to twice as large.

D.2 Clustering Time vs. Analysis Time

Velo's total running time can be separated into the *clustering time* to find the approximate traffic matrix and the time to find the maximum link loads, which we call the *analysis time*. Figure 8 shows both the clustering and the analysis time for the largest 75 networks on a log-log plot, for both $l = 1$ and $l = 2$. We use the same parameters as described in §6.1. We find that the clustering time dominates the running time for small networks, while it can be neglected for large networks. Further, both the analysis time and the clustering time appear as straight lines in the log-log plot, indicating they scale polynomially in the network size. The analysis time scales roughly quadratically for $l = 1$ and cubic for $l = 2$, as Velo considers $|E|$ additional failure scenarios. Finally, the clustering time scales linearly in the network size which corresponds to the dimensionality of the clustering problem.

Table 2: Velo's accuracy and running time for varying number of clusters

$\|C\|$	time	error bound ε (0 – 20%)	approx. error δ (0 – 3%)
100	178 s	 16.23%	 2.66%
300	210 s	 7.34%	 0.88%
600	268 s	 4.95%	 0.39%
1000	343 s	 3.71%	 0.22%

D.3 Performance–Accuracy Tradeoff

Velo can be configured to an arbitrary number of clusters. Increasing the number of clusters will yield more accurate results at the expense of longer running times, as Algorithm 1 is executed for each cluster.

To evaluate the performance–accuracy tradeoff, we run Velo on the same 6 topologies as in §6.2, choose the traffic matrix using the same baseline distributions, and choose $l = 2$, $k = 10$, and $|N_B| = 30$.

Table 2 shows both Velo's running time and accuracy for 100, 300, 600, and 1 000 clusters. We identify a substantial improvement in accuracy when increasing $|C|$ from 100 to 300, especially in terms of the error bounds. However, increasing $|C|$ to 1000 yields a much smaller accuracy improvement at the expense of a 1.6× increase in running time.

Based on the results in Table 2, we choose $|C|$ as a reasonable tradeoff between accuracy and running time. Yet, the number of clusters can be tuned to the operator's use case to either reduce the running time or improve the accuracy.

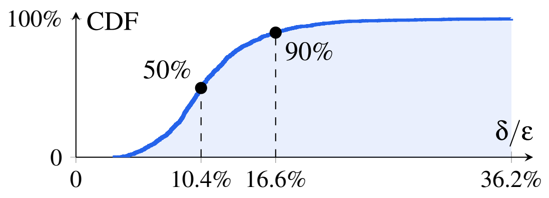


Figure 9: The approximation error δ is typically over 100× smaller than the error bounds ε. This plot shows the CDF of the fraction δ/ε, a measure of how "tight" ε bounds δ.

D.4 Tightness of error bounds.

Table 1 shows that the clustering error ε typically 10 times larger than the approximation error δ. This demonstrates how our error bounds is pessimistic, as it *sums* the absolute values of all the clustering errors as if all the traffic would traverse every single link. In practice, however, differences in a single destination's traffic distribution do not necessarily cause the worst-case routing state for that destination to change. Further, if a few destinations in that cluster have different worst-case routing state, their impact on the resulting load can be both positive or negative, likely canceling each other out.

Figure 9 displays the CDF of the approximation error δ divided by the error bounds ε to visualize how tight the theoretical and real errors are. In 50% of our experiments, the approximation error δ is within 10.4% of the error bounds ε. In 90%, it is within 16.6%.