A Quantitative Theory of Bottleneck Structures for Data Networks

Technical Report, August 2021, New York, USA

Jordi Ros-Giralt, Noah Amsel, Sruthi Yellamraju, James Ezick, Richard Lethin
Yuang Jiang, Aosong Feng, Leandros Tassiulas

contact: jros@qti.qualcomm.com,
1Qualcomm Europe, Inc
2Formerly of Qualcomm Technologies, Inc
3Qualcomm Technologies, Inc
4Yale, University

Abstract—The conventional view of the congestion control problem in data networks is based on the principle that a flow's performance is uniquely determined by the state of its bottleneck link, regardless of the topological properties of the network. However, recent work has shown that the behavior of congestion-controlled networks is better explained by models that account for the interactions between bottleneck links. These interactions are captured by a latent bottleneck structure, a model describing the complex ripple effects that changes in one part of the network exert on the other parts. In this paper, we present a quantitative theory of bottleneck structures (QTBS), a mathematical and engineering framework comprising a family of polynomial-time algorithms that can be used to reason about a wide variety of network optimization problems, including routing, capacity planning and flow control. QTBS can contribute to traffic engineering by making clear predictions about the relative performance of alternative flow routes, and by providing numerical recommendations for the optimal rate settings of traffic shapers. A particularly novel result in the domain of capacity planning indicates that previously established rules for the design of folded-Clos networks are suboptimal when flows are congestion controlled. We show that QTBS can be used to derive the optimal rules for this important class of topologies, and empirically demonstrate the correctness and efficacy of these results using the BBR and Cubic congestion-control algorithms.

1. Introduction

Most research on the problem of congestion control for data networks is based on the principle that the performance of a flow is solely determined by the state of its bottleneck link. This view was presented in the original congestion control algorithm by Jacobson [1], which helped the Internet recover from congestion collapse in 1988, and it persisted throughout the more than 30 years of research and development that followed, including Google’s new BBR algorithm [2]. While it is certainly true that a flow’s performance is limited by the state of its bottleneck link, recent work [3] reveals a deeper view of network behavior, describing how bottlenecks interact with each other through a latent structure—called the bottleneck structure—that depends on the topological, routing and flow control properties of the network. This latent structure explains how the performance of one bottleneck can affect other bottlenecks, and provides a framework to understand how perturbations in the capacity of a link or the rate of a flow propagate through a network, affecting other links and flows.

While [3] introduced the concept of bottleneck structure, the analysis provided was qualitative. In this paper we present a quantitative theory of bottleneck structures (QTBS), a mathematical framework that yields a set of polynomial time algorithms for quantifying the ripple effects of perturbations in a network. Perturbations can either be unintentional (such as the effect of a link failure or the sudden arrival of a large flow in a network) or intentional (such as the upgrade of a network link to a higher capacity or the modification of a route with the goal of optimizing performance). With QTBS, a network operator can quantify the effect of such perturbations and use this information to optimize network performance.

The theoretical contributions of this paper are as follows:

- A new generalized bottleneck structure called gradient graph is studied in detail. A key difference with the bottleneck structure introduced in [3] is that the gradient graph allows us to not only qualify the influences that flows and bottlenecks exert on each other, but also to quantify them. This leads to the development of a quantitative theory of bottleneck structures (QTBS), introduced in this paper. (Section 2.2)
- A novel, fast algorithm to compute the gradient graph is developed. This algorithm constitutes an asymptotic speed-up compared to those presented in [3], allowing us to scale our methodology to large production networks (Sections 2.2)
- The concepts of link and flow gradient are introduced. These mathematical operators quantify the effects of infinitesimally small perturbations in a network, the core building blocks of QTBS. A new, fast method to effi-
ciently compute the gradients by leveraging the bottleneck structure is presented. (Section 2.3)

Applications demonstrating the practical implications of QTBS are provided in the areas of routing, capacity planning and flow control. In each of these applications, we show how QTBS can potentially alter some of the established conventional best practices. Our practical contributions are as follows:

- In the routing application, we introduce an algorithm to find maximal-throughput routes by anticipating the effects of the congestion control algorithm. While in traditional traffic engineering approaches (e.g., [4]), the problems of routing and flow control are considered independently, we show how QTBS can help resolve them jointly, allowing operators to design routes that are efficient from a congestion control standpoint. (Section 3.1)

- In the capacity planning application, we use QTBS to optimize the bandwidth allocation between the spine and leaf links of a fat-tree (also known as folded-Clos [5]). We demonstrate that, due to the effects of congestion control, the optimal design differs from the full-fat-tree configuration proposed by Leiserson [6]. (Section 3.2)

- In the flow control application, we show that QTBS can be used to precisely compute the rate reduction that a set of traffic shapers must impose on the network’s low priority flows in order to achieve a quantifiable positive impact on the high-priority flows. (Section 3.3)

- To demonstrate that networks behave according to QTBS, we carry out experiments for each application we consider using production TCP/IP code and the widely adopted BBR [2] and Cubic [7] congestion control algorithms. (Section 3.4)

2. Quantitative Theory of Bottleneck Structures (QTBS)

2.1. Network Model

In their simplest form, networks can be modeled using two kinds of elements: links, which are communication resources with a limited capacity; and flows, which make use of these communication resources. We formalize the definition of a network as follows:

Definition 1. Network. We say that a tuple \( N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) is a network if:
- \( \mathcal{L} \) is a set of links of the form \( \{l_1, l_2, ..., l_{|\mathcal{L}|}\} \),
- \( \mathcal{F} \) is a set of flows of the form \( \{f_1, f_2, ..., f_{|\mathcal{F}|}\} \), and
- \( c_l \) is the capacity of link \( l \), for all \( l \in \mathcal{L} \).

Each flow \( f \) traverses a subset of links \( \mathcal{L}_f \subset \mathcal{L} \) and, similarly, each link \( l \) is traversed by a subset of flows \( \mathcal{F}_l \subset \mathcal{F} \). Finally, each flow \( f \) transmits data at a rate \( r_f \) and the capacity constraint \( \sum_{f \in \mathcal{F}} r_f \leq c_l \) must hold for all \( l \in \mathcal{L} \).

A core concept upon which our mathematical framework rests is the notion of a bottleneck link. Intuitively, a flow is bottlenecked at a link if bypassing the link would allow its transmission rate to increase. A link whose capacity is fully utilized is always a bottleneck of at least one flow, though not necessarily of all the flows traversing it. In this work, we adopt the following formal definition:

Definition 2. Bottleneck link. Let \( N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) be a network where each flow \( f \in \mathcal{F} \) transmits data at a rate \( r_f \) as determined by a congestion control algorithm (e.g., TCP’s algorithm [1]). We will say that flow \( f \) is bottlenecked at link \( l \)—equivalently, that link \( l \) is a bottleneck of flow \( f \)—if and only if:

- Flow \( f \) traverses link \( l \).
- \( \partial r_f / \partial c_l \neq 0 \). That is, the transmission rate of flow \( f \) changes upon small reductions in link \( l \)'s capacity.

This characterization of bottlenecks is a generalization of some of the classic definitions found in the literature. Unlike previous work, however, it is based on the notion of a perturbation, mathematically expressed as a derivative of a flow rate with respect to the capacity of a link \( \partial r_f / \partial c_l \). As an example to illustrate that our definition of bottleneck is relatively loose, in Appendix A we show that it generalizes the classic max-min definition of Bertsekas and Gallager [8]. The generality of the definition of bottlenecks used in this paper suggests that our framework can be applied to a wide variety of rate allocation schemes—not only to max-min fairness [8], proportional fairness [9] and specific algorithms (e.g., BBR [2], Cubic [7], Reno [10], etc.), but to other classes of congestion control solutions that meet the conditions of Definition 2. We leave this promising direction for future work, and focus on the classic max-min setting considered in [3].

We complete the description of our network model by defining the concept of a link’s fair share:

Definition 3. Fair share of a link. Let \( N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) be a network. The fair share \( s_l \) of a link \( l \in \mathcal{L} \) is the rate of the flows that are bottlenecked at link \( l \).

As we will see throughout this work, the concept of link fair share is dual to the concept of flow rate, in that many of the mathematical properties that are applicable to the rate of a flow are also applicable to the fair share of a link.

2.2. The Gradient Graph

Our objective is to derive a mathematical framework capable not just of detecting but also of quantifying the influences that links and flows exert on each other. In [3], the authors introduced two bottleneck structures, the bottleneck precedence graph (BPG) and the gradient graph, and demonstrated that data networks qualitatively operate according to the BPG structure. The authors briefly described the concept of the gradient graph, but their work focused mostly on the mathematical properties of the bottleneck precedence 1. We use the notation \( \partial y/\partial x^- \) to denote the left derivative. This subtlety is necessary because a flow can have multiple bottleneck links. In this case, decreasing the capacity of only one bottleneck would affect the rate of the flow, while increasing its capacity would not; thus, the (two-sided) derivative would not exist.
graph. In our paper, we instead focus on a modified version of the gradient graph structure. Our work stems from the insight that, as we will show, this structure enables not just qualitative analysis, as in [3], but also quantitative analysis, providing a framework to better understand and optimize network performance.

We start with the definition of the gradient graph:

**Definition 4.** Gradient graph. Let \( \mathcal{N} = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) be a network. The gradient graph is a directed graph such that:

1) There exists a vertex for each bottleneck link and each flow in the network.
2) For every flow \( f \in \mathcal{F} \):
   a) If \( f \) is bottlenecked at link \( l \in \mathcal{L} \), then there exists a directed edge from \( l \) to \( f \);
   b) If \( f \) traverses link \( l \in \mathcal{L} \), then there exists a directed edge from \( f \) to \( l \);

For ease of exposition, in this paper we will use the terms gradient graph and bottleneck structure interchangeably. This definition is borrowed from [3], except for a subtle but relevant modification of 2b. (The rest of the theoretical developments presented in this work are new contributions.) Previously, edges were only included from flows to links that they traverse, but that do not bottleneck them. In this work, we also include edges from flows to their bottleneck links. We call these “backward edges”, and we introduce them because they are required by several of our theorems and algorithms.

The utility of our definition of gradient graph as a data structure for understanding network performance is captured in the following theorem:

**Theorem 1.** Propagation of network perturbations. Let \( x, y \in \mathcal{L} \cup \mathcal{F} \) be a pair of links or flows in the network. Then a perturbation in the capacity \( c_x \) (for \( x \in \mathcal{L} \)) or transmission rate \( r_x \) (for \( x \in \mathcal{F} \)) of \( x \) will affect the fair share \( s_y \) (for \( y \in \mathcal{L} \)) or transmission rate \( r_y \) (for \( y \in \mathcal{F} \)) of \( y \) if only if there exists a directed path from \( x \) to \( y \) in the gradient graph.

**Proof.** See Appendix [3]

Intuitively, the gradient graph of a network describes how perturbations in link capacities and flow transmission rates propagate through the network. Imagine that flow \( f \) is bottlenecked at link \( l \). From Definition [3], this necessarily implies that a perturbation in the capacity of link \( l \) will cause a change on the transmission rate of flow \( f \), \( \partial r_f / \partial c_l \neq 0 \). This is reflected in the gradient graph by the presence of a directed edge from a link \( l \) to a flow \( f \) (Condition 2a in Definition [4]). A change in the value of \( r_f \), in turn, affects all the other links traversed by flow \( f \). This is reflected by the directed edges from \( f \) to the links it traverses (Condition 2b).

In this section, we focus on the problem of quantifying the ripple effects created by perturbations in a network. Because networks are composed of links and flows, there are two kinds of perturbations: (1) those originating from changes to the capacity of a link and (2) those originating from changes to the rate of a flow. When such changes occur, the congestion control algorithm adjusts its allocation...
of bandwidth to the flows so as to maintain two objectives: (1) maximizing network utilization while (2) ensuring fairness among competing flows. The congestion control algorithm acts like a function mapping network conditions (including its topology, link capacities, and flow paths) to rate allocations. Large changes in any of these inputs can have complicated ripple effects on the flow rates, but for sufficiently small changes, the bandwidth allocation function is linear. This local linearity property naturally motivates the concept of link and flow gradients.

Definition 6. Link and flow gradients. Let \( N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) be a network. We define:

- The gradient of a link \( l \in \mathcal{L} \) with respect to another link \( l' \in \mathcal{L} \) as \( \nabla_{l'}(l) = \partial s_l / \partial c_{l'} \).
- The gradient of a flow \( f \in \mathcal{F} \) with respect to some link \( l' \in \mathcal{L} \) as \( \nabla_{l'}(f) = \partial r_f / \partial c_{l'} \).
- The gradient of a link \( l \in \mathcal{L} \) with respect to a flow \( f^* \in \mathcal{F} \) as \( \nabla_{l'}(f^*) = \partial s_l / \partial r_{f'} \).
- The gradient of a flow \( f \in \mathcal{F} \) with respect to another flow \( f^* \in \mathcal{F} \) as \( \nabla_{l'}(f) = \partial r_f / \partial r_{f'} \).

Intuitively, the gradient with respect to a link measures the impact that a small perturbation in its capacity has on another link or flow. In real networks, this corresponds to the scenario of physically upgrading a link or, in programmable networks (e.g., SDN), logically modifying the capacity of a virtual link. Thus, link gradients can generally be used to resolve network design and capacity planning problems. Similarly, the gradient with respect to a flow measures the impact that a perturbation in its rate has on a link or another flow. This scenario corresponds, for instance, to the case of traffic shaping a flow to alter its transmission rate or changing the route of a flow—which can be seen as dropping the rate of that flow down to zero and adding a new flow with a different path. Thus, flow gradients can generally be used to resolve traffic engineering problems. In Section 3 we will see applications in real networks that illustrate each of these scenarios.

We now present an algorithm called ForwardGrad() (Algorithm 2) for calculating link and flow gradients. The algorithm takes a set of links and flows, the gradient graph of the corresponding network, a link or flow \( x \) with respect to which to compute the gradients, and a direction \( \Delta x \) of the perturbation. It outputs the gradients of all links and flows in the network with respect to \( x \). ForwardGrad() takes inspiration from forward mode automatic differentiation (“Forward Prop”)[12], an algorithm that uses directed acyclic graphs to represent complicated mathematical functions as compositions of simpler functions, whose derivatives can be composed by repeatedly applying the chain rule. In the case of congestion control, we do not have a closed-form mathematical formula that relates network conditions (the inputs) to the flow rates and fair share values (the outputs), but we can use the gradient graph to break down and optimize this function.

The thrust of the algorithm is as follows. For all \( l \in \mathcal{L} \), let \( \Delta l \) be the change in the fair share rate of link \( l \). For all \( f \in \mathcal{F} \), let \( \Delta f \) be the change in the rate of flow \( f \). We call these variables the “drifts” caused by a perturbation. Before the perturbation, \( \Delta l = \Delta f = 0 \) for all links and flows. To begin the algorithm, we make an infinitesimally small perturbation in the independent variable (the one in the “denominator” of the derivative) that can be positive or negative. If the independent variable \( x \) is a flow \( f \), we set \( \Delta f = \delta \) (line 2). If it is a link \( l \), and \( S_l \) is the set of direct successors of node \( l \) in the gradient graph, we set \( \Delta l = \delta / |S_l| \) (line 3). This is done since, by definition of the gradient graph, \( |S_l| \) is the number of flows bottlenecked at \( l \) and the change in \( l \)’s capacity will be distributed evenly among these flows. To determine how this perturbation propagates to the rest of the network, we follow all directed paths from that vertex and update the drifts according to the following two invariants:

- Invariant 1: Flow Equation. A flow’s drift \( \Delta f \) equals the minimum drift of its bottleneck links. That is, \( \Delta f = \min_{l \in P_f} \Delta l \), where \( P_f \) is the set of links visited directly before flow vertex \( f \) on a path from the starting vertex \( x \) (the predecessors in the graph).
- Invariant 2: Link Equation. A link’s drift \( \Delta l \) is the negative of the flow drifts entering its vertex, divided by the number of flow drifts leaving it. That is, \( \Delta l = - \sum_{f \in P_l} \Delta f / |S_l| \), where \( P_l \) is the set of flow vertices visited directly before link vertex \( l \) and \( S_l \) is the set of flow vertices visited directly after link vertex \( l \) on a path from the starting vertex \( x \).

Finally, the derivative of a given variable with respect to the independent variable that we perturbed can be calculated by dividing its drift by \( \delta \). In particular, assume the capacity of link \( l \) is the independent variable that we perturbed and let the rate of flow \( f \) be the dependent variable in which
we want to measure the effect of this perturbation. Then, \( \partial r_f / \partial c_l = \Delta_f / \delta \).

Since the flow and link equations lie at the heart of the algorithm, we provide some further explanation. Invariant 1 ensures that the capacity limits are respected and the network’s resources are not wasted. Each flow must use exactly the amount of bandwidth allocated by its bottleneck link, so if the bottleneck’s fair share changes, the flow’s rate must change too. It also ensures fairness, since each flow bottlenecked at a certain link will experience the same drift. Invariant 2 ensures that capacity is neither created nor destroyed through the process of propagating a perturbation, except at the link whose capacity was initially perturbed. If a link’s predecessors are using less bandwidth than before, then the savings must be redistributed evenly among the other flows that traverse the link.

Fig. [1](a) and (b) show graphical representations of the link and flow equations. Fig. [1](c) and (d) present two simple examples of gradient graphs that we use to illustrate how to compute link and flow gradients. Note that throughout the paper, we use white vertices to denote bottleneck links and gray vertices to denote flows. We also omit backward edges for visual simplicity. Fig. [1](c) presents the case of computing the link gradient \( \nabla f_1 (f_2) \). A perturbation is applied to link \( l_1 \) that decreases its capacity \( c_{l_1} \) by an infinitesimally small amount \( \delta \). Since only one flow is bottlenecked at \( l_1 \), we have \( \Delta_{l_1} = -\delta \). This perturbation propagates to flow \( f_1 \) according to the flow equation: \( \Delta f_{l_1} = \min \{ \Delta l_1, \Delta_{l_2} \} \). The perturbation is propagated down to link \( l_2 \) according to the link equation: \( \Delta l_2 = -\Delta f_{l_1} / 2 = \delta / 2 \). Finally, applying the flow equation for \( f_2 \), we obtain the flow drift \( \Delta f_2 = \delta / 2 \). Thus, the gradient of flow \( f_2 \) with respect to link \( l_1 \) is \( \nabla f_1 (f_2) = \Delta f_2 / \delta = 1 / 2 \).

To make this process into a precise algorithm, we still must specify the order in which to process the vertices of the graph. At each step, the vertex process must be a neighbor of one of the vertices we have already visited. Even though backward edges create loops in the gradient graph, we never visit a vertex twice. If multiple vertices meet these criteria, we pick the one with the minimal rate or fair share value. If there are multiple vertices with the minimal rate or fair share value, we pick the one that would receive the minimum drift if it were processed next (see line 15 where keys in the heap are ordered pairs of rate/fair share and drift). This reflects the order in which the bottleneck structures are constructed in Algorithm 1 which itself reflects the order in which the rates and fair shares converge in congestion controlled networks [3]. That is, we first visit the vertex that would receive the smallest rate or fair share if the perturbation were applied and bandwidth were reallocated from scratch. This completes the description of the Forward-Grad() algorithm.

**Algorithm 2** ForwardGrad(\( L, F, G, x \in L \cup F, \Delta x \in \{ \pm 1 \} \))

\[
\begin{align*}
1: & \quad \Delta c_l = 0 \quad \forall l \in L \\
2: & \quad \Delta r_f = 0 \quad \forall f \in F \quad \# \text{Drift of flow } f \\
3: & \quad \Delta s_l = 0 \quad \forall l \in L \quad \# \text{Drift of link } l \\
4: & \quad \text{if } x \in F \text{ then} \\
5: & \quad \Delta u_x = \Delta x \\
6: & \quad \text{MinHeapAdd(key = (} u_x, \Delta u_x \text{), value = } x) \\
7: & \quad \text{else if } x \in L \text{ then} \\
8: & \quad \Delta c_x = \Delta x \\
9: & \quad \Delta s_{r_x} = \Delta c_x \cdot \text{successors}(x, G) \\
10: & \quad \text{MinHeapAdd(key = (} s_x, \Delta s_x \text{), value = } x) \\
11: & \quad \text{end if} \\
12: & \quad V = \emptyset \quad \# \text{the set of previously visited nodes} \\
13: & \quad \text{repeat} \\
14: & \quad \text{repeat} \\
15: & \quad y = \text{MinHeapPop}(); \quad \# \text{Get the next unvisited node} \\
16: & \quad \text{until } y \notin V \\
17: & \quad V = V \cup \{y\} \\
18: & \quad \text{if } (y \in L \text{ and } \Delta s_y = 0) \text{ or } (y \in F \text{ and } \Delta u_y = 0) \text{ then} \\
19: & \quad \text{Continue} \\
20: & \quad \text{end if} \\
21: & \quad \text{for } y' \in \text{successors}(y, G) \setminus V \text{ do} \\
22: & \quad \text{if } y' \in F \text{ then} \\
23: & \quad \Delta r_{y'} = \min \{ r_{y'} \} \quad \# \text{Flow equation invariant} \\
24: & \quad \text{MinHeapAdd(key = (} r_{y'}, \Delta r_{y'} \text{), value = } y') \\
25: & \quad \text{else if } y' \in L \text{ then} \\
26: & \quad \Delta s_{y'} = \Delta c_{y'} - \Delta u_y \\
27: & \quad \Delta s_{y'} = \Delta s_{y'} \mid \text{successors}(y', G) \setminus V \mid \# \text{Link equation invariant} \\
28: & \quad \text{MinHeapAdd(key = (} s_{y'}, \Delta s_{y'} \text{), value = } y') \\
29: & \quad \text{end if} \\
30: & \quad \text{end for} \\
31: & \quad \text{until MinHeapEmpty}() \\
32: & \quad \text{return } (\Delta s_l \forall l \in L, \Delta r_f \forall f \in F) \\
\end{align*}
\]

The next two theorems show that Algorithm 2 is both correct and efficient.

**Theorem 2. Correctness of ForwardGrad.** Let \( N = \langle L, F, G, \{ c_l, \forall l \in L \} \rangle \) be a network and let \( G \) be the corresponding gradient graph. Let \( x \in L \cup F \). After running Algorithm 2, \( \Delta s_l = \nabla_s (l) \) for all \( l \in L \), and \( \Delta r_f = \nabla_r (f) \) for all \( f \in F \).

**Proof.** See Appendix [C] □

**Theorem 3. Time complexity of ForwardGrad.** Let \( x \in L \cup F \). Then Algorithm 2 finds the gradients of all links and flows in the network with respect to \( x \) in time \( O(|R(x)| \cdot \log |R(x)|) \).

**Proof.** See Appendix [D] □

To conclude and complement this section, we state an upper bound on the value of the gradients:

**Property 1. Gradient bound.** Let \( N = \langle L, F, \{ c_l, \forall l \in L \} \rangle \) be a network and let \( G \) be its gradient graph. Let \( \delta \) be an
infinitesimally small perturbation performed on a flow or link \( x \in L \cup F \), producing a drift \( \Delta y \), for all \( y \in L \cup F \). Then, \( |\nabla_x(y)| = |\Delta y|/\delta \leq d(D(G))/4 \), where \( D(X) \) is the diameter of a graph \( X \) and \( d \) is the maximum indegree and outdegree of any vertex in the graph.

**Proof.** See Appendix [F]. \( \square \)

3. Applications to Data Networks and Experimental Results

Because bottleneck structures are a fundamental property intrinsic to any congestion-controlled data network, its applications span a variety of networking problems. In this section, our goal is to present examples and experiments illustrating how QTBS can be used to resolve some of these problems. We will see that in each of them, the framework is able to provide new insights into one or more operational aspects of a network. The examples presented in this section are not exhaustive, but only illustrative. To help organize the applications, we divide them in two main classes: traffic engineering and capacity planning. For each of these classes, we provide specific examples of problems that relate to applications commonly found in modern production networks.

To experimentally demonstrate that data networks behave qualitatively and quantitatively according to QTBS, we use *Mininet-G2* [13], a network emulation framework developed by our team that consists of a set of software modules and extensions to Mininet [14]. Leveraging software define networking (SDN), *Mininet-G2* enables the creation and analysis of arbitrary network architectures using real production TCP/IP code, including production-grade implementations of congestion control algorithms such as BBR, Cubic or Reno. (See also Appendix [G] for more information.) We are open sourcing *Mininet-G2* and all the experiments presented in this paper, hoping this will also enable the research community to verify our findings and further experiment with the theory of bottleneck structures.

All the experimental results presented in this section are based on Google’s BBR congestion control algorithm [2]. Results for similar experiments using Cubic [7] can be found in Appendix [G]. For each experiment, we used Jain’s fairness index [15] as an estimator to measure how closely the predictions of the theory of bottleneck structure model match the experimental results. For all BBR experiments presented in the next sections, this index was above 0.99 accuracy on a scale from 0 to 1 (See Appendix [G], reflecting the strength of QTBS in modeling network behavior.

### 3.1. Traffic Engineering: Computation of the Highest-Throughput Route

In traditional IP networks, the problems of flow routing and congestion control are separately resolved by following a two-step process: first, a routing protocol (e.g., BGP [16], OSPF, etc.) is used to determine the path between any two nodes in a network; then, flows are routed according to such paths and their transmission rates are regulated using a congestion control algorithm (e.g., BBR [2]). This layered and disjoint approach is known to be scalable but suboptimal because the routing algorithm identifies paths without taking into account the flow transmission rates assigned by the congestion control algorithm [4], [17], [18], [19].

In this section, we use QTBS to resolve the following joint routing and congestion control problem in a scalable manner:

**Definition 7.** Flow-rate maximal routing. Let \( N = (L, F, \{c_l, \forall l \in L\}) \) be a network and suppose that a new flow \( f \) arrives. We will say that a routing algorithm is flow-rate maximal if it routes flow \( f \) through a path that maximizes its transmission rate \( r_f \).

In traditional IP routing, all packets transmitted from a source to a destination node follow the same lowest-cost route [16]. This rigidity leads to the well-known fish problem [8], whereby certain paths in a network become congested while other paths are underutilized. A flow-rate maximal algorithm, instead, is able to bypass points of congestion by assigning new flows to the highest-throughput path available given the current usage of the network.

One might mistakenly think that the least congested path can be identified by looking for links with small fair shares (Definition [F]). However, the placement of a new flow onto a given path will itself alter the state of the network, changing those fair shares and potentially rendering the chosen path sub-optimal. In this section, we show that QTBS can be used to identify the maximal-rate path for a flow while taking into account the perturbations created by the placement of the flow itself, thus solving the flow-rate maximal routing problem.

MaxRatePath() (Algorithm 3) is an algorithm that uses QTBS to compute flow-rate maximal paths. It takes the following inputs: a network \( N = (L, F, \{c_l, \forall l \in L\}) \), the set of routers \( U \), and the source and the destination routers of the flow we intend to route, \( u_s \) and \( u_d \). By convention, a link \( l \in L \) is identified with the tuple \( l = (u_x, u_y) \), where \( u_x, u_y \in U \) are the two routers connected by link \( l \). The algorithm returns the new flow \( f \), expressed as the set of links it traverses, guaranteeing they form a path from \( u_s \) to \( u_d \) that yields the maximal rate \( r_f \) for \( f \).

As the pseudocode shows, MaxRatePath() is based on Dijkstra’s shortest path algorithm, with routers as vertices and links as edges in the network topology graph. The difference resides in the way the “distance” to a neighboring router \( u' \) is calculated (lines 12-14). In MaxRatePath(), this value represents not the number of hops on the shortest path from \( u_s \) to \( u' \), but the inverse of the largest possible rate that a flow would experience if it were added on some path from \( u_s \) to \( u' \). That is, the distance to \( u' \) is the smallest possible time needed to send 1 bit of information from \( u_s \) to \( u' \). Unlike in the standard Dijkstra’s algorithm, this value cannot be computed by adding an edge length to \( d_{eu} \) the distance to a neighbor of \( u' \). Instead, we create a new flow \( f \) by extending the optimal path from \( u_s \) to \( u \). So at each iteration of the algorithm, \( f \) takes the path \( u_s \to \cdots \to u \to u' \)
We will assume there are two flows (one for each direction) described in \[20\], the SDN-WAN network that connects a path for a given flow, consider the network shown in Fig. 4. To illustrate how we can use QTBS and the MaxRatePath algorithm to compute the highest-throughput path of a flow using the bottleneck structure. See Appendix G for a procedure to efficiently compute the maximal-rate path of a flow.

Lemma 2. Correctness of the MaxRatePath algorithm. Let \(N = \langle L, F, \{e_l, v_l \in L\} \rangle \) be a network and \(U \) the set of its routers. Suppose that \(f \) and \( f' \) are two flows not in \( F \) that originate at router \( u_s \) and end at router \( u_d \). Then \( f = \text{MaxRatePath}(N, U, u_s, u_d) \) implies \( r_f \geq r_{f'} \).

Proof. The proof of this lemma is constructive and describes a procedure to efficiently compute the maximal-rate path of a flow using the bottleneck structure. See Appendix G for details.

To illustrate how we can use QTBS and the MaxRatePath algorithm to compute the highest-throughput path for a given flow, consider the network shown in Fig. 2. This topology corresponds to Google’s B4 network as described in [20], the SDN-WAN network that connects Google’s data centers globally. For the sake of illustration, we will assume there are two flows (one for each direction) connecting every data center in the US with every data center in Europe, with all flows routed along a shortest path from source to destination. Since there are six data centers in the US and four in Europe, this configuration has a total of 48 flows (\(|F| = 6 \times 4 = 24\)). (See Table 7 in Appendix G for a description of the exact path followed by each flow.) All links are assumed to have a capacity of 10 Gbps except for the transatlantic links, which are configured at 25 Gbps (i.e., \(c_l = 10\), for all \(l \notin \{l_8, l_{10}\}\), \(c_{l_8} = c_{l_{10}} = 25\)). While obviously production networks operate with a much higher number of flows, in our example we use a reduced number to simplify the descriptions of the bottleneck structures and the steps followed to resolve the given problem. This simplification is without loss of generality, and the same approach is applicable to large scale operational networks. (See Appendix G for notes on integration with production networks.)

Fig. 2 shows the corresponding bottleneck structure obtained from running Algorithm 3 on the proposed network configuration. This structure shows that flows are organized in two levels: the top-level includes flows \(\{f_1, f_2, f_3, f_4, f_5, f_7, f_8, f_{10}, f_{13}, f_{14}, f_{15}, f_{16}\}\) and the low-level includes flows \(\{f_6, f_9, f_{11}, f_{12}, f_{17}, f_{18}, f_{19}, f_{20}, f_{21}, f_{22}, f_{23}, f_{24}\}\). Note that because each pair of data centers is connected via two flows (one for each direction), without loss of generality, in Fig. 2 we only include the first 24 flows (flows transferring data from US to Europe), since the results are symmetric for rest of the flows—i.e., flow \(f_i\) has the same theoretical transmission rate and is positioned at the same level in the bottleneck structure as flow \(f_{i+24}\) for all \(1 \leq i \leq 24\). Note also that all the top-level flows operate at a lower transmission rate (with all rates at 1.667) than the bottom-level flows (with rates between 2.143 and 3). As was proven in [3], this is in fact a property of all bottleneck structures: flows operating at lower levels of the bottleneck structure have greater transmission rates than those operating at higher levels.

![Diagram](image_url)
In Fig. 2c we show the bottleneck structure obtained for the case that f_{25} uses the shortest path l_{15} → l_{10}. For instance, this corresponds to the solution obtained from running BGP [16] with a link cost metric equal to 1. Using this path, the new flow would be placed at the upper bottleneck level—i.e., the lower-throughput level—in the bottleneck structure, receiving a theoretical rate of r_{25} = 1.429. Note that the presence of this new flow slightly modifies the performance of some of the flows on the first level (flows \{f_1, f_3, f_4, f_5, f_7, f_8\} experience a rate reduction from 1.667 to 1.429), but it does not modify the performance of the flows operating at the bottom level. This is because, for the given configuration, the new flow only creates a shift in the distribution of bandwidth on the top level, but the total amount of bandwidth used in this level stays constant. (In Fig. 2b, the sum of all the flow rates on the top bottleneck level is 1.667 × 12 = 20, and in Fig. 2c this value is the same: 1.429 × 7 + 1.667 × 6 = 20.) As a result, the ripple effects produced from adding flow f_{25} into the network cancel each other out without propagating to the bottom level.

While l_{15} → l_{10} is the shortest path, it is not the path with the highest throughput. To find such a path, we run the MaxRatePath procedure (Algorithm 3) and obtain the solution l_{16} → l_8 → l_{19}. The resulting bottleneck structure is shown in Fig. 2d. Using this path, flow f_{25} would now be placed at the bottom level—the higher-throughput level—in the bottleneck structure, thus resulting in a rate value r_{25} = 2.5, an increase of 74.95% with respect to the shortest path solution. Another positive outcome of this solution is that none of the flows operating at the upper level (the flows that receive less bandwidth) see their rate reduced. This is a direct consequence of Theorem 1, since a perturbation on lower levels can have no ripple effects on upper levels. It constitutes also a natural fairness property of the MaxRatePath algorithm: as the procedure assigns maximal-throughput paths to new incoming flows, such flows tend to be placed at the bottom of the bottleneck structure (where the high-throughput links are located), thus tending to create no negative impact on the lower-throughput flows located at the top of the structure.

In the remainder of this section, we set out to empirically confirm these results. We start by creating the B4 network configuration shown in Fig. 2a using Mininet-G2. Following our example, we deploy a total of 48 shortest-path flows connecting every pair of nodes (in both directions) between the US and Europe. (Table 7 in Appendix G presents the exact path followed by each flow.) We then add two extra flows labeled f_{25} and f_{30} (one for each direction) to connect data centers 4 and 11 and perform two separate experiments: one placing the flows on the shortest path l_{15} ↔ l_{10} and another one placing them on the longer path l_{16} ↔ l_8 ↔ l_{19}.

Fig. 3a shows the rate of flow f_{25} for the two experiments (very similar results are obtained for the reverse-path flow f_{30}, see Appendix G). In the legend of this plot, experiment 1 and 2 correspond to the shortest and the (longer) maximal-throughput path configurations, respectively. As predicted by the bottleneck structure, the longer path achieves a higher throughput and, thus, a lower flow completion time. Fig. 3b presents the average throughput obtained for all twenty-five flows from the US to Europe and for each of the two experiments, alongside the theoretical values according to the bottleneck structure. (The results obtained from the other twenty-five flows on the reverse path are similar and can be found in Appendix C.) As shown, flow f_{25} achieves a performance of 1.226 and 2.386 Mbps for the shortest and longer paths, respectively— with the theoretical rates being 1.428 and 2.5 Mbps, respectively. Thus, the longer path yields a 94% improvement on flow throughput compared to the shortest path. For all the experiments run in this section, Jain’s fairness index was above 0.99 (see Appendix C), indicating the accuracy of QTBS in predicting flow performance.

This experiment illustrates that using QTBS, it is possible to identify routes that are highly efficient from a congestion control standpoint. Note that this contrasts with traditional approaches that perform traffic engineering by separating the routing and congestion control problems, so that the routing algorithm is unaware of the choices made by the congestion control algorithm and vice versa. See for instance Section 5.3.1 in [4], which discusses the potential advantages of performing joint routing and congestion control in Google’s WAN, but leaves this direction as future work. We reason that QTBS provides a mathematical framework to connect both problems, identifying routes that are globally efficient from both a topological and a congestion control standpoint.
to demonstrate that, due to the effects of the congestion control algorithm, there exists an optimal trade-off in the allocation of capacity at the top levels of the fat-tree. Further, we show that the optimal bandwidth allocation on the top level deviates from commonly accepted best practices in the design of full fat-tree networks that tend to equate the amount of bandwidth going up and down the tree at each switch [22].

Consider the network topology in Fig. 4a, which corresponds to a binary fat-tree with three levels and six links (\( L = \{l_1, l_2, \ldots, l_6\} \)). Assume also that there are two flows (one for each direction) connecting every pair of leaves in the fat-tree network, providing bidirectional full-mesh connectivity among the leaves. Since there are four leaves, that’s a total of \( 4 \times 3 = 24 \) flows. All of the flows are routed following the shortest path. (See Table 8 in Appendix [23] for a description of the exact path followed by each flow.) For the sake of convention, we will adopt the terminology from data center architectures and use the names spine and leaf links to refer to the upper and lower links of the fat-tree network, respectively [5].

We fix the capacity of the leaf links to a value \( \lambda \) (i.e., \( c_{l_1} = c_{l_2} = c_{l_3} = c_{l_4} = \lambda \)) and the capacity of the spine links to \( \lambda \times \tau \) (i.e., \( c_{l_5} = c_{l_6} = \lambda \times \tau \)), where \( \tau \) is used as a design parameter enabling a variety of network configurations. For instance, in our binary fat-tree example, the case \( \tau = 2 \) corresponds to a full fat-tree network [22], because the total aggregate bandwidth at each level of the tree is constant, \( c_{l_1} + c_{l_2} + c_{l_3} + c_{l_4} = c_{l_5} + c_{l_6} = 4\lambda \). Similarly, the case \( \tau = 1 \) corresponds to a thin-tree network, since it results with all the links having the same capacity, \( c_{l_i} = \lambda \), for all \( 1 \leq i \leq 6 \). The technique of optimizing the performance-cost trade-off of a fat-tree network by adjusting the capacity of the spine links is sometimes known as bandwidth tapering [21]. [23]. The focus of our experiment is to use the bottleneck structure analysis to identify optimized choices for the tapering parameter \( \tau \).

In Fig. 4 we present a sequence of bottleneck structures (obtained from running Algorithm 1) corresponding to our fat-tree network with three different values of the tapering parameter \( \tau \) and fixing \( \lambda = 20 \). (Note that the fixing of \( \lambda \) to this value is without loss of generality, as the following analysis applies to any arbitrary value \( \lambda > 0 \).) The first bottleneck structure (Fig. 4b) corresponds to the case \( \tau = 1 \) (i.e., all links have the same capacity, \( c_{l_i} = 20 \), for all \( 1 \leq i \leq 6 \)), which has all flows confined in one of two possible levels: a top level, where flows perform at a lower rate, \( r_{f_2} = r_{f_3} = r_{f_5} = r_{f_6} = r_{f_7} = r_{f_8} = r_{f_{10}} = 2.5 \); and a bottom level, where flows perform at twice the rate of the top-level flows, \( r_{f_1} = r_{f_4} = r_{f_9} = r_{f_{11}} = 5 \). This configuration is thus unfair to those flows operating at the top bottleneck, which receive half the bandwidth of the flows at the bottom level. Furthermore, this configuration is also inefficient at supporting applications with symmetric workload patterns—where all nodes approximately send the same amount of bytes to each other—because the completion time of the slowest flows is significantly higher (twice as high since they get half the rate) than the faster flows. Let us next consider how we can use QTBS to identify a value of \( \tau \) that minimizes the maximum completion time of any of the flows under the assumption of symmetric workloads.

By looking at the bottleneck structure in Fig. 4b, we see that the slowest flows are confined in the top bottleneck level. In order to increase the rates of these flows, we need to increase the tapering parameter \( \tau \) that controls the capacity of the spine links \( l_5 \) and \( l_6 \). This change transforms the bottleneck structure by bringing the two levels closer together, until, for a large enough value of \( \tau \), they fold. We can find this collision point using the link gradients as follows. Using ForwardGrad() (Algorithm 2), we obtain a link gradient value of \( \nabla_i(f) = 0.125 \) for all spine links \( l \in \{l_5, l_6\} \) and top-level flows \( f \in \{f_2, f_3, f_5, f_6, f_7, f_8, f_{10}, f_{11}\} \). On the other hand, the link gradient of any of the low-level flows with respect to any of the spine links is \( \nabla_i(f) = -0.25 \), for all \( l \in \{l_5, l_6\} \) and \( f \in \{f_1, f_4, f_9, f_{12}\} \). That is, an increase by one unit on the capacity of the spine links increases the rate of the top-level flows by 0.125 and decreases the rate of the low-level flows by 0.25. Since the rates of the top and low-level flows are 2.5 and 5, respectively, this means that the two levels will fold at a point where the tapering parameter satisfies the equation \( 2.5 + 0.125 \cdot \tau \cdot \lambda = 0.25 \cdot \tau \cdot \lambda \), yielding \( \tau = 4/3 \) and \( c_{l_5} = c_{l_6} = 26.667 \). The resulting bottleneck structure for this configuration is shown in Fig. 4c confirming the folding of the two levels. This fat-tree configuration is optimal in that the flow completion time of the slowest flow is minimal. Because the bottleneck structure is folded into a single level, this configuration also ensures that all flows perform at the same rate, \( r_{f_i} = 3.333 \), for all \( 1 \leq i \leq 6 \).

![Fat-tree network.](image)

![Tapering parameter \( \tau = 1 \).](image)

![Tapering parameter \( \tau = 4/3 \).](image)

![Tapering parameter \( \tau = 2 \).](image)

Figure 4: Design of optimal 3-level binary fat-trees.

What is the effect of increasing the tapering parameter above 4/3? This result is shown in Fig. 4d for the case of a full fat-tree \( \tau = 2 \), i.e., \( c_{l_5} = c_{l_6} = 40 \). In this case, the two spine links are no longer bottlenecks to any of the flows (since these links are leaves in the bottleneck structure), but all flows continue to perform at the same rate, \( r_{f_i} = 3.333 \), for all \( 1 \leq i \leq 6 \). Thus, increasing the capacity of the upper-level links above 26.667 does not yield any benefit, but increases the cost of the network. This result indicates that the fat-tree network shown in Fig. 4a should not be designed with an allocation of capacity on the spine links.
higher than \( \tau = 4/3 \) times the capacity of the leaf links. In summary, we have that:

- A tapering parameter \( \tau \geq 4/3 \) should not be used, since the resulting network is just as efficient as a design with \( \tau = 4/3 \), but more costly.
- A tapering parameter \( \tau = 4/3 \) is optimal in that it minimizes the flow completion time of the slowest flow. This should be the preferred design in symmetric workloads that transfer about the same amount of data between all pairs of nodes.
- A tapering parameter \( \tau < 4/3 \) can be used if workloads are asymmetric, identifying the right value of \( \tau \) that produces the right amount of bandwidth at each level of the bottleneck structure according to the workload.

In the rest of this section, we empirically demonstrate the existence of an optimal fat-tree design at \( \tau = 4/3 \) using Mininet-G2 [13] configured with the congestion control algorithm BBR. Fig. 5 presents the results of the experiments for the three values of the tapering parameter, \( \tau \in \{1, 4/3, 2\} \). Each plot shows the transmission rate of all twelve flows as part of the network configuration, with each flow transmitting a total of 64 MB of data. Following the example in Section 3.2, the link capacities are set as follows: \( c_{t_1} = c_{t_2} = c_{t_3} = c_{t_4} = \lambda = 20 \text{ Mbps} \) and \( c_{t_5} = c_{t_6} = \lambda \times \tau = 20 \times \tau \text{ Mbps} \).

TABLE 1: Flow completion times (seconds) of the fat-tree experiments.

| Flow | \( \tau = 1 \) | \( \tau = 4/3 \) | \( \tau = 2 \) | Flow | \( \tau = 1 \) | \( \tau = 4/3 \) | \( \tau = 2 \) |
|------|----------------|----------------|----------------|------|----------------|----------------|----------------|
| \( f_1 \) | 175 | 172 | 175 | \( f_2 \) | 123 | 152 | 144 |
| \( f_2 \) | 237 | 171 | 164 | \( f_3 \) | 212 | 170 | 143 |
| \( f_3 \) | 239 | 177 | 156 | \( f_4 \) | 112 | 171 | 178 |
| \( f_4 \) | 111 | 172 | 173 | \( f_5 \) | 201 | 173 | 153 |
| \( f_5 \) | 236 | 167 | 158 | \( f_6 \) | 226 | 174 | 154 |
| \( f_6 \) | 233 | 172 | 147 | \( f_7 \) | 113 | 175 | 123 |

As predicated by QTBS, the case \( \tau = 1 \) has flows operating at one of two bottleneck levels, close to the rates predicted by the bottleneck structure (2.5 Mbps for the upper-level flows and 5 Mbps for the lower-level flows, see Fig. 2b). This fat-tree design is inefficient for symmetric workloads since the flow completion time of the slowest flow is not minimal. Under this configuration, flow \( f_2 \) is the slowest flow and its completion time is 239 seconds. (See Table 1 for all flow completion time values.) If we want to maximize the rate of the slowest flow, QTBS tells us that the right tapering parameter value is 4/3. This case is presented in Fig. 5b, which indeed shows how all flows perform at a very similar rate close to the theoretical value of 3.333 Mbps (see Fig. 4c). This configuration is optimal in that it minimizes the maximum completion time of any of the flows. In this experiment, the completion time of the slowest flow is 177 seconds, an improvement of 25.9% with respect to the case of \( \tau = 1 \). Fig. 5c shows the results for the case of a full fat-tree network, \( \tau = 2 \). Once again, as predicted by QTBS, this solution achieves about the same completion time as the case \( \tau = 4/3 \) (the slowest flow completes in 178 seconds), since in this configuration the leaf links become the bottlenecks and the extra bandwidth added in the spine links does not produce any net benefit, as shown by the bottleneck structure in Fig. 4a. In summary, as predicted by QTBS, the case \( \tau = 4/3 \) constitutes an optimal design in that it is the least costly network that minimizes the maximum completion time of any of the flows.

Note that the existence of an optimal design with a tapering parameter \( \tau = 4/3 \) argues against some of the established conventional best practices in fat-tree networks. For instance, while a full fat-tree (\( \tau = 2 \)) is considered to be universally efficient [6], the analysis of its bottleneck structure demonstrates that such design is in general inefficient when flows are regulated by a congestion-control protocol. This is because the fairness and throughput maximization objectives targeted by the congestion control algorithm effectively bends the solution space and, as a result, the optimal fat-tree design deviates from the general full fat-tree configuration. This result has implications in the design of data centers that use fat-tree topologies (also known as folded-Clos [5]). While in this section we have illustrated how QTBS can be used to optimize a single fat-tree topology for the case of a symmetric workload pattern, the authors are currently working on deriving the general equations for the optimal design of fat-trees with arbitrary number of spine and leaf links and for generalized (non-symmetric) workload patterns. We will be presenting these results in a forthcoming paper.

3.3. Traffic Engineering: Accelerating Time-Bound Constrained Flows

Suppose now that our goal is to accelerate a certain flow \( f_t \in \mathcal{F} \) in a network \( \mathcal{N} \) so it completes before a target time. A common application for the optimization of time-bound flows can be found in research and education networks, where users need to share data obtained from their experiments, often involving terabytes (or even petabytes) of information, with collaborators around the globe—e.g., when scientists at the European Organization for Nuclear Research (CERN) need to share data with other researchers using the LHCONE network [24] across the globe. Another common use case can be found in large scale data centers, where massive data backups need to be transferred between sites to ensure redundancy [20]. In this context, assume operators are only allowed to sacrifice the performance of a subset of flows \( \mathcal{F}' \subset \mathcal{F} \setminus \{f_t\} \), considered to be of lower priority than \( f_t \). What flows in \( \mathcal{F}' \) constitute an optimal choice to traffic-shape so as to accelerate \( f_t \)? By what amount should the rate of such flows be reduced? And by what amount will flow \( f_t \) be accelerated?

To illustrate that we can use QTBS to resolve the above problem, we will use the topology of Google’s B4 network (Fig. 2a) introduced in Section 3.1. Assume the network is transporting eight flows, \( \mathcal{F} = \{f_1, f_2, \ldots, f_8\} \), routed as shown in the Fig. 6. This is without loss of generality as we can apply the same procedure to optimize networks with arbitrary number of flows and topology. We will use the network’s bottleneck structure to identify an optimal strategy for accelerating an arbitrary flow in a network. Assume that
our objective is to accelerate flow $f_7$ (i.e., $f_3 = f_7$) in Fig. 6—i.e., the transatlantic flow that connects data centers 8 and 12—to meet a certain flow completion time constraint. Assume also that in order to maximize the performance of $f_7$ we are allowed to traffic shape any of the flows in the set $F' = \{f_1, f_3, f_4, f_6\}$. In other words, the set of flows in $F'$ are considered by the network operator to be of lower priority.

Fig. 7 displays the sequence of gradient graphs that lead to the acceleration of flow $f_7$ to meet its time constraint. The graphs include the values of the capacity $c_l$ and fair share $s_l$ next to each link vertex $l$ and the rate $r_l$ next to each flow vertex $f$. Fig. 7a corresponds to the gradient graph of the initial network configuration shown in Fig. 6 as computed by Algorithm 1. From Theorem 1 we know that only the flows that are ancestors to $f_7$ can have an effect on its performance. That means we can discard traffic shaping flow $f_3$ as that will have no impact.

We can use the ForwardGrad() algorithm (Algorithm 2) to obtain the gradients of flow $f_7$ with respect to the flows in the low priority set $F'$: $\nabla f_7(f_2) = 2$, $\nabla f_7(f_3) = -1$, $\nabla f_7(f_4) = -2$, and $\nabla f_7(f_6) = 0$. We are interested in finding the gradient of a flow in $F'$ that has the highest negative value, so that the traffic shaping of such a flow (i.e., the reduction of its rate) creates a maximal positive increase in the rate of $f_7$. We have that flow $f_4$ has the highest negative gradient with a value of $-2$, yielding an optimal traffic shaping decision. From Fig. 7a it can be observed that the reduction of flow $f_4$’s rate creates a perturbation that propagates through the bottleneck structure via two different paths: $f_4 \rightarrow l_2 \rightarrow f_2 \rightarrow l_3 \rightarrow f_3 \rightarrow l_4 \rightarrow f_7$ and $f_4 \rightarrow l_9 \rightarrow f_7$. Each of these paths has an equal contribution to the gradient of value $-1$, resulting in $\nabla f_4(f_7) = -2$.

We can use the bottleneck structure again to calculate the exact value of the traffic shaper—i.e., the rate reduction applied to flow $f_4$. The core idea is that traffic shaping flow $f_4$ constitutes an optimal decision as long as the bottleneck structure does not change, since a change in the structure would also imply a change in the gradients. As the rate of flow $f_4$ is reduced, some levels in the bottleneck structure will become further away from each other, while the others will become closer to each other. Thus, the latter set will fold if the rate reduction imposed by the traffic shaper is large enough. The speed at which two links in the bottleneck structure get closer to (or further away from) each other is given by their gradients. In particular, if the traffic shaper reduces the rate of flow $f_4$ by an amount of $\rho$ bps, then two links $l$ and $l'$ in the bottleneck structure will collide at a value of $\rho$ that satisfies the equation $s_l - \rho \cdot \nabla f_4(l) = s_{l'} - \rho \cdot \nabla f_4(l')$. From the bottleneck structure (Fig. 7a) we can obtain the fair share values $s_l$ and using the ForwardGrad() algorithm we can compute the link gradients $\nabla f_4(l)$: $s_{l_2} = 5.125; s_{l_3} = 7.375; s_{l_4} = 10.25; s_{l_9} = 12.25; \nabla f_4(l_2) = -1; \nabla f_4(l_3) = 1; \nabla f_4(l_4) = -2; \nabla f_4(l_9) = 2$. Using these values, we have that the smallest value of $\rho$ that satisfies the collision equation corresponds to the case $l = l_4$ and $l' = l_9$, yielding a $\rho$ value of 0.5 (since $10.25 - 0.5 \cdot 2 = 12.25 - \rho \cdot 2 \Rightarrow \rho = 0.5$). Thus, we conclude that to maximally increase the rate of flow $f_7$, an optimal strategy is to decrease the rate of flow $f_4$ by an amount of 0.5 units of bandwidth. The resulting bottleneck structure is presented in Fig. 7b where a new link $l_7$ has been added that corresponds to the new traffic shaper set to reduce the rate of flow $f_4$ by an amount of 0.5 (from 2.375 down to 1.875). Note that as expected, in this new bottleneck structure links $l_4$ and $l_9$ are folded into the same level and have the same fair share: $s_4 = s_9 = 11.25$.

Since $f_7$ has now two bottleneck links ($l_4$ and $l_9$), we cannot accelerate it further unless we increase the fair-shares of both. Using the new bottleneck structure (Fig. 7b), it is easy to see that this can be achieved by decreasing the rate of flows $f_3$ and $f_8$, since the resulting link gradients are each negative $\nabla f_3(l_4) = \nabla f_8(l_9) = -1$. Thus, we add two new traffic shapers $l_8$ and $l_9$ to throttle the rate of flows $f_3$ and $f_8$, respectively, down from their current rates of 6.875 and 11.25. That is: $c_{l_8} = 6.875 - \rho$ and $c_{l_9} = 11.25 - \rho$, for some traffic shaping amount $\rho$. In Fig. 7c we show the resulting bottleneck structure when choosing a value of

![Figure 5: Optimizing bandwidth tapering on a 3-level binary fat-tree.](image)

![Figure 6: Network configuration used in Section 3.3.](image)
flows in performs at a rate lower than the slowest flow in the initial accelerates the rate of flow produces a non-positive impact on the rate of flow since the gradients negative effect or no effect at all on the rate of flow to reduce the rate of flow computed by this algorithm (Fig. 7) tell us that choosing of flow \(\rho = 5.625\) (so \(c_{l_4} = 1.25\) and \(c_{l_6} = 5.625\)), which further accelerates the rate of flow \(f_7\) to \(r_7 = s_{l_4} - \rho \cdot \nabla f_7(l_4) = s_{l_6} - \rho \cdot f_6(l_6) = 11.25 - 5.625 \cdot (-1) = 16.875\). Note that there is some flexibility in choosing the value of this parameter, depending on the amount of acceleration required on flow \(f_1\). In this case, we chose a value that maximally accelerates flow \(f_7\) while ensuring none of the flows that are traffic shaped receives a rate lower than any other flow. With this configuration, flow \(f_3\)'s rate is reduced to the lowest transmission rate among all flows in the network, but this value is no lower than the rate of flows \(f_5\) and \(f_6\) (\(r_{f_5} = r_{f_6} = 1.25\)). Thus, the flow completion time of the slowest flow is preserved throughout the transformations performed in this example.

In summary, a strategy to accelerate the performance of flow \(f_7\) consists in traffic shaping the rates of flows \(f_3, f_4\) and \(f_6\) down to 1.25, 1.875 and 5.625, respectively. Such a configuration results in a theoretical increase to the rate of flow \(f_7\) from 10.25 to 16.875, while ensuring no flow performs at a rate lower than the slowest flow in the initial network configuration. Note that among all the low priority flows in \(F^i\), in the above process we opted for not reducing the rate of flow \(f_1\). Indeed, the three bottleneck structures computed by this algorithm (Fig. 7) tell us that choosing to reduce the rate of flow \(f_1\) would in fact have either a negative effect or no effect at all on the rate of flow \(f_7\), since the gradients \(\nabla f_1(f_7)\) for each structure are 2, 0 and 1, respectively—that is, a reduction on the rate of flow \(f_1\) produces a non-positive impact on the rate of flow \(f_7\) in all cases. Thus, the quantitative analysis resulting from the bottleneck structure of the network reveals not only the set of flows that should be traffic shaped, but also the flows that should not be traffic shaped, as doing so would actually hurt the performance of the flow we intend to accelerate. Note that this result challenges some of the established best practices for traffic engineering flows, which include many proposed algorithms that focus on reducing the rate of the heavy-hitter flows to improve high-priority flows. As shown in this example, without taking into account the bottleneck structure of a network, such algorithms may recommend a traffic shaping configuration that actually has the opposite of the intended effect.

To empirically demonstrate the accuracy of QTBS in identifying the set of traffic shapers and their optimal rate, we reproduce the experiments described in this section using Mininet-G2. Fig. 8 illustrates the performance of the flows for each of the three traffic shaping configurations shown in Fig. 7 using the BBR congestion control algorithm. The legends in these figures describe the flows, where the notation \(h_x - h_y\) means that the flow goes from host \(h_x\) to host \(h_y\). To map the flows according to Fig. 6, we use the convention that host \(h_x\) is located in data center \(x\). For instance, flow \(h_8 - h_{12}\) in Fig. 8 corresponds with flow \(f_7\) in Fig. 6, which starts at datacenter 8 and ends at datacenter 12. Table 2 shows the average transmission rate obtained for each of the flows and for each of the three experiments. Next to each experimental rate value, this table also includes the theoretical flow transmission rate according to the bottleneck structure. It is easy to see that these values match the transmission rate \(r_f\) shown next to each flow vertex (gray vertices) from the corresponding bottleneck structures in Fig. 7.

Fig. 8a shows the results of running the initial network without any traffic shapers, corresponding to the bottleneck structure in Fig. 7a. From Table 2 we see that all experimentally measured flow rates usually track their theoretical value from slightly below. Such an offset between experimental and theoretical rates is a characteristic that holds for all experiments, and is due to imperfections in the distributed nature of the congestion control algorithm (e.g., due to its inability to instantaneously converge to the optimal transmission rate or due to statistical packet drops produced by the asynchronous nature of the network). However, the table clearly demonstrates that the experimental rates behave according to the bottleneck structure of the network. This result is also reinforced by the fact that Jain’s fairness index is above 0.99 for all experiments, as shown in Appendix C.

Fig. 8b shows the result of adding the first traffic shaper, configured to reduce the rate of flow \(f_3\) by an amount of 0.5 Mbps. As predicted by QTBS, this increases the rate of flow \(f_7\) (the purple flow \(h_8 - h_{12}\) in Fig. 8), in this case from 9.51 to 9.81 Mbps (Table 2). Fig. 8c shows the result of adding two additional traffic shapers to reduce the rate of flows \(f_3\) and \(f_4\) by an amount of 5.625 Mbps, according to our quantitative analysis of the bottleneck structure. Recall that this configuration was designed to ensure a maximal increase in the rate of flow \(f_7\) without decreasing any of the flows’ rate below the rate of the slowest flow. We see this behavior in Fig. 8c where flow \(f_7\) (purple flow) has now the highest rate, while the flow completion time of the slowest flow remains at slightly above 400 seconds, throughout the three experiments (Fig. 8a, 8b and 8c). In summary, the combined effect of the three traffic shapers accelerates the observed rate of flow \(f_7\) from 9.51 to 15.34 Mbps. As shown in Table 2, this result closely matches the behavior predicted by the bottleneck structure—that the rate would increase from 10.25 to 16.87 Mbps, while the observed maximum flow completion time of the network remains constant throughout the three experiments.
(b) Traffic shaping \( f_4 \).

Figure 8: Flow performance obtained when deploying the traffic shapers to accelerate flow \( f_7 (h_8 - h_{12}) \).

| Flow  | Experiment 1 | Experiment 2 | Experiment 3 |
|-------|--------------|--------------|--------------|
| \( f_1 \) | 2.44 / 2.37  | 2.57 / 2.87  | 2.65 / 2.87  |
| \( f_2 \) | 4.78 / 5.12  | 5.16 / 5.62  | 5.33 / 5.62  |
| \( f_3 \) | 6.99 / 7.37  | 6.57 / 6.87  | 1.18 / 1.25  |
| \( f_4 \) | 2.72 / 2.37  | 1.74 / 1.87  | 1.73 / 1.87  |
| \( f_5 \) | 1.18 / 1.25  | 1.33 / 1.25  | 1.29 / 1.25  |
| \( f_6 \) | 1.42 / 1.25  | 1.19 / 1.25  | 1.19 / 1.25  |
| \( f_7 \) | 9.51 / 10.25 | 9.81 / 11.25 | 15.34 / 16.87|
| \( f_8 \) | 11.48 / 12.25 | 11.06 / 11.25 | 5.27 / 5.62 |

4. Related Work

The problem of congestion control is one of the most widely studied areas in data networks. The first congestion control algorithm for the Internet was introduced by Jacobson in [1] and implemented as part of the TCP protocol, initiating a more than three-decade long period of intense research. This has resulted in a long list of congestion control algorithms (e.g., [10], [7]), including the BBR algorithm recently proposed by Google [2]. All of these algorithms are based on the belief that the performance of a flow is not taken into account the system-wide properties of a network. QTBS addresses this gap and provides a methodology to numerically estimate flow throughput. In future research we plan to incorporate the effects of latency and packet loss to QTBS.

The concept of bottleneck structure was recently introduced in [3]. That work focused on the qualitative properties of the bottleneck precedence graph (BPG), a structure that organizes the relationships among links. Our work focuses on the analysis of a bottleneck structure called the gradient graph. The key difference between the gradient graph and the BPG is that the gradient graph describes the relationships among flows and links, not just links, providing a more comprehensive view of the network. As a result, the gradient graph provides a framework to quantify the interactions among flows and links, resulting in a new class of algorithms to optimize network performance. To the best of our knowledge, this paper presents the first quantitative theory of the analysis of bottleneck structures in data networks.

The problem of traffic engineering (TE) has also been widely studied and continues to be a very active area of research and development. Because QTBS provides a new approach to network optimization, it can be used co-located with existing TE frameworks, augmenting them with detailed information about the interactions among bottleneck links and flows. For instance, in [4], Google introduces Bandwidth Enforcer (BwE), a centralized bandwidth allocation infrastructure for wide area networking that targets high network utilization. QTBS complement tools like BwE by providing a view of the network's bottleneck structure (for online or offline analysis) and providing traffic shaping and flow routing recommendations such as those presented in Sections 3.3 and 3.4.

5. Conclusions

The analytical strength of a bottleneck structure stems from its ability to capture the solution-space produced by a congestion-control algorithm taking into account the topological and routing constraints of the network. Based on this concept, we develop a quantitative theory of bottleneck structures (QTBS), a new mathematical framework that allows to optimize congestion-controlled networks by providing very efficient algorithms to compute derivatives on the performance parameters of links and flows. To explore the analytical power of QTBS, we use it to reveal insights in traffic engineering and network design problems that are themselves contributions to the literature. In one experiment, we use QTBS to develop a novel routing algorithm that identifies maximal throughput paths, enabling a scalable methodology to jointly solve the problems of routing and congestion control. In another experiment, we use QTBS to reveal the existence of optimal capacity allocations in
the spine links of a fat-tree network that outperform (in cost and/or performance) the traditional full fat-tree network designs found in some large-scale data centers and supercomputers. In a third experiment, we demonstrate how to use bottleneck structures to compute the numerical values of optimal rate settings in traffic shapers to help improve the performance of high-priority flows. This paper positions the concept of bottleneck structures as a promising analytical framework to optimize network performance.

References

[1] V. Jacobson, “Congestion Avoidance and Control,” SIGCOMM computer communication review, vol. 18, no. 4, pp. 314–329, August 1988. [Online]. Available: https://doi.org/10.1145/52335.52356

[2] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR: Congestion-Based Congestion Control,” ACM Queue, vol. 14, no. 5, pp. 50:20–50:53, October 2016. [Online]. Available: http://doi.acm.org/10.1145/3012426.3022184

[3] J. Ros-Giralt, A. Bohara, S. Yellamraju, M. H. Langston, R. Lethin, Y. Jiang, L. Tassiulas, J. Li, Y. Tan, and M. Veeraraghavan, “On the bottleneck structure of congestion-controlled networks,” Proc. ACM Meas. Anal. Comput. Syst., vol. 3, no. 3, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3366707

[4] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarande-Stavila, M. Robin, A. Siganporia, S. Stuwart, and A. Vahdat, “BwE: Flexible, hierarchical bandwidth allocation for wan distributed computing,” SIGCOMM Comput. Commun. Rev., vol. 45, no. 4, p. 1–14, Aug. 2015. [Online]. Available: https://doi.org/10.1145/2829988.2878478

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, ser. SIGCOMM ‘08. New York, NY, USA: Association for Computing Machinery, 2008, p. 63–74. [Online]. Available: http://doi.org/10.1145/1402958.1402967

[6] C. E. Leiserson, “Fat-trees: universal networks for hardware-efficient supercomputing,” IEEE transactions on Computers, vol. 100, no. 10, pp. 892–901, 1985.

[7] S. Ha, I. Rhee, and L. Xu, “CUBIC: A New TCP-friendly High-speed TCP Variant,” SIGOPS operating systems review, vol. 42, no. 5, pp. 64–74, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[8] D. P. Bertsekas and R. G. Gallager, Data Networks. Englewood Cliffs, New Jersey 07632: Prentice-Hall Inc., 1992, vol. 2.

[9] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan, “Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability,” Journal of the Operational research society, vol. 49, no. 3, pp. 237–252, March 1998. [Online]. Available: https://doi.org/10.1057/palgrave.jors.2600523

[10] K. Fall and S. Floyd, “Simulation-based Comparisons of Tahoe, Reno and SACK TCP,” SIGCOMM Computer Communication Review, vol. 26, no. 3, pp. 5–21, July 1996. [Online]. Available: http://doi.acm.org/10.1145/235160.235162

[11] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015.

[12] R. D. Neidinger, “Introduction to automatic differentiation and matlab object-oriented programming,” SIAM Rev., vol. 52, no. 3, p. 545–563, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1137/080743627

[13] (2019) Mininet-g2: Mininet extensions to support the analysis of the bottleneck structure of networks. [url]. [Online]. Available: https://github.com/reservoirlabs/g2-mininet

[14] Mininet. (2019) Mininet: An instant virtual network on your laptop (or other pc). [Online]. Available: http://mininet.org

[15] R. Jain, D.-M. W. Chiu, and W. R. Hawe, “A quantitative measure of fairness and discrimination for resource allocation in shared computer systems,” CoRR, vol. cs.NI/9809099, 1998. [Online]. Available: http://arxiv.org/abs/cs.NI/9809099

[16] S. H. Y. Rekhter, T. Li, “A border gateway protocol 4 (bgp-4),” Internet Engineering Task Force (IETF), Request for Comments, 2006.

[17] A. Eryilmaz and R. Srikant, “Joint congestion control, routing, and mac for stability and fairness in wireless networks,” IEEE Journal on Selected Areas in Communications, vol. 24, no. 8, pp. 1514–1524, 2006.

[18] J. Liu and H. D. Sherali, “A distributed newton’s method for joint multi-hop routing and flow control: Theory and algorithm,” in 2012 Proceedings IEEE INFOCOM, 2012, pp. 2489–2497.

[19] L. Tassiulas and A. Ephremides, “Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks,” IEEE Transactions on Automatic Control, vol. 37, no. 12, pp. 1936–1948, 1992.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Potusievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Höltle, S. Stuwart, and A. Vahdat, “B4: Experience with a Globally-Deployed Software Defined WAN,” SIGCOMM Computer Communication Review, vol. 43, no. 4, pp. 3–14, August 2013. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486019

[21] G. Michelogiannakis, Y. Shen, M. Y. Teh, X. Meng, B. Aivaz, T. Groves, J. Shallf, M. Glick, M. Ghabadi, L. Dennison, and K. Bergman, “Bandwidth steering in hpc using silicon nanophotonics,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356145

[22] W. E. Denzel, J. Li, P. Walker, and Y. Jin, “A framework for end-to-end simulation of high-performance computing systems,” Simulation, vol. 86, no. 5-6, pp. 331–350, 2010.

[23] E. A. León, I. Karlin, A. Bhaete, S. H. Langer, C. Chambreau, L. H. Howell, T. D’Hooge, and M. L. Leininger, “Characterizing highly parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree,” in SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2016, pp. 909–920.

[24] E. Martelli and S. Stancu, “Uicpsn and lhcme: status and future evolution,” in Journal of Physics: Conference Series, vol. 664, no. 5, IOP Publishing, 2015, p. 052025.

[25] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, “The macroscopic behavior of the tcp congestion avoidance algorithm,” SIGCOMM Comput. Commun. Rev., vol. 27, no. 3, pp. 67–82, Jul. 1997. [Online]. Available: http://doi.acm.org/10.1145/263932.264023

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, Third Edition, 3rd ed. The MIT Press, 2009.

[27] N. R. POX, “The pox network software platform,” 2019, https://noxrepo.github.io/pox-doc/html/, Last accessed 09/24/2019.

[28] Iperf.fr. (2019) iperf - the ultimate speed test tool for tcp, udp and scpt. [Online]. Available: https://iperf.fr/

[29] P. Phaal, S. Panchen, and N. McKee, “sFlow Specifications,” InMon Corporation, 2012, pp. 2489–2497.

[30] B. Claive, G. Sadasivan, V. Valuri, and M. Djamens. (2004) Netflow specifications, cisco systems. [Online]. Available: https://www.ietf.org/rfc/rfc3954.txt

[31] J. Case, M. Fedor, M. Schoffstall, and J. Davin. (1990) A simple network management protocol (snmp). [Online]. Available: https://tools.ietf.org/html/rfc1517
6. Appendices

Appendix

1. Generalization to Max-min Fairness

Lemma 3. If a link is a bottleneck in the max-min sense \[8\], then it is also a bottleneck according to Definition \[2\] but not vice-versa.

Proof. Bertsekas and Gallager \[8\] proved that if a flow \( f \) is bottlenecked at link \( l \) in the max-min sense, then such a flow must traverse link \( l \) and its rate is equal to the link’s fair share, \( r_f = s_l \). Since a change in the capacity of a link always leads to a change in its fair share, i.e. \( \partial s_l / \partial c_l \neq 0 \), this necessarily implies \( \partial r_f / \partial c_l \neq 0 \). Thus, \( f \) is also bottlenecked at link \( l \) in the sense of Definition \[2\]. The reverse, however, does not hold because Definition \[2\] does not require that \( r_f = s_l \) for a flow \( f \) bottlenecked at link \( l \). (It can be seen that this is also true for other definitions of bottleneck. For instance, a flow that is bottlenecked at a link according to proportional fairness \[9\], is also bottlenecked according to Definition \[2\] but the reverse is also not true.) \( \square \)

2. Proof of Theorem 1: Propagation of Network Perturbations

Let \( x, y \in L \cup F \) be a pair of links or flows in the network. Then a perturbation in the capacity \( c_x \) (for \( x \in L \)) or transmission rate \( r_x \) (for \( x \in F \)) of \( x \) will affect the fair share \( s_y \) (for \( y \in L \)) or transmission rate \( r_y \) (for \( y \in F \)) of \( y \) if only if there exists a directed path from \( x \) to \( y \) in the gradient graph.

Proof. Consider the case \( x = l \in L \) and assume link \( l \) is affected by a perturbation in its capacity. From Definition \[2\], we have that \( \partial r_f / \partial c_l \neq 0 \), for any flow \( f^* \) bottleneck at link \( l \). From Definition \[4\] these correspond to all flows \( f^* \) for which there exists an edge \((l, f^*)\) in \( G \). Let \( f_1 \) be any of these flows and assume \( \Delta r_f \) is its drift. Such drift will induce a perturbation in all the links traversed by \( f_1 \). From Definition \[4\] this corresponds to all the links \( l^* \) for which there exists an edge \((l_1, f_1)\) in \( G \). This process of perturbation followed by a propagation repeats itself, affecting all the link and flow vertices that can be reached from link \( l \) through a directed path in the gradient graph, and ending at leaf vertices. This demonstrates the sufficient condition of the theorem. The necessary condition is also true because, from the definition of region of influence (Definition \[5\]), none of the links and flows outside \( R(l) \) will be affected by the perturbation. The proof for the case \( x = f \in F \) follows the same argument except that the initial perturbation is applied on the rate of flow \( f \). \( \square \)

3. Proof of Theorem 2: Correctness of Gradient-Graph()

Let \( N = \langle L, F, \{c_l, \forall l \in L\} \rangle \) be a network and let \( G \) be the corresponding gradient graph. Let \( x \in L \cup F \).

After running Algorithm \[2\], \( \Delta s_l = \nabla_x(l) \) for all \( l \in L \), and \( \Delta r_f = \nabla_x(f) \) for all \( f \in F \).

Proof. Assume without loss of generality that \( x \in L \). Our goal is to prove:

\[
\frac{\partial r_f}{\partial c_x} = \Delta r_f \quad \forall f \in F \\
\frac{\partial s_l}{\partial c_x} = \Delta s_l \quad \forall l \in L
\]

This is equivalent to showing that, for sufficiently small \( \delta \), if we perturbed the capacity of link \( x, c'_x = c_x + \delta \), and recomputed all the flows’ rates and links’ fair shares using \( \text{GradientGraph()} \), we would get:

\[
r'_{f} = r_f + \Delta r_f \cdot \delta \quad \forall f \in F \\
s'_l = s_l + \Delta s_l \cdot \delta \quad \forall l \in L
\]

Let \( y \in L \cup F \). There are two cases. First, assume \( y \notin R(x) \), where \( R(x) \) is the region of influence of \( x \), then by definition there is no directed path from \( x \) to \( y \). The algorithm only processes vertices that lie on a directed path from vertex \( x \) to \( y \). Thus, if \( y \in L, \Delta s_y = 0 \) (line \[3\]) and if \( y \in F, \Delta r_y = 0 \) (line \[2\]). Moreover, by Theorem 1, \( y \) is not affected by the perturbation of \( c_x \). That is, \( s'_l = s_l \) or \( r'_f = r_f \). Thus the equations hold.

Now let \( y \in R(x) \). We proceed by induction. As a base case, let \( y = x \). The amount of leftover bandwidth at node \( x \), which is called \( a_x \) in the \( \text{GradientGraph()} \) Algorithm, is the capacity \( c_x \) minus the rates of vertices that lie outside the region of influence of \( x \), since they are not bottlenecked at \( x \) (see \( \text{GradientGraph()} \) lines \[3\] and \[16\]). Thus \( a'_x = a_x + (c'_x - c_x) = a_x + \delta \). The fair share rate of \( x \) is \( s_x = a_x / |F_x \setminus \forall| \) (line \[17\] of \( \text{GradientGraph()} \)), where \( V \) is the set of vertices that were added to the gradient graph before \( x \). These necessarily lie outside the region of influence of \( x \), so this set is the same after the perturbation. Indeed, \( F_x \setminus \forall \) is the set of flows which are bottlenecked at \( x \), which is also the set of successors of \( x \). Thus,

\[
s'_x = \frac{a_x + \delta}{|F_x \setminus \forall|} = s_x + \frac{\delta}{\sigma(x, G)} = s_x + \Delta s_x \cdot \delta
\]

where \( \sigma(x, G) \) corresponds to the set of successors of node \( x \) as indicated by the gradient graph \( G \). Thus, the equations hold for \( x \).

Now assume the equations hold for all vertices which are added to the new gradient graph prior to vertex \( y \). First, assume \( y \in F \). Then \( r'_y \) will be the minimum fair share rate of the links it traverses, that is, \( \min s'_l \) for all \( l \in L_y \). However, it suffices to take the minimum over links which were predecessors (bottlenecks) of \( y \) in the original gradient graph (these links’ fair shares were strictly smaller than those of the other vertices that \( y \) traverses, so after an infinitesimally small perturbation they will remain strictly smaller). Thus,

\[
r'_y = \min_{l \in \pi(y, G)} s'_l = \min_{l \in \pi(y, G)} (s_l + \Delta s_l \cdot \delta)
\]

15
where \( \pi(y, \mathcal{G}) \) corresponds to the set of predecessors of node \( y \) as indicated by the gradient graph \( \mathcal{G} \). Now we can substitute \( s_l = s_l + \Delta s_l \cdot \delta \) because of the induction hypothesis. If \( y \) had multiple bottleneck links in the original graph, then they all had the same fair share. That is, \( s_l = r_y \) for all \( l \in \pi(y, \mathcal{G}) \), so

\[
    r'_y = r_y + \left[ \min_{l \in \pi(y, \mathcal{G})} \Delta s_l \right] \cdot \delta
\]

Combining this with line 23 of ForwardGraph(),

\[
    \Delta r_y = \min_{l \in \pi(y, \mathcal{G})} \Delta s_l
\]

\[
\Rightarrow r'_y = r_y + \Delta r_y \cdot \delta
\]

which is what we wanted to prove under the assumption that \( y \in \mathcal{F} \). Now, assume \( y \in \mathcal{L} \). The old fair share of \( y \) was its available capacity (called \( a_y \) in GradientGraph()) divided by the number of flows that it bottlenecked, \(|\sigma(y, \mathcal{G})|\). That is,

\[
    s_y = \frac{a_y}{|\sigma(y, \mathcal{G})|} = \frac{c_y - \sum_{f \in \mathcal{F}_y \setminus \sigma(y, \mathcal{G})} r_f}{|\sigma(y, \mathcal{G})|}
\]

(See GradientGraph() lines 3, 16, and 17). The new fair share is

\[
    s'_y = \frac{a'_y}{|\sigma(y, \mathcal{G}')|} = \frac{c'_y - \sum_{f \in \mathcal{F}_y \setminus \sigma(y, \mathcal{G}')} r'_f}{|\sigma(y, \mathcal{G}')|}
\]

The capacity of \( y \) has not changed (unless \( y = x \), which we already considered) so \( c'_y = c_y \). As a result of the perturbation, some flows which used to be bottlenecked at \( y \) may no longer be (if they were perturbed directly, or if they had a second bottleneck that was affected). But no new bottlenecked flows were created, since flows that were not bottlenecked at \( y \) before had rates strictly smaller than \( s_y \), and the perturbation is infinitesimally small. Let

\[
    T_y = \sigma(y, \mathcal{G}) \setminus \sigma(y, \mathcal{G}')
\]

be the flows that were bottlenecked at \( y \), but whose rate was reduced an infinitesimal amount by the perturbation, so that they no longer are. Then with some algebraic manipulation,

\[
    s_y = a_y
    \Rightarrow s'_y = \frac{a_y - |T_y|s_y}{|\sigma(y, \mathcal{G})| - |T_y|} = \frac{|\sigma(y, \mathcal{G})|s_y - |T_y|s_y}{|\sigma(y, \mathcal{G})| - |T_y|} = s_y
\]

Therefore,\( \delta \) is processed, all the nodes in \( \mathcal{T} \) heap have at most \( \log |R(\mathcal{F})\) time is updating the heap (lines 24 and 28). Since the only operation that is not constant-time is updating the heap (lines 15 - 16). The only operation that is not constant-time is updating the heap (lines 24 and 28). Since the heap has at most \(|R(\mathcal{F})|\) elements, each of these operations takes \( \log |R(\mathcal{F})| \), so the total runtime of the algorithm is \( O(|R(\mathcal{F})| \cdot \log |R(\mathcal{F})|) \).
5. Proof of Lemma 1: Time Complexity of GradientGraph()

The time complexity of running GradientGraph() is $O(|L| \log |L| \cdot H)$, where $H$ is the maximum number of flows that traverse a single link.

Proof. Note that each statement in the algorithm runs in constant time except for lines 5, 8 and 18. Each is an operation on a heap of size at most $|L|$, so each will run in $\log |L|$ time. Lines 5 and 8 will each run $|L|$ times, since the two outer loops run at most once for each link. Line 18 will run at most once for every pair of a link with a flow that traverses it. Note that this value is less than the number of edges that are added to the gradient graph in lines 10 and 15. Thus, the number of times line 18 is run is bounded by $|L| \cdot H$, where $H$ is the number of maximum number of flows that traverse a single link. Thus, in total, the algorithm runs in time $O(H|L|\log(|L|))$. \hfill \qed

6. Proof of Property 1: Gradient Bound

Let $N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\})$ be a network and let $G$ be its gradient graph. Let $\delta$ be an infinitesimally small perturbation performed on a flow or link $x \in \mathcal{L} \cup \mathcal{F}$, producing a drift $\Delta_x$, for all $y \in \mathcal{L} \cup \mathcal{F}$. Then, $\nabla_x(y) = \Delta_x/\delta \leq d^{O(\delta)/4}$, where $D(X)$ is the diameter of a graph $X$ and $d$ is the maximum indegree and outdegree of any vertex in the graph.

Proof. From the invariants of the flow and link equations, we observe that the absolute value of a perturbation can only increase when traversing a link vertex. This is because the flow equation $\Delta_f = \min_{l \in F} \Delta_l$, necessarily implies that the size of the perturbation will either stay the same or decrease. The link equation $\Delta_l = -\sum_{f \in F} \Delta_f/|S_l|$, however, allows perturbations to grow in absolute value. This will happen whenever the sum of the flow drifts arriving at a link vertex is larger than the outdegree of such vertex: $\sum_{f \in F} \Delta_f > |S_l|$. The size of the perturbation will in fact maximally increase when the link outdegree is 1 and the sum of the flow drifts arriving at it is maximal. This is achieved when the bottleneck structure is configured with flows having an outdegree of $d$ and having an indegree of $d$, connected by a stage of inter-medium links and flows of indegree and outdegree equal to 1, as shown in Fig. A1. Concatenating this bottleneck structure block, we have that at each block the perturbation increases $d$ times. Because the length of this block is 4, there are a maximum of $D(G)/4$ blocks, where $D(G)$ is the diameter of the gradient graph. This leads to the upper bound $\nabla_x(y) = \Delta_x/\delta \leq d^{O(\delta)/4}$. \hfill \qed

7. Lemmas to Demonstrate the Correctness of the MaxRatePathAlgorithm

Lemma 4. Flow rate decay with incremental hop count.

Let $N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\})$ be a network and let $r_f$ be the transmission rate of a flow $f \in \mathcal{F}$. Let $r_f'$ be the new transmission rate of flow $f$ after we extend it to traverse an additional link $l^* \in \mathcal{L} \setminus f$. Then, $r_f' \leq r_f$.

Proof. Fig. A2-a shows the initial situation of the lemma, with flow $f$ bottlenecked at a link $l$. Since the transmission rate of flow $f$ is $r_f$, we have that $s_l = r_f$. Suppose that we extend flow $f$ to traverse an extra link $l^* \in \mathcal{L} \setminus f$. Now consider the next set of transformations applied on the network:

1) Create a new flow $f^*$ configured to only traverse link $l^*$.
2) Add a traffic shaper $l_s$ to flow $f^*$ and set its rate to zero, i.e. $s_{l_s} = c_{l_s} = 0$.
3) Increase the rate of the traffic shaper until either (a) $l^*$ becomes a bottleneck of $f^*$ or (b) $s_{l_s} = s_l$.
4) Connect flows $f$ and $f^*$ together.
5) Remove the traffic shaper $l_s$.

It is easy to see that the above process yields a bottleneck structure that is the same as if flow $f$ had been extended to traverse the extra link $l^*$, since at the end of these steps the two flows $f$ and $f^*$ are merged into a single flow (effectively extending flow $f$ to traverse the additional link $l^*$) and the traffic shaper is removed.

Let us now derive the bottleneck structure of the network after applying the above transformations. Steps (1) and (2) are shown in Fig. A2-b. In step (3), as we increase the capacity of the traffic shaper $s_{l_s}$, the rate of flow $f^*$ increases at the same pace. Suppose that condition (3-a) holds so that $l^*$ becomes a bottleneck of $f^*$. This situation is shown in Fig. A2-c.1. Since flow $f^*$ is bottlenecked at both links $l^*$ and $l_s$, we have that $s_{l_s} = s_{l_s}$. Because
If \( l^* \) becomes a bottleneck of \( f^* \), then \( l^* \) is the new bottleneck of flow \( f^* \) and \( s_{l^*} = \) Increment \( l_s \) until either \( l^* \) becomes a bottleneck of \( f^* \) or \( s_{l_s} = s_{l_x} \); If instead \( s_{l_x} = s_{l_s} < s_{l^*} \), then \( l_x \) continues to be the bottleneck; Connect the two flows together;

(a) (b) (c.1) (c.2)

Let \( N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) be a network and let \( r_f \) be the transmission rate of a flow \( f \in \mathcal{F} \). Let \( r'_{f} \) be the new transmission rate of flow \( f \) after we extend it to traverse an additional link \( l^* \in \mathcal{L} \setminus f \). If, \( r'_{f} < r_f \), then the newly extended flow is bottlenecked at link \( l^* \).

Corollary 1. New bottleneck with incremental hop count. Let \( N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) be a network and let \( r_f \) be the transmission rate of a flow \( f \in \mathcal{F} \). Let \( r'_{f} \) be the transmission rate of flow \( f \) after we extend it to traverse an additional link \( l^* \in \mathcal{L} \setminus f \). If, \( r'_{f} < r_f \), then the newly extended flow is bottlenecked at link \( l^* \).

Proof. This result is a direct consequence of Lemma 4 since the condition \( r'_{f} < r_f \) corresponds to case (3-a), which leads to the bottleneck structure in Fig. A2c.2.

Lemma 2. Let \( N = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}) \) be a network and \( \mathcal{U} \) the set of its routers. Suppose that \( f \) and \( f' \) are two flows not in \( \mathcal{F} \) that originate at router \( u_s \) and end at router \( u_d \). Then \( f = \text{MaxRatePath}(N, \mathcal{U}, u_s, u_d) \) implies \( r_f \geq r'_{f} \).

Proof. Consider the network configuration in Fig. A3 and assume that routing data from router \( u_s \) to \( u_x \) using flow \( f_1 \) leads to a higher transmission rate than using flow \( f_2, r_{f_1} > r_{f_2} \). Since the MaxRatePath algorithm uses the inverse of the rate as the path cost metric, this implies that \( d_{f_1} < d_{f_2} \), where we use the notation \( d_f \) to denote the cost of using flow \( f \) to route traffic through the network. To demonstrate the correctness of the algorithm, we need to show that \( d_{f_1} < d_{f_2} \) implies \( d_{f_1 \cup \{r*\}} \leq d_{f_2 \cup \{r*\}} \), since this condition is enough to demonstrate convergence in the Dijkstra algorithm [25].

We will assume that \( d_{f_1} < d_{f_2} \) and \( d_{f_1 \cup \{r*\}} > d_{f_2 \cup \{r*\}} \) are both true and arrive at a contradiction. From Lemma 4 we have that \( d_{f_2} \leq d_{f_2 \cup \{r*\}} \). This implies that \( d_{f_1 \cup \{r*\}} > d_{f_2 \cup \{r*\}} \). Using Corollary 1 it must be that \( l^* \) is the bottleneck of the flow \( f_1 \cup \{r*\} \). Now since flow \( f_2 \cup \{r^*\} \) also traverses link \( l^* \), it must be that its rate cannot be higher than that of flow \( f_1 \cup \{r^*\} \). But this implies \( d_{f_1 \cup \{r^*\}} \leq d_{f_2 \cup \{r^*\}} \), arriving at a contradiction. 

The Mininet-G2 tool [13] provides a powerful, flexible interface to emulate networks of choice with customizable topology, routing and traffic flow configurations. It uses Mininet [14] and the POX SDN controller [27] to create such highly customizable networks. It also uses iPerf [28] internally to generate network traffic and offers an interface to configure various flow parameters such as the source and destination hosts, start time, and data size, among others. Mininet-G2 also offers an integration with sFlow-RT [29] agent that enables real-time access to traffic flows from Mininet emulated network. Since Mininet uses real, production grade TCP/IP stack from the Linux kernel, it proves to be an ideal testbed to run experiments using congestion control protocols such as BBR and Cubic to study bottleneck structures and flow performance in a realistic way. Apart from its flexible configuration interface, Mininet-G2 also offers a set of useful utilities to compute and plot various performance metrics such as instantaneous network throughput, flow convergence time, flow completion time, Jain’s fairness index among others for a given experiment.

Tables 3, 4 and 5 and Fig. A4 and A5 present the results for the experiments described in Sections 3.3, 3.1 and 3.2 when using Cubic as the congestion control protocol.

Jain’s index [15] is a metric that rates the fairness of a set of values \( x_1, x_2, ..., x_n \) according to the following equation:

\[
\mathcal{J}(x_1, x_2, ..., x_n) = \left( \frac{\sum_{i=1}^{n} x_i^2}{n \cdot \sum_{i=1}^{n} x_i} \right) = \frac{x^2}{\bar{x}^2}
\]
(c) Traffic shaping

Figure A4: Traffic shaping schedule to accelerate flow $f_7$ ($h_8 - h_{12}$) (TCP Cubic).

(a) Without any traffic shaping.

(b) Traffic shaping $f_4$.

(c) Traffic shaping $f_3$, $f_4$, and $f_8$.

Figure A5: Optimizing bandwidth tapering on a 3-level binary fat-tree (TCP Cubic).

(a) Tapering parameter $\tau = 1$.

(b) Tapering parameter $\tau = 4/3$.

(c) Tapering parameter $\tau = 2$.

TABLE 3: Experimental versus theoretical average flow transmission rate (units in Mbps) for Section 3.3 when using TCP Cubic.

| Flow | Experiment 1 | Experiment 2 | Experiment 3 |
|------|--------------|--------------|--------------|
| $f_1$ | 3.91 / 2.37 | 5.10 / 2.37 | 4.43 / 2.37 |
| $f_2$ | 5.26 / 5.12 | 6.39 / 5.62 | 5.94 / 5.62 |
| $f_3$ | 6.83 / 7.37 | 6.17 / 6.87 | 1.04 / 1.25 |
| $f_4$ | 2.74 / 2.37 | 1.40 / 1.87 | 1.38 1.87 |
| $f_5$ | 1.09 / 1.25 | 1.15 / 1.25 | 1.14 / 1.25 |
| $f_6$ | 2.04 / 1.25 | 2.10 / 1.25 | 2.01 / 1.25 |
| $f_7$ | 10.22 / 10.25 | 10.49 / 11.25 | 14.4 / 16.87 |
| $f_8$ | 10.62 / 12.25 | 10.58 / 11.25 | 5.37 / 5.62 |

TABLE 4: Experimental versus theoretical average flow transmission rate (units in Mbps) for Section 3.1 when using TCP Cubic.

| Flow | Shortest path | Longer path | Flow | Shortest path | Longer path |
|------|--------------|------------|------|--------------|------------|
| $f_1$ | 0.921 / 1.428 | 0.962 / 1.666 | $f_{14}$ | 1.841 / 1.666 | 1.884 / 1.666 |
| $f_2$ | 1.206 / 1.666 | 1.278 / 1.666 | $f_{15}$ | 1.284 / 1.666 | 1.330 / 1.666 |
| $f_3$ | 1.202 / 1.428 | 1.315 / 1.666 | $f_{16}$ | 1.294 / 1.666 | 1.292 / 1.666 |
| $f_4$ | 0.897 / 1.428 | 0.9085 / 1.666 | $f_{17}$ | 2.035 / 2.142 | 2.132 / 2.142 |
| $f_5$ | 1.186 / 1.428 | 1.336 / 1.666 | $f_{18}$ | 2.097 / 2.142 | 2.141 / 2.142 |
| $f_6$ | 2.227 / 3.000 | 2.126 / 2.500 | $f_{19}$ | 3.792 / 2.142 | 4.065 / 2.142 |
| $f_7$ | 1.716 / 1.428 | 1.966 / 1.666 | $f_{20}$ | 2.101 / 2.142 | 2.115 / 2.142 |
| $f_8$ | 1.211 / 2.142 | 1.333 / 1.666 | $f_{21}$ | 2.195 / 3.000 | 2.170 / 2.500 |
| $f_9$ | 1.000 / 2.142 | 1.022 / 2.142 | $f_{22}$ | 4.168 / 3.000 | 3.767 / 2.500 |
| $f_{10}$ | 1.291 / 1.666 | 1.296 / 1.666 | $f_{23}$ | 2.189 / 3.000 | 2.046 / 2.500 |
| $f_{11}$ | 1.411 / 2.142 | 1.397 / 2.142 | $f_{24}$ | 2.242 / 3.000 | 2.104 / 2.500 |
| $f_{12}$ | 0.984 / 2.142 | 0.999 / 2.142 | $f_{25}$ | 1.679 / 1.428 | 1.377 / 2.500 |
| $f_{13}$ | 1.276 / 1.666 | 1.252 / 1.666 | |

TABLE 5: Flow completion time (seconds) for Section 3.2 when using TCP Cubic.

| Flow | $\tau = 1$ | $\tau = 4/3$ | $\tau = 2$ | $\tau = 4/3$ | $\tau = 2$ |
|------|------------|-------------|-------------|-------------|-------------|
| $f_1$ | 108 | 120 | 139 | 206 | 166 | 156 |
| $f_2$ | 208 | 149 | 135 | $f_8$ | 210 | 143 | 146 |
| $f_3$ | 187 | 166 | 164 | $f_9$ | 102 | 107 | 140 |
| $f_4$ | 80 | 127 | 139 | $f_{10}$ | 190 | 163 | 158 |
| $f_5$ | 176 | 170 | 159 | $f_{11}$ | 220 | 161 | 131 |
| $f_6$ | 206 | 162 | 145 | $f_{12}$ | 80 | 149 | 138 |

The index value ranges from $\frac{1}{\tau}$ (worst case) to 1 (best case). As suggested in [15], for multi-link networks the value $x_i$ must be normalized to an optimal fairness allocation. Throughout this paper, we normalize $x_i$ as the ratio $f_i / O_i$, where $f_i$ is the rate of flow $f_i$ achieved through the experiments and $O_i$ is its expected max-min fair throughput. This provides an index that qualitatively measures how closely the rates obtained from the experiments are to the theoretical rates predicted by the bottleneck structure of the network. The closer this index is to 1, the more accurate the mathematical model is to the experimental results. Table 7 shows the Jain’s fairness index we obtained for all the experiments presented in this paper (Sections 3.3, 3.1 and 3.2). Table 7 presents the specific route configurations used for various experiments in Sections 3.1 and 3.2. Fig. A6 shows the performance of flow $f_{25}$ in the experiment presented in Section 3.1. As expected, its performance is very similar to flow $f_{25}$ shown in Fig. 3a.

19
TABLE 7: Path followed by each flow in the routing optimi-
TABLE 8: Path followed by each flow in the fat-tree net-
TABLE 9: Experimental versus theoretical average flow transmis-
Figure A6: Acceleration of flow $f_{50}$ by routing it through

tion rate (units in Mbps).
production networks. If that is not the case, then the set of
links traversed by each flow can also be derived by looking
up the routing tables, for instance using a BGP collector [16]
or traceroute-like route discovery applications. The set of
links $\mathcal{L}$ and their capacity $\{c_l, \forall l \in \mathcal{L}\}$ can be derived from
protocols like SNMP [31] or simply from network topology
information usually available to the network operator.

| Algorithm | Experiment 1 | Experiment 2 | Experiment 3 |
|-----------|---------------|---------------|---------------|
| BBR       | 0.9926        | 0.9983        | 0.9983        |
| Cubic     | 0.9353        | 0.9074        | 0.9218        |

algorithm 3.2: Experiment 1 3.3: Experiment 2 3.3: Experiment 3

| Flow | Links traversed | Flow | Links traversed |
|------|-----------------|------|-----------------|
| $f_1$ | $(l_1,l_2,l_3,l_4)$ | $f_4$ | $(l_1,l_2,l_3,l_4)$ |
| $f_2$ | $(l_1,l_2,l_3,l_4)$ | $f_7$ | $(l_1,l_2,l_3,l_4)$ |
| $f_3$ | $(l_1,l_2,l_3,l_4)$ | $f_10$ | $(l_1,l_2,l_3,l_4)$ |
| $f_4$ | $(l_1,l_2,l_3,l_4)$ | $f_13$ | $(l_1,l_2,l_3,l_4)$ |

| Flow | Links traversed | Flow | Links traversed |
|------|-----------------|------|-----------------|
| $f_1$ | $(l_1,l_2,l_3,l_4)$ | $f_4$ | $(l_1,l_2,l_3,l_4)$ |
| $f_2$ | $(l_1,l_2,l_3,l_4)$ | $f_7$ | $(l_1,l_2,l_3,l_4)$ |
| $f_3$ | $(l_1,l_2,l_3,l_4)$ | $f_10$ | $(l_1,l_2,l_3,l_4)$ |
| $f_4$ | $(l_1,l_2,l_3,l_4)$ | $f_13$ | $(l_1,l_2,l_3,l_4)$ |

Table 9 presents the performance of flows $f_{26}$-$f_{50}$ for
the experiment described in Section 3.1.

To construct the gradient graph of a network, the only
information about a network $\mathcal{N} = (\mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\})$
is needed. The set of flows $\mathcal{F}$ can be obtained from
traditional network monitoring tools such as NetFlow [30] or
sFlow [29]. For each flow, the GradientGraph() procedure
(Algorithm 1) needs to know the set of links it traverses.
This information can also be obtained from NetFlow or
sFlow provided that traffic sampling is performed at all the
switches and routers of a network, as is often the case with

| Flow | Experiment 1,2,3-Links traversed |
|------|---------------------------------|
| $f_1$ | $(l_1,l_2)$ |
| $f_2$ | $(l_1,l_2,l_3,l_4)$ |
| $f_3$ | $(l_1,l_2,l_3,l_4)$ |
| $f_4$ | $(l_1,l_2,l_3,l_4)$ |
| $f_5$ | $(l_1,l_2,l_3,l_4)$ |
| $f_6$ | $(l_1,l_2,l_3,l_4)$ |
| $f_7$ | $(l_1,l_2,l_3,l_4)$ |
| $f_8$ | $(l_1,l_2,l_3,l_4)$ |
| $f_9$ | $(l_1,l_2,l_3,l_4)$ |
| $f_{10}$ | $(l_1,l_2,l_3,l_4)$ |
| $f_{11}$ | $(l_1,l_2,l_3,l_4)$ |
| $f_{12}$ | $(l_1,l_2,l_3,l_4)$ |

| Flow | Experiment 1,2,3-Links traversed |
|------|---------------------------------|
| $f_1$ | $(l_1,l_2)$ |
| $f_2$ | $(l_1,l_2,l_3,l_4)$ |
| $f_3$ | $(l_1,l_2,l_3,l_4)$ |
| $f_4$ | $(l_1,l_2,l_3,l_4)$ |
| $f_5$ | $(l_1,l_2,l_3,l_4)$ |
| $f_6$ | $(l_1,l_2,l_3,l_4)$ |
| $f_7$ | $(l_1,l_2,l_3,l_4)$ |
| $f_8$ | $(l_1,l_2,l_3,l_4)$ |
| $f_9$ | $(l_1,l_2,l_3,l_4)$ |
| $f_{10}$ | $(l_1,l_2,l_3,l_4)$ |
| $f_{11}$ | $(l_1,l_2,l_3,l_4)$ |
| $f_{12}$ | $(l_1,l_2,l_3,l_4)$ |