The Augmentation-Speed Tradeoff for Consistent Network Updates

Monika Henzinger
monika.henzinger@univie.ac.at
Department of Computer Science,
University of Vienna
Vienna, Austria

Ami Paz
ami.paz@LISN.fr
LISN - CNRS
& Paris-Saclay University
Paris, France

Arash Pourdamghani
pourdamghani@tu-berlin.de
TU Berlin
Berlin, Germany

Stefan Schmid
stefan.schmid@tu-berlin.de
TU Berlin & Fraunhofer SIT
Berlin, Germany

ABSTRACT

Emerging software-defined networking technologies enable more adaptive communication infrastructures, allowing for quick reactions to changes in networking requirements by exploiting the workload's temporal structure. However, operating networks adaptively is algorithmically challenging, as meeting networks' stringent dependability requirements relies on maintaining basic consistency and performance properties, such as loop freedom and congestion minimization, even during the update process. This paper leverages an augmentation-speed tradeoff to significantly speed up consistent network updates. We show that allowing for a small and short (hence practically tolerable, e.g., using buffering) oversubscription of links allows us to solve many network update instances much faster, as well as to reduce computational complexities (i.e., the running times of the algorithms). We first explore this tradeoff formally, revealing the computational complexity of scheduling updates. We then present and analyze algorithms that maintain logical and performance properties during the update. Using an extensive simulation study, we find that the tradeoff is even more favorable in practice than our analytical bounds suggest. In particular, we find that by allowing just 10% augmentation, update times reduce by more than 32% on average, across a spectrum of real-world networks.

CCS CONCEPTS

• **Theory of computation** → *Complexity classes*; *Network optimization*; **Network flows**; • **Networks** → **Network control algorithms**; **Programmable networks**.

KEYWORDS

Software-defined networking, network algorithms, scheduling

1 INTRODUCTION

To render communication networks more dependable, the networking community currently makes great efforts to automate network operations. The envisioned "self-driving" networks [11] can relieve operators of their most complex tasks, hence minimizing the chances for human errors, which frequently are the cause of major outages [5, 7, 43]. Furthermore, more automated networks enable more adaptive network operations, allowing to quickly react to network events such as shifts in the demand and hence to exploit temporal structure in the traffic patterns for optimizations [4, 32, 44]. These more automated and adaptive network operations are enabled, among others, by emerging software-defined and programmable networking technologies, that allow direct control over the forwarding tables of switches and routers.

A programmatic and software-defined control and update of forwarding paths can be attractive in many situations [14]. For example, fast route updates can be useful in reacting to security policy changes or security threats, by actively rerouting traffic through a firewall. In wide-area networks, Internet Service Providers may adjust their traffic engineering policy in reaction to changes in the load. Adaptions to the routes taken by packets may also be required to react to link failures or to support maintenance work or service relocations.

However, a more adaptive network operation introduces an algorithmic challenge: in order to meet the stringent dependability and performance requirements, networks need to be reconfigured *quickly and consistently*. A key challenge here is that updates at different switches occur asynchronously, and update times can vary significantly, between milliseconds to fractions of a second [20, 24]. Especially when adaptions are frequent, it is important that the network fulfill certain properties, such as congestion freedom and loop freedom, *even during the update*.

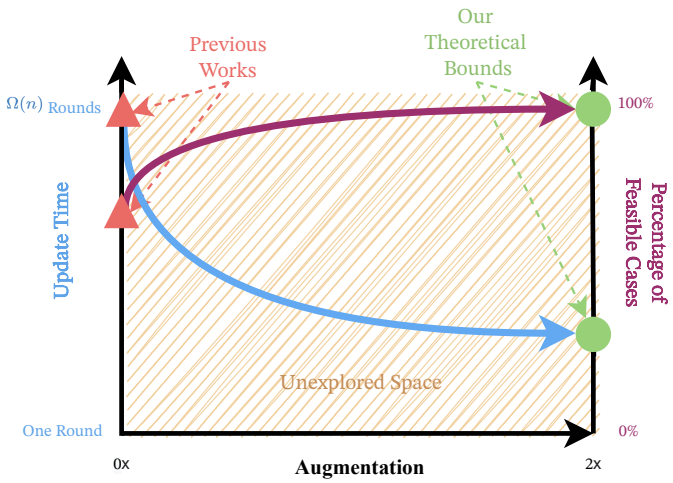


Figure 1: This paper explores the benefits of augmentation on the speed and feasibility of network updates. While prior work (red triangles on the left) did not consider augmentation (the dashed orange area), our approach with augmentation provides flexibility for operators and hence supports more fine-grained network operations. The green circles represent our theoretically optimal bounds that we proved in this paper, and curves represent the tradeoff that we saw in our empirical results, as can also be seen in the counterpart of this qualitative plot in the evaluation section (Figure 4).

Over the last few years, the consistent network update problem has received much attention in the literature. Seminal work focused on logical properties [15, 35–38, 42], but also performance aspects received much attention early on [6, 8, 14, 33]. While the question of how to reroute flows in a congestion-free manner is still not well-understood algorithmically (especially if one requires that algorithms come with provable performance and approximation guarantees), it has been shown recently that the resulting update schedules can be long and complex, requiring many rounds of updates [2, 35].

This paper is motivated by the observation that already a small and short oversubscription of links (which we will refer to as augmentation) can lead to significantly faster update schedules (see Figure 1 for an illustration). This, in turn, may allow more fine-grained network operations. Such augmentation is often feasible in practice and mitigated by buffering and congestion control if the augmentation is bounded in magnitude and time. Short oversubscriptions are common today in congestion control, especially unproblematic in virtual networks, which provide soft capacity constraints, and typically do not affect prices [30].

1.1 Our Contributions

This paper argues that existing literature on network update scheduling ignores the fact that short link oversubscriptions are unproblematic due to buffering, and uncovers an interesting tradeoff between the tolerable oversubscription and the speed at which networks can be updated, measured by the number of rounds in the rerouting schedule. In general, several challenges might occur when trying to update a network's routing policy: infeasibility, i.e., cases where an update schedule without overloading any link simply does not exist; speed, i.e., the time it takes for the updates to complete; and computability, i.e., cases where an update schedule exists, but finding a feasible or optimal update schedule is NP-hard. We show that all these challenges can be overcome by allowing a small oversubscription of the communication links. In this paper, we will distinguish between additive and multiplicative *augmentation* of a link. Additive augmentation refers to the maximum capacity increase on any given link in the network, and multiplicative augmentation is a factor by which we multiply capacities.

We first explore this tradeoff analytically and show that under a factor-2 multiplicative augmentation, fast update schedules is always feasible; we also show that for smaller factors, the problem of computing update schedules is NP-hard—i.e., a short schedule may exist, but finding it is computationally infeasible in the worst case. We then present both optimal and fast algorithms to exploit the augmentation-speed tradeoff while provably maintaining basic consistency properties. We report on an extensive simulation study using real-world networks, and we find that we can reach higher speeds with a slight augmentation. More precisely, a 10% augmentation can reduce update times by 32% on average across a range of networks derived from the Internet Topology Zoo. As a contribution to the research community, we release our experimental artifacts as well as our simulation code (as open source) together with this paper at github.com/inet-tub/AugmentRoute.

1.2 Organization

The remainder of this paper is organized as follows. In §2, we introduce our formal model and define the properties which need to be maintained transiently. §3 details the analytical study of the valid update schedules under augmentation and derives hardness results. Then we present two polynomial-time greedy algorithms and an optimal algorithm based on mixed integer programming in §4, and explore their behavior on real-world networks in §5. After reviewing the related work in §6, we conclude our contribution in §7.

2 MODELLING CONSISTENT NETWORK UPDATES AND TRADEOFFS

We model a network as a directed graph $G = (V, E)$. The set V consists of n nodes representing the switches in the network, and the set $E \subseteq V \times V$ of m directed edges denoting the links. A directed edge $e = (v, w) \in E$ connects the *tail* node v to the *head* node w. An edge e has a real-valued positive capacity $c_e \in \mathbb{R}_{\geq 0}$, and $C_{\max} = \max_e c_e$ is the maximum edge capacity among all edges.

Flows are unsplittable and routed along unique paths, which are dictated by the network's routing policy. When the policy changes, it may become necessary to update the routing path by updating the outgoing edges of the nodes (i.e., the forwarding rules). We consider flow pairs consisting of an *old* flow (which is already in use) and an *updated* flow (which the new policy enforces). We emphasize that flows in each flow pair share the same source and terminal node. We also consider that the forwarding is done based on *both* the source and the terminal.

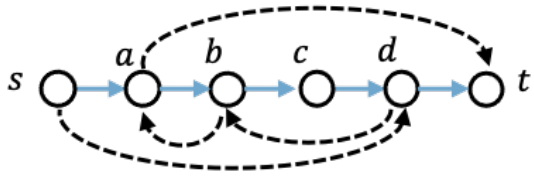
Definition 1 (Flow pairs). *A set of k* flow pairs *is defined as $P = \{P_1, \ldots, P_k\}$, where each flow pair P_i consists of an old flow F_i^o and an updated flow F_i^u. Flows of the i^{th} flow pair are* unsplittable*, which means each is only a simple s_i-t_i path in G. Each flow corresponds to a real-valued positive demand d_i, initiating from the same source node s_i and ending at the same terminal node t_i.*

In order to update a flow pair P_i from and old flow F_i^o to an updated flow F_i^u, we need to update all the nodes that are appearing only in F_i^o or only in F_i^u. An *update schedule* from the old flow F_i^o to the updated flow F_i^u is a sequence of *update rounds*. In each round, a subset of the nodes changes their outgoing edges from the edges used in F_i^o to the edges used in F_i^u. Formally, for a flow pair P_i, we define an R-round update schedule $U_i = \{U_i^1, \ldots, U_i^R\}$ such that in each round $r = 1, \ldots, R$, a set of nodes that were not included in previous updates, i.e., $U_i^r \subseteq V(F_i^o \cup F_i^u) \setminus (U_i^1 \cup \cdots \cup U_i^r - 1)$, are updated. We assume that all the changes in the same round happen asynchronously, i.e., in an unpredictable order. This makes the problem harder, as a worst-case order inside each update set must be taken into consideration.

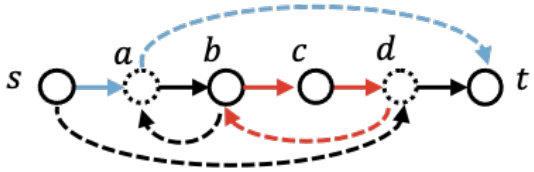
To maintain consistency during network updates, an update schedule must provide *loop freedom* and *congestion freedom*.

2.1 Loop Freedom

When scheduling batches of updates simultaneously, individual updates at nodes can happen at different times, which might cause transient *forwarding loops*, see Figure 2 for an example. *Loop freedom* requires that forwarding loops never happen during the update of a flow pair, regardless of whether nodes in the loop can be reached from the source node or not.



(a) Initial flow



(b) After updating nodes a and d

Figure 2: Solid lines represent the old flow, dashed lines the updated flow. Solid circles show nodes that are not updated, and dotted circles are updated nodes. (a) Initially, a flow passes through blue lines. (b) After the update a and d, a transient loop appears (red lines) that violates loop-freedom property, even though the terminal is still reachable (through blue lines).

In the case of nodes that are in the old path but not in the updated path, $F_i^o \setminus F_i^u$, their routing policy is updated from edge to no-edge. Since we need to ensure reachability at all times, we update these nodes after the nodes leading to them. Similarly, nodes in $F_i^u \setminus F_i^o$ initially have not been assigned an outgoing edge and need to be updated before the nodes leading to them. Note that together with loop-freedom, handling these cases guarantees that a path from the source to the terminal node exists at all times: Each node that has an incoming edge has a single outgoing edge (except for the terminal node), and there are no loops in the graph.

2.2 Congestion Freedom

Assume that for each flow pair, we have an update schedule that is valid and loop-free. We want to make sure that we can apply all the updates simultaneously without causing congestion on the network links. For this, fix an update round r in all flow pairs, and for each flow pair i, consider the temporary flow F_i, that includes edges from old flow F_i^o and edges from the updated flow F_i^u that pass flow "during" round r, i.e. edges from old flow that has not been updated "before" round r, edges from the updated flow that will be used "after" round r.

Definition 2 (Valid schedule). *An update schedule is* valid *if at any time, the set of temporary flows $F = \{F_1, \ldots, F_m\}$*

satisfies that for every edge $e \in E$, the sum of demands of flows that pass through an edge is at most its capacity, i.e. $\forall e \in E : \sum_{j:e\in F_j} d_j \leq c_e$.

This paper is motivated by the benefits of slight augmentation of the current capacity. We investigate two possibilities for augmentation: *multiplicative augmentation* and *additive augmentation*. In the first approach, we consider all capacities multiplied by a real number $\alpha \geq 1$. In the latter one, we allow capacities to be increased by a fixed number $\beta \geq 0$.

Definition 3 (($\times\alpha$)-valid schedule, ($+\beta$)-valid schedule)**.** *For $\alpha \geq 1, \beta \geq 0$, we say that a schedule is (α, β)-*valid *if at any time, the set of temporary flows $F = \{F_1, \ldots, F_m\}$ satisfies that for every edge $e \in E$, we have $\sum_{j:e\in F_j} d_j \leq \alpha c_e + \beta$. A schedule is $(\times\alpha)$-*valid *if it is $(\alpha, 0)$-valid, and $(+\beta)$-*valid *if it is $(1, \beta)$-valid.*

We first show that every update schedule is $(\times 2)$-valid and $(+C_{\max})$-valid, and then prove that for any $\epsilon > 0$, , the two problems of deciding if a $(\times(2-\epsilon))$-valid update schedule exists, and if a $(+(C_{\max}/3-\epsilon))$-valid update schedule exists, are both NP-hard.

3 THEORETICAL ANALYSIS

In this section, we start exploring the speed-congestion tradeoff analytically. We first derive upper bounds on the required additive and multiplicative augmentation that make update schedules valid. We further explore the computational complexity of finding valid update schedules, showing it is NP-hard to decide whether a valid update schedule exists when the augmentation is below some threshold.

3.1 Upper Bounds

The following theorem characterizes the amount of multiplicative and additive augmentation needed to render update schedules valid.

Theorem 1. *Every update schedule is $(\times 2)$-valid and $(+C_{\max})$-valid.*

Proof. Consider an update schedule at any given point in time during update, and an edge e. Let S^o be the set of indices of flows that are not yet updated at this time point at e, and S^u the set indices of flows that are updated; the current load on e is at most $\sum_{i\in S^o} d_i + \sum_{i\in S^u} d_i$.

Since the set of old flows is valid, we have $\sum_{i:e\in F_i^o} d_i \leq c_e$, and similarly the set of updated flows is valid and $\sum_{i:e\in F_i^u} d_i \leq c_e$. Note that $S^o \subseteq \{i : e \in F_i^o\}$ and $S^u \subseteq \{i : e \in F_i^u\}$, and hence $\sum_{i\in S^o} d_i \leq \sum_{i:e\in F_i^o} d_i \leq c_e$ and $\sum_{i\in S^u} d_i \leq \sum_{i:e\in F_i^u} d_i \leq c_e$. Thus, $\sum_{i\in S^o} d_i + \sum_{i\in S^u} d_i \leq 2c_e$, as desired.

For the additive case, note that $2c_e \leq c_e + C_{\max}$, and the proof immediately follows from the multiplicative case. □

These bounds leave the question of how to find a schedule that minimizes the number of rounds, a question that needs to be resolved for each flow pair separately. We explore techniques to minimize the number of rounds later in the paper.

3.2 Hardness Results

We next study the computational complexity and prove a tight converse of the multiplicative case in Theorem 1, and a non-tight converse for the additive case.

Theorem 2. *For every constant $1/3 > \epsilon > 0$, deciding if a $(\times(2-\epsilon))$-valid update schedule exists is NP-hard.*

The proof of this theorem extends a construction from [1], and shows that an algorithm for finding a $(\times(2-\epsilon))$-valid update schedule can be used in order to find a satisfying assignment for a 3-CNF formula of the well-known 3SAT problem [26]. In the 3SAT problem, we should assign 0-1 values to a set of binary variables such that a set of boolean clauses become valid. Each clause consists of only three variables, or their negation.

The details of the proof can be found in Appendix A. Our proof transforms each clause or variable into a series of *gadgets*. A gadget is a series of old and updated paths between a pair of nodes, designed to transform a valid update into a solution for 3SAT.

Theorem 3. *For every constant $1 > \epsilon > 0$ deciding if a $(+(C_{\max}/3-\epsilon))$-valid update schedule exists is NP-hard.*

The proof of the theorem of the additive case goes along the lines of the proof of the multiplicative case of Theorem 4. The proof of the additive case introduces a new variable gadget between pairs of source-terminal nodes of the valid update problem. We present details of the proof in the Appendix A for completeness of the paper.

4 ALGORITHMS

This section presents algorithms that navigate and exploit the tradeoff between speed and augmentation. We first detail a fast algorithm that will be useful for efficient updates in large networks. In addition, we provide an optimal algorithm based on mixed integer programming, which is useful for small networks and will also serve as a baseline in the evaluation of fast algorithms.

4.1 Fast Algorithm for Short Update Schedules

Our first approach is a natural greedy algorithm that computes a short loop-free update schedule. We then extend this algorithm by adding a post-processing step that reduces augmentation by allowing delays in update schedules. These

algorithms are fast enough to be used in most of real-world scenarios.

Algorithm Greedy. For this algorithm, we assume a fixed flow pair i and a certain round. As mentioned in the preliminaries, if a node v is only in the updated flow, it should be updated before nodes that are in both updated and old flows, and if v is only in the old flow, it should be updated after these nodes; Greedy starts by doing exactly that. Thus, we only need to describe how to order the nodes that belong both to the old and the updated flow of the flow pair i.

For each flow pair, we maintain a set A_i of *active* edges, edges that are actively passing the flow for the pair i. Also, define $\overline{A_i}$ as the edges from F_i^u that are not yet updated, and sort them based on their distance to the terminal node in the updated flow F_i^u. Before the first round, A_i consists of all the edges of the old flow, and $\overline{A_i}$ includes all the edges from the updated flow. During each round, we go through the edges of $\overline{A_i}$, starting from the one nearest to the terminal node, and for each edge, if it does not create a directed cycle in the graph induced by A_i, we add the edge to A_i, and remove it from $\overline{A_i}$, then mark its tail to be updated in the current round. Such an edge always exists in each update round: In the first round, the edge of F_i^u nearest to the terminal is directly connected to the terminal, and thus can be updated without creating a cycle. Later, $F_i^u \cap A_i$ is a set of disjoint paths, and one of them leads to the terminal node. On this path, consider the node v that is furthest away from the terminal node; then, F_i^u contains an edge $(u, v) \in \overline{A_i}$, and node u can be updated without creating a cycle (otherwise, the set $\overline{A_i}$ is empty, which means that update has been finished successfully).

After going through all of $\overline{A_i}$, we update A_i by removing all the edges from F_i^o whose tail was updated, and move to the next round. Note that in each round, at least the edge currently closest to the terminal will be added to A_i and removed from $\overline{A_i}$; thus, the algorithm ends after at most $|F_i^u|$ rounds.

Algorithm Delay. We propose an improved algorithm, called Delay, that modifies any valid schedule by delaying selected flow pairs up to T rounds (usually only 1-2 rounds) to reduce augmentation while increasing the number of rounds up to T. Delay operates in phases. In each phase, the algorithm greedily chooses the flow pair P_i and the delay value $d_i \leq T$ such that delaying the start of P_i's update schedule by d_i rounds provides the highest decrease the augmentation. Delay terminates if no such flow pairs exist anymore. Note that the schedule of a flow pair can be delayed in multiple phases.

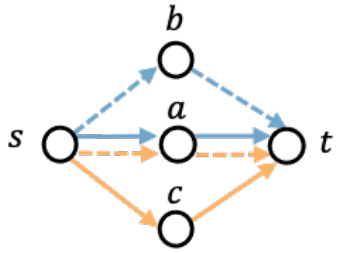


Figure 3: Example where delay reduces the congestion. The first flow pair consists of $F_1^o = (s, a, t), F_1^u = (s, b, t)$ and the second is $F_2^o = (s, c, t), F_2^u = (s, a, t)$; all demands and capacities equal to 1. During the second round of the Greedy algorithm, it is possible that node s gets updated for the second flow pair before it is updated for the first flow pair, in which case the edges (s, a) and (a, t) get congested. This is avoided by the delay algorithm, which shifts the schedule of the second flow pair by one round.

Figure 3 shows the potential benefit of delaying the update of flow pairs. In this example, delaying the bottom flow pair for one round eliminates congestion in the network.

4.2 Optimal Algorithm

We next present an optimal algorithm that is based on a *Mixed Integer Program* (MIP) for finding a loop-free update schedule that minimizes either the number of rounds or the augmentation. For the sake of explanation, we consider a fixed multiplicative (additive) augmentation ratio α (β) and describe the minimization of the number of rounds R. However, given a fixed number of rounds, with a simple change to the MIP, we can minimize the augmentation instead[1].

Variables: Let us fix the flow pair i and the round r. We then describe the set of variables related to a node $v \in V(F_i^o \cup F_i^u)$.

- *Node update variable* $x_{v,i}^r$ is a binary variable that indicates whether node v from flow pair i updates in round r or not.

We define special variables for nodes that appear in both old and updated flows. A node $v \in V(F_i^o \cap F_i^u)$ is a *branching* node if it has an outgoing edge (v, w) in the old flow F_i^o and another outgoing edge (v, w') in the updated flow F_i^u. Similarly, $v \in V(F_i^o \cap F_i^u)$ is a *merging* node if it has an incoming edge (w, v) in the old flow F_i^o and another incoming edge (w', v) in the updated flow F_i^u.

After each branching node v, there exists a merging node by, denoted by v'. The node v' must exit since the old and

[1]One can think of optimizing both at the same time using biconvex optimization.

updated flows need to intersect again, as they share a common terminal node. Furthermore, no other branching node appears on the old and updated flows between v and v', as a branching node needs to be part of both old and updated flows. Hence a branching node v is uniquely matched with a merging node v'.

- *Branching variable* $\Lambda^r_{v,i}$ is a binary variable indicating whether node v is a branching node in the flow pair i and gets updated in round r.
- *Merging variable* $\Upsilon^r_{v,i}$ is a binary variable indicating whether node v is a merging node in the flow pair i and receives flow from both old and updated flows during round r.

To avoid creating loops during an update round r, we assign an order to all nodes in the flow pair i.

- *Ordering variable* $o^r_{v,i}$ is an integer variable assigned to node v of flow pair i in round r in the range of $[1, n]$.

Now we look at the variables for an edge $e = (v, w) \in F^o_i \cup F^u_i$ of the flow pair i. We say edge e is *active* during (after) round r if it appears in F^o_i and its tail v has not been updated, or it is part of F^u_i and v has been updated. If an edge e is active for pair i, it might be part of the source-terminal flow of pair i.

- *Edge activity variable* $y^r_{e,i}$ is a binary variable indicating if edge e is active *after* round r in flow pair i or not.
- *Edge transitivity variable* $\gamma^r_{e,i}$ is a binary variable indicating whether edge e is active *during* round r in the flow pair i or not.
- *Edge flow variable* $f^r_{e,i}$ is fractional variable in the range $(0, 1)$ indicating whether the source-terminal flow of flow pair i passes over e during round r or not.

Constraints: As before, let us fix a flow pair i. We now go through properties that an optimal schedule needs to satisfy.

Node update. Each node in the flow pair i except for the terminal needs to be updated exactly once. Thus, for any given node $v \in V(F^o_i \cup F^u_i)$, the node update variable $x^r_{v,i}$ should be equal to 1 in precisely one of the rounds, and 0 in others. Hence, the sum of $x^r_{v,i}$ over all rounds should be 1, as shown in Constraint 3. Also, the number of rounds is lower bounded by the last round a node updates, implying Constraint 7.

During round r, node v is a branching node if and only if it is updating from an edge (v, w) in the old flow to a different edge (v, w') in the updated flow of flow pair i (see Constraints 16 and 17). Based on our definition, a node is a merging node in round r if it receives flow from two different edges, as shown in Constraints 18 and 19.

Mixed Integer Program to Compute Optimal Solution

1:	**Minimize** R (or α, β)	
2:	**for all** $i \in [\lvert P \rvert]$	
3:	$\sum_{r\in[R]} x^r_{v,i} = 1$	$\forall v \in V(F^o_i \cup F^u_i) \setminus \{t_i\}$
4:	$y^0_{(v,w),i} = 1$	$\forall (v, w) \in F^o_i$
5:	$y^0_{(v,w),i} = 0$	$\forall (v, w) \notin F^o_i$
6:	**for all** $r \in [R]$	
7:	$R \geq r \cdot x^r_{v,i}$	$\forall v \in V(F^o_i \cup F^u_i) \setminus \{t_i\}$
8:	$y^r_{(v,w),i} = 1$	$\forall (v, w) \in F^o_i \cap F^u_i$
9:	$y^r_{(v,w),i} = \sum_{r' \leq r} x^{r'}_{v,i}$	$\forall (v, w) \in F^u_i \setminus F^o_i$
10:	$y^r_{(v,w),i} = 1 - \sum_{r' \leq r} x^{r'}_{v,i}$	$\forall (v, w) \in F^o_i \setminus F^u_i$
11:	**for all** $\forall (v, w) \in F^o_i \cup F^u_i$	
12:	$\gamma^r_{(v,w),i} \geq y^{r-1}_{(v,w),i}$	
13:	$\gamma^r_{(v,w),i} \geq y^r_{(v,w),i}$	
14:	$\gamma^r_{(v,w),i} \leq \frac{o^r_{w,i} - o^r_{v,i} - 1}{\lvert V \rvert - 1} + 1$	
15:	**for all** $\forall v \in P_i$	
16:	$\Lambda^r_{v,i} = x^r_{v,i}$	$\exists (v, w) \in F^o_i \wedge (v, w') \in F^u_i$
17:	$\Lambda^r_{v,i} = 0$	$\nexists (v, w) \in F^o_i \wedge (v, w') \in F^u_i$
18:	$\Upsilon^r_{v,i} \leq f^r_{(w,v),i}, f^r_{(w',v),i}$	$\exists (w, v) \in F^o_i \wedge (w', v) \in F^u_i$
19:	$\Upsilon^r_{v,i} = 0$	$\nexists (w, v) \in F^o_i \wedge (w', v) \in F^u_i$
20:	$f^r_{(v,w),i} \leq \gamma^r_{(v,w),i}$	$\forall (v, w) \in F^o_i \cup F^u_i$
21:	$\sum_{(s_i,v)} f^r_{(s_i,v),i} = 1 + \Lambda^r_{s_i,i}$	$s_i \in P_i$
22:	$\sum_{(v,t_i)} f^r_{(v,t_i),i} = 1 + \Upsilon^r_{t_i,i}$	$t_i \in P_i$
23:	$\sum_{(v,w)} f^r_{(v,w),i} - \sum_{(w',v)} f^r_{(w',v),i} = \Lambda^r_{v,i} - \Upsilon^r_{v,i}$ $\forall v \in v \in V(F^o_i \cup F^u_i) \setminus \{s_i, t_i\}$ $(v, w), (w', v) \in F^o_i \cup F^u_i$	
24:	$\sum_{i\in[\lvert U \rvert]} f^r_{(v,w),i} \cdot d_i \leq \alpha \cdot c_{(v,w)} + \beta$	$\forall (v, w) \in E$

Edge activity. Constraints 4 and 5 guarantee that only edges that are active in the initial round, round 0, are edges from the old flow.

After the round 0, if an edge was part of both old and updated flow, i.e. $e \in F^o_i \cup F^u_i$, edge e remains active (Constraint 8. Otherwise, if an edge e is only in F^o_i, it remains active *until* the round in which its tail v is updated (Constraint 10), and if the edge e is only in F^u_i it becomes active *after* node v updates (Constraint 9).

If the activity of an edge e changes during a round r, it can happen at any time during the round r. Therefore we say edge e is active *during* round r if it was active in the previous round (Constraint 12) or it becomes active after this round (Constraint 13).

Loop freedom. The loop freedom property does not allow any transitive loops to appear at any time during the update. Thus, for all edges $e = (v, w) \in F^o_i \cup F^u_i$ that can be active during around, i.e. $\gamma^r_{(v,w),i} = 1$, we need the order of edge's head, the node w, be always larger than edge's tail, the node v. Inspired by the Miller-Tucker-Zemlin constraint used in

sub-tour elimination in traveling salesman problem [40], we define Constraint 14 that prevents loops from being created. Note that the right side of the inequality assumes a value in $[1, 2)$ if $o^r_{v,i} < o^r_{w,i}$ and a value less than 1, otherwise.

Congestion freedom. Similar to loop-freedom, we need to maintain congestion freedom at any time during the update. Any flow pair i during round r can only use the edges that are active during that round (see Constraint 20).

As flows are unsplittable, if flow pair i uses edge e, it passes all of its demand d_i through it. That is why we can define $f^r_{e,i}$ as a binary value and just multiply them by d_i in Constraint 24, which ensures the augmented capacity of edge e limits the sum of the flows that pass through e.

We know that for any given branching node v in flow pair i, there exists a corresponding merging node u. If node v updates in round r, all edges on both paths between v and u must be active during round r. Hence, edges in both paths are prone to congestion. That is why in round r, from branching node v, we send flow on both paths, and in merging node u, we unify these flows. By Constraint 21, we enforce the source node of flow pair i to send a unit of flow. If the source node is also a branching node, it can temporarily send one additional unit of flow in the round that it updates. Similarly, the terminal node of flow pair i needs to receive one unit flow, unless it is a merging node that receives one additional unit of flow (Constraint 22). For all other nodes of the flow pair i, the incoming flow must match the outgoing flow, unless the node is a branching node that can output an additional unit flow, or whether it is a merging node that decreases the flow by one unit (Constraint 23).

Objectives: We describe two objectives, minimizing the number of rounds given a fixed augmentation and minimizing augmentation given a fixed number of rounds.

Minimizing number of rounds. To enforce loop freedom on each of the flow pairs, we might need up to $n-1$ rounds, $n = |U|$ is the number of nodes in the network [13]. As we saw in the discussion about the delay algorithm, in order to minimize the augmentation, it might be beneficial for us not to update each flow pair in each round. Thus, we consider the possibility that flow pairs update one after the each other, and consider the number of rounds R to be in the range $\{1, \ldots, k \cdot (n-1)\}$, where k is the number of flow pairs. To minimize the number of rounds, we assume that we are given the allowed augmentation, i.e., both α and β, and run the mixed integer program with the objective to minimize R.

Minimizing Augmentation. By Theorem 1, multiplicative augmentation factor α is at most 2. Additive augmentation value β can be between 0 and the maximum capacity of an edge. To minimize the multiplicative or additive augmentation, we assume that we are given the number of rounds R. We change Constraint 6 to iterate only up to R rounds, and also remove Constraint 7. Finally, we change the objective in Line 1 to minimize α (or β) instead of R.

5 EMPIRICAL RESULTS

We complement our analytical results by studying the augmentation-speed tradeoff in practical scenarios and performing an extensive simulation study. We first report our methodology and then present our main insights. The implementation is available at github.com/inet-tub/AugmentRoute.

5.1 Methodology

We implemented our algorithms in python 3.6, using NetworkX 2.5 [18], Numpy 1.19 [19] and Matplotlib 3.3 [22] libraries. To solve the mixed integer program, we used Gurobi 9.1 [17]. We have executed our program on a machine that has Intel Xeons E5-2697V3 SR1XF as CPU and provides 128 GB DDR4 RAM.

Topologies. We have evaluated our algorithms on 218 connected and directed graphs from the Internet Topology Zoo [28] that have at most 100 nodes. We removed trees from the graph set, in which any schedule can update trivially in one round. The limitation on the number of nodes is due to the high running time of the mixed integer program. As the data set only provides the network topology, we now describe how to set the link capacities, flow pairs, and their demands.

Generating Flows. For the flow pair i, we choose two distinct nodes uniformly at random as the source s_i and terminal t_i. If we use the shortest paths, old and updated paths fully overlap with each other, so instead, we extend the shortest path routing with Valiant routing [31] (or more generally, segment routing [12]) to *segment path routing*. In the segment path routing, from nodes other than source and terminal, we randomly choose one of the waypoint nodes, w^1_i, and find the shortest (s_i, w^1_i) and (w^1_i, t_i) paths. The basis of calculating shortest paths is the weights that we assign to each edge, in the range of $(1, 100)$. Since the high running time of MIP is a limiting factor, we use 250 flow pairs in our experiments.

Setting Link Capacities. We construct a set of *baseline flows* used to set the link capacities, with demand chosen uniformly at random from $(10, 20)$. This method is preferable to other methods, such as setting the capacity of each edge independently, since it assigns link capacities proportionally to a possible link usage from old and updated paths.

Setting Demands of Old and Updated Flows. To generate demand for old and updated paths such that both of them

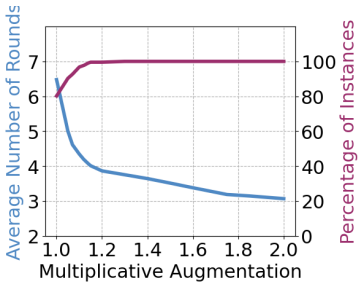


Figure 4: The experimental augmentation-speed tradeoff (blue) and augmentation-feasibility tradeoff (purple). With only 20% augmentation, we observe a sharp decrease from more than 6 rounds to less than 4 rounds on average. With as little as 15% augmentation, the percentage of solvable flow pairs increases from 80% to over 99%.

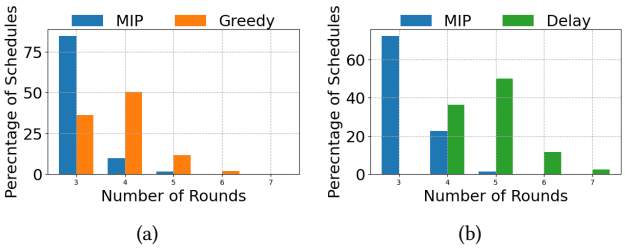


Figure 5: The percentage of cases in which a given number of rounds is required for the optimal algorithm, compared to (a) Greedy and (b) Delay algorithms. The optimal algorithm optimizes the number of rounds, given the multiplicative augmentation resulting from the other algorithm.

remain valid, we use an approach reminiscent of congestion control protocols (and in particular, the slow start algorithm [23, 48]). We also set the initial demands of all old and updated flows to 1, and arranged them in round-robin order. We then multiply the demand by a growth factor g. We stop multiplying the demand of a flow pair if either the old or the updated flow cannot be increased due to link capacity constraints. The value of the growth factor directly affects link utilization, as links get less utilized with an increased growth factor. In our experiments, the default value of the growth factor is 1.1, as it provides more than 99% link utilization.

Running MIP. We first run Greedy and Delay on each input graph, receiving their update schedules. We then calculate the augmentation and the number of rounds that those schedules need. In the end, we run the mixed integer program first based on the number of rounds that each of those algorithms needs, and also based on the augmentation that they require.

5.2 Results

We first study the achievable improvement of optimal update schedules with additional augmentation, then evaluate the performance of our greedy algorithms, and finally describe how our algorithms can be used in other practical settings.

Benefits of Augmentation. To evaluate the benefits of augmentation, we use our optimal algorithm to find the best update schedule given a certain amount of augmentation. We only focus on multiplicative augmentation, as additive augmentation follows a similar trend. Having Theorem 1, it is enough to check augmentations between 1 and 2. We compare instances with different augmentations on two metrics: the average number of rounds which represents the speed of an update schedule, and the percentage of feasible solutions. As shown in Figure 4, the number of rounds drops considerably with a slight increase in augmentation. With only 5% additional augmentation, the number of rounds reduces by more than 22%, and with 10% augmentation, the number of rounds drops by more than 32%. The number of feasible solutions increases by more than 16% with 10% augmentation and grows above 99% while using less than 15% augmentation.

Comparing Optimal and Fast Algorithms. We start by comparing the algorithms when the optimal algorithm is allowed the same augmentation as the Greedy or Delay algorithms, with the delay threshold equal to three. In Figure 5 we compare the percentage of optimal and Greedy (Delay) schedules that have a certain number of rounds. We see that the MIP schedules more than half of the cases in three rounds, and the Greedy and Delay algorithms can have up to 6 and 7 rounds, respectively. The difference between the number of rounds matches our intuition since the cases that Greedy algorithm performs poorly are rare, but they are possible.

We then fix the number of rounds from the Greedy and Delay algorithms and find the required augmentation for the mixed integer program. In Figure 6, we compare the average augmentation given fixed rounds. The top figures show the comparison between multiplicative augmentations as shown on the y-axis, and in the bottom figures, we can see additive augmentations. When the number of rounds is limited, the optimal algorithm needs as high augmentation as Greedy (Delay), but when the number of rounds increases, the optimal algorithm can delay or provide a gap between updates of each flow pair, which leads to lower augmentation.

We compare the running time of our algorithms in Figure 7. As expected from an integer program, the running time grows fast when increasing the number of nodes or flows (the variance is due to the various heuristics applied by the solver). The fast growth of running time of MIP continues

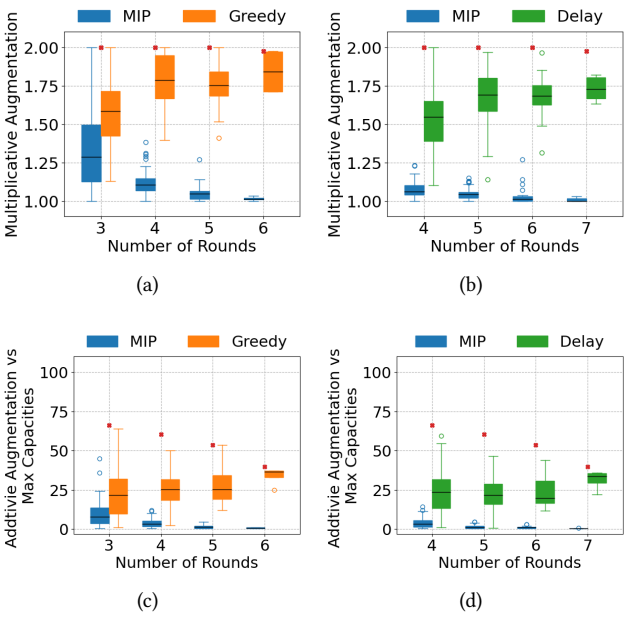


Figure 6: Comparison of an optimal scheduling vs. Greedy (a,c) and Delay (b,d), in terms of multiplicative augmentation (a,b) and additive augmentation as percentage of $C_{\max}$ (c,d). In each case, the MIP uses the number of rounds needed by the other algorithm. The red dots denote the worst-case scenario when all the possible old and updated flows overlap on each edge.

even after 100 nodes. Hence, Greedy and Delay are particularly useful in large networks as well as when the main focus is on speed. On the other hand, the MIP can be attractive to provide minimal augmentation on small networks.

Takeaway for other settings. Given the above results, and based on today's flexible networks, allowing for momentary link augmentations can solve issues, such as long update schedules. Particularly, our recommendation is that in events of long update schedules in practice, optimizing schedules based on efficient algorithms like Greedy or Delay could be enough to reduce the number of rounds significantly.

6 ADDITIONAL RELATED WORK

With the rise of software-defined networks, researchers have started exploring the benefits and challenges of more adaptive network operations on many fronts [27, 45]. The consistent network update problem has already received much attention. We refer to the extensive survey by Foerster et al. [14] for an overview. There also exists interesting empirical studies on the topic of consistent network update, for example, Kuzniar *et al.* [29] showed the high variance in the

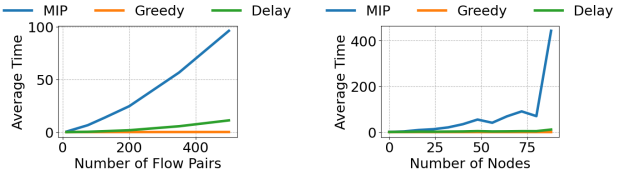


Figure 7: Average time (in seconds) needed to run our algorithms on networks from the Internet Topology Zoo: (a) on graphs with at most 50 nodes and varying numbers of flow pairs; (b) on graphs of varying sizes and 250 flow pairs.

timing of updates in switches. While approaches to maintaining logical properties such as loop-freedom and waypoint enforcement alone are fairly well-understood in the literature, much less is known about algorithms that provably account for performance aspects such as congestion [14].

In the seminal work in the area of consistent network update, Reitblatt *et al.* [42] introduced a fairly general two-phase approach to update networks while preserving reachability to the terminal. Mahajan and Wattenhofer [37] have initiated the study of update mechanisms that do not require packet tagging. They introduced the first approach to maximize the number of links that can be updated in each round. The problem was shown to be NP-hard for a single terminal [3, 16], and the study has been extended for multiple terminals in [16, 49].

When there are multiple routes with different policies, it may further be important to minimize the number of interactions with an individual router [10]. Recently, a few papers have focused on the synthesis of update schedules [9, 25, 39, 46, 47], however, in this paper, we are interested in the natural and well-studied objective of minimizing the number of rounds. Ludwig *et al.* [13, 35] showed that finding a schedule that provides loop freedom in 3 rounds is NP-hard, and there is always a pair of flows that requires $\Theta(n)$ rounds. The authors of [34] presented a mixed integer linear program to compute optimal solutions, however, without considering congestion; our formulation builds upon that and accounts for congestion aspects, as studied in [2] for loop-free scenarios.

On the other hand, relatively little is known about scheduling congestion-free updates, and most existing works revolve around heuristics [14, 21, 41, 50]. Amiri *et al.* [2] presented a first algorithmic result, devising a fixed-parameter tractable algorithm to update a fixed number of flows on directed acyclic graphs. The authors also showed that the problem is NP-hard in general, already for two flows. The authors later extended their results in [1], considering more general but

still acyclic graphs, and focusing on optimal solutions. To the best of our knowledge, we are the first to observe and exploit the augmentation of links to speed up and improve the feasibility of update schedules.

7 CONCLUSION

This paper uncovered an interesting tradeoff between augmentation, speed and feasibility of network updates. We proved that 2 times multiplicative augmentation is sufficient to make any update schedule feasible and provided insight into the complexity of scenarios with lower augmentation. We further presented fast and optimal algorithms for finding consistent update schedules and empirically showed that the tradeoff between augmentation and speed is even better in practice.

In future works, it would be interesting to further explore this tradeoff considering additional consistency properties (such as waypoint enforcement), or to explore extensions to scenarios supporting splittable flows.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101019564 "The Design of Modern Fully Dynamic Data Structures (MoDynStruct)") and from the Austrian Science Fund (FWF) project "Fast Algorithms for a Reactive Network Layer (ReactNet)", P 33775-N, with additional funding from the *netidee SCIENCE Stiftung*, 2020–2024. This project is also supported by the German Federal Ministry of Education and Research (BMBF), 6G-RIC grant 16KISK020K, 2021-2025, as well as by the European Research Council (ERC) under grant agreement No. 864228 (AdjustNet), 2020-2025.

REFERENCES

[1] Saeed Akhoondian Amiri, Szymon Dudycz, Mahmoud Parham, Stefan Schmid, and Sebastian Wiederrecht. 2019. On Polynomial-Time Congestion-Free Software-Defined Network Updates. In *Proc. of the IFIP Networking Conference*.

[2] Saeed Akhoondian Amiri, Szymon Dudycz, Stefan Schmid, and Sebastian Wiederrecht. 2018. Congestion-Free Rerouting of Flows on DAGs. In *Proc. of the ICALP*.

[3] Saeed Akhoondian Amiri, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. 2016. Transiently Consistent SDN Updates: Being Greedy is Hard. In *Proc. of the SIROCCO*.

[4] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. 2020. On the Complexity of Traffic Traces and Implications. *Proc. ACM Meas. Anal. Comput. Syst.* (2020).

[5] Ryan Beckett, Ratul Mahajan, Todd D. Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proc. of the ACM SIGCOMM*.

[6] Sebastian Brandt, Klaus-Tycho Förster, and Roger Wattenhofer. 2016. On consistent migration of flows in SDNs. In *Proc. of the IEEE INFOCOM*.

[7] Richard Chirgwin. 2017. Google routing blunder sent japan's internet dark on friday. *The Register* (2017).

[8] Niels Christensen, Mark Glavind, Stefan Schmid, and Jiří Srba. 2021. Latte: improving the latency of transiently consistent network update schedules. *ACM SIGMETRICS Performance Evaluation Review* (2021).

[9] M. Didriksen, P.G. Jensen, J.F. Jønler, A.-I. Katona, S.D.L. Lama, F.B. Lottrup, S. Shajarat, and J. Srba. 2021. Automatic Synthesis of Transiently Correct Network Updates via Petri Games. In *Proc. of the Petri Nets*.

[10] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. 2016. Can't Touch This: Consistent Network Updates for Multiple Policies. In *Proc. of IEEE/IFIP DSN*.

[11] Nick Feamster and Jennifer Rexford. 2018. Why (and How) Networks Should Run Themselves. In *Proc. of the Applied Networking Research Workshop, ANRW*.

[12] Clarence Filsfils, Nagendra Kumar Nainar, Carlos Pignataro, Juan Camilo Cardona, and Pierre François. 2015. The Segment Routing Architecture. In *Proc. of the IEEE GLOBECOM*.

[13] Klaus-Tycho Foerster, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. 2018. Loop-Free Route Updates for Software-Defined Networks. *IEEE/ACM Trans. Netw.* (2018).

[14] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2019. Survey of Consistent Software-Defined Network Updates. *IEEE Commun. Surv. Tutorials* (2019).

[15] Klaus-Tycho Förster, Ratul Mahajan, and Roger Wattenhofer. 2016. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *Proc. of the IFIP Networking Conference*.

[16] Klaus-Tycho Förster and Roger Wattenhofer. 2016. The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration. In *Proc. of the ICCCN*.

[17] Gurobi Optimization, LLC. 2021. Gurobi Optimizer Reference Manual. "https://www.gurobi.com"

[18] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report.

[19] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* (2020).

[20] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. 2015. Measuring control plane latency in SDN-enabled switches. In *Proc. of 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR*.

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

[22] John D. Hunter. 2007. Matplotlib: A 2D Graphics Environment. *Comput. Sci. Eng.* (2007).

[23] Van Jacobson. 2008. Congestion avoidance and control. In *Proc. of the ACM SIGCOMM.*
[24] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. In *Proc. of the ACM SIGCOMM.*
[25] N.S. Johansen, L.B. Kaer, A.L. Madsen, K.O. Nielsen, J. Srba, and R.G. Tollund. 2022. Kaki: Concurrent Update Synthesis for Regular Policies via Petri Games. In *Proc. of the International Conference on Integrated Formal Methods, iFM'22.*
[26] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proc. of a symposium on the Complexity of Computer Computations (The IBM Research Symposia Series).*
[27] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. 2019. Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges. In *Proc. of the IEEE, PIEEE.*
[28] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Alistair Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE J. Sel. Areas Commun.* (2011).
[29] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. 2015. What You Need to Know About SDN Flow Tables. In *Proc. of the PAM.*
[30] Nikolaos Laoutaris, Georgios Smaragdakis, Pablo Rodriguez, and Ravi Sundaram. 2009. Delay tolerant bulk data transfers on the internet. In *Proc. of the ACM SIGMETRICS/Performance.*
[31] Gavriela Freund Lev, Nicholas Pippenger, and Leslie G. Valiant. 1981. A Fast Parallel Algorithm for Routing in Permutation Networks. *Proc. of the IEEE Trans. Computers* (1981).
[32] Christos Liaskos, Lefteris Mamatas, Arash Pourdamghani, Atsioli Tsioliaridou, Sotiris Ioannidis, Andreas Pitsillides, Stefan Schmid, and Ian F. Akyildi. 2022. Software-Defined Reconfigurable Intelligent Surfaces: From Theory to End-to-End Implementation. In *Proc. of the IEEE (PIEEE).*
[33] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David A. Maltz. 2013. zUpdate: updating data center networks with zero loss. In *Proc. of the ACM SIGCOMM.*
[34] Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. 2016. Transiently Secure Network Updates. In *Proc. of ACM SIGMETRICS.*
[35] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. 2015. Scheduling Loop-free Network Updates: It's Good to Relax!. In *Proc. of the ACM PODC.*
[36] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. 2014. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proc. of the ACM HotNets.*
[37] Ratul Mahajan and Roger Wattenhofer. 2013. On consistent updates in software defined networks. In *Proc. of ACM HotNets.*
[38] Jedidiah McClurg, Hossein Hojjat, Pavol Černỳ, and Nate Foster. 2015. Efficient synthesis of network updates. *ACM Sigplan Notices* (2015).
[39] Jedidiah McClurg, Hossein Hojjat, Pavol Cerný, and Nate Foster. 2015. Efficient synthesis of network updates. In *Proceedings of the ACM SIGPLAN.*
[40] Clair E Miller, Albert W Tucker, and Richard A Zemlin. 1960. Integer programming formulation of traveling salesman problems. *J. of the ACM, JACM* (1960).
[41] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. 2017. Decentralized Consistent Updates in SDN. In *Proc. of ACM SIGCOMM the Symposium on SDN Research, SOSR.*
[42] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *Proc. of the ACM SIGCOMM.*
[43] Duluth News Tribune. 2018. Officials: Human error to blame in Minn. 911 outage. https://www.ems1.com/911/articles/officials-human-error-to-blame-in-minn-911-outage-xjwcEfhzmbDD8tub/
[44] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proc. of the ACM SIGCOMM.*
[45] Karla Saur, Joseph M. Collard, Nate Foster, Arjun Guha, Laurent Vanbever, and Michael W. Hicks. 2016. Safe and Flexible Controller Upgrades for SDNs. In *Proc. of ACM SIGCOMM the Symposium on SDN Research, SOSR.*
[46] Stefan Schmid, Bernhard Schrenk, and Alvaro Torralba. 2022. NetStack: A Game Approach to Synthesizing Consistent Network Updates. In *Proc. IFIP Networking.*
[47] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. 2021. Snowcap: synthesizing network-wide configuration updates. In *Proc. of the ACM SIGCOMM.*
[48] W. Richard Stevens. 1997. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *RFC* (1997).
[49] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre François, and Olivier Bonaventure. 2012. Lossless migrations of link-state IGPs. *IEEE/ACM Trans. Netw.* (2012).
[50] Jiaqi Zheng, Guihai Chen, Stefan Schmid, Haipeng Dai, Jie Wu, and Qiang Ni. 2017. Scheduling Congestion- and Loop-Free Network Update in Timed SDNs. *IEEE J. Sel. Areas Commun.* (2017).

A OMITTED PROOFS

In this section, we detail the proofs of Theorems 4 and 5 that prove the NP-hardness of finding valid schedules with a limited augmentation.

THEOREM 4. *For every constant $1/3 > \epsilon > 0$, deciding if a $(\times(2-\epsilon))$-valid update schedule exists is NP-hard.*

Given a 3-CNF formula and a constant $0 < \epsilon \leq 1/3$, we build a graph and define flow pairs on it. The core of the construction is variable gadgets, one for each variable x_j in the formula, with two flow pairs representing truth values, such that one of them must be updated before the other. A choice to update the true flow first or the false flow first, for each variable, implies a corresponding assignment of the value true or false to the variable, giving a satisfying assignment for the formula. In addition, the construction contains another gadget, that guarantees that the choice of flows to update first indeed satisfies each of the clauses.

The reduction. The reduction graph is composed of the nodes s and t, a pair of nodes u^i, v^i for each clause C^i, a pair of nodes u^i_j, v^i_j for each occurrence of a variable x_j (negated or not) in a clause C^i, and, for each variable x_j, the nodes w^1_j, w^2_j and $w_j(0), \ldots, w_j(\sqrt{a})$, where $a = \left(\left\lfloor \frac{1}{2\epsilon} \right\rfloor + 1\right)^2$. The graph edges and their capacities are defined as part of the flow definitions, next.

The variable gadget. For each variable x_j, we define a gadget — see Figure 8. The gadget is composed of paths from w^1_j to w^2_j, of different capacities and flows.

First, a path $(w^1_j, u^{i_1}_j, v^{i_1}_j \ldots, u^{i_{\max(j)}}_j, v^{i_{\max(j)}}_j, w^2_j)$ from w^1_j, through each edge (u^i_j, v^i_j) such that x_j appears in C^i without

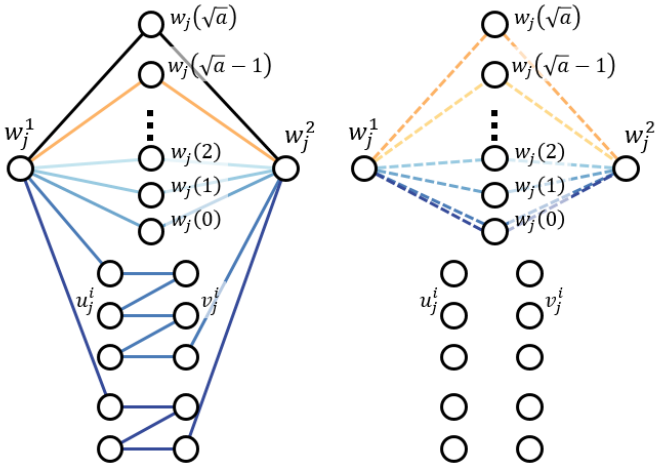


Figure 8: The variable gadget for variable x_j which appears in two clauses without negation and in three clauses negated. Left: original flows; right: updated flows; edges without flows are omitted. The original flows, from bottom up, have sizes $a, a, 2a, 2a+\sqrt{a}, 2a+2\sqrt{a}, \ldots, 2a+(\sqrt{a}-1)\sqrt{a}, 3a$.

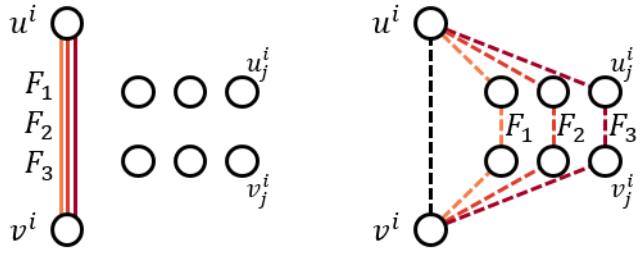


Figure 9: The clause gadget for a clause C^i which contains the variable x_j. Left: original flows; right: updated flows; edges without flows are omitted. The three flows F_1, F_2, F_3 appear in both figures and have demand a each; the updated blocking flow, which appears only on the right, has demand $3a$, and its original flow appears in the variable gadget (Figure 10).

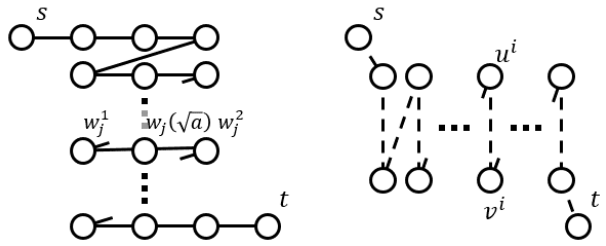


Figure 10: The blocking flow. Left: the old flow, going through all the variable gadgets; right: the updated flow, going through all the clause gadgets. The flow has demand $3a$, and this is also the capacity of all the edges in the figure.

negation (in an arbitrary order), and to w_j^2; and, a similar path through each edge (u_j^i, v_j^i) such that x_j appears in C^i negated. All edges in both paths have capacity a, and initial flows F_{true} and F_{false} respectively, of demand a each. Then, for $k = 0, \ldots, \sqrt{a}$, the gadget contains a path $(w_j^1, w_j(k), w_j^2)$ with edges of capacity $2a + k\sqrt{a}$, and flow $F(k)$ of demand $2a + k\sqrt{a}$ as well; the flow $F(\sqrt{a})$ will also be referred to as the *blocking flow*. The updated flows for both F_{true} and F_{false} are flows through $(w_j^1, w_j(0), w_j^2)$. For $k = 0, \ldots, \sqrt{a}-1$, the updated flow for $F(k)$ is on the path $(w_j^1, w_j(k+1), w_j^2)$, i.e., the old path of the flow $F(k+1)$. The updated flow for $F(\sqrt{a})$ is through a different gadget, described next.

The clause gadget. For a clause C^i, we build a gadget composed of three flows F_1, F_2, F_3, which are initially identical — each flow has demand a, and they all go through the edge (u^i, v^i), which has capacity $3a$. The updated flows for these flows are through the paths (u^i, u_j^i, v_j^i, v^i) for the three variables x_j appearing in the clause C^i (negated or not), one flow on each path; the edges of this path has capacity a. Clauses that contain both a variable and its negation are omitted from the construction, as they are always satisfied — see Figure 9.

In addition, the edge (u^i, v^i) is a part of the updated flow of the blocking flow, in a way described next.

Putting everything together. We now explain how the flows go through the different gadgets. Each of the flows defined above, i.e., $F_{true}, F_{false}, F_1, F_2, F_3$ and the flows $F(k)$ for $k = 0, \ldots, \sqrt{a}$ all start from s, and then traverse all the relevant gadgets in an arbitrary order. For example, the flow F_{true} goes from s to w_j^1 for some j, through the gadget of x_j to w_j^2, then to $w_{j'}^1$ for some j' and so on, and finally from some $w_{j''}^2$ to t. The corresponding updated flow goes through the same gadgets in the same order, and inside each gadget j goes through $(w_j^1, w_j(0), w_j^2)$, as described above. The flows F_{false} and $F(k)$ for $k = 0, \ldots, \sqrt{a}$ are all connected in a similar manner. Each of the flows F_1, F_2, F_3 go from s to u^i for some clause C^i, to v^i through the gadget of C^i, to $v^{i'}$, and so on, until it leaves the last $v^{i''}$ to t. The updated flows are similarly connected. We assume there are edges connecting s and t to the gadgets and between the gadgets, where each edge has enough capacity to contain all the initial and updated flows assigned to it; note that the flows on these edges (except for $F(\sqrt{a})$) are not changed by the updates. The blocking flow $F(\sqrt{a})$ initially traverses all the variable gadgets, and when updated, traverses all the clause gadgets, as depicted

in Figure 10. Except for this blocking flow, each flow can be seen as is split into segments, one for each gadget, and the updates on one gadget are independent the updates in others.

Proof of the reduction. Using the above construction, we show that deciding if a $(\times(2-\epsilon))$-valid update schedule exists is NP-hard.

Proof of Theorem 4. Consider a 3-CNF formula, a constant $0 < \epsilon < 1/3$, and define a, the graph and the flow pairs as described above. We show that the formula has a satisfying assignment if and only if the graph has a $(\times(2-\epsilon))$-valid update schedule.

Assume the formula has a satisfying assignment, and define an update schedule in the following order: in each variable clause, update F_{true} or F_{false} by the assignment; in each variable clause, update F_1, F_2 or F_3 by the variable satisfying the clause; consecutively update $F(k)$, for $k = \sqrt{a}, \ldots, 0$; complete the updates of F_{true} or F_{false}; complete the updates of F_1, F_2 and F_3. We now detail these updates.

In update step one, update in each variable gadget, F_{true} or F_{false} according to the assignment, in parallel. For a variable x_j, this forms a $(\times(2-\epsilon))$-valid flow on $(w_j^1, w_j(0), w_j^2)$: the capacity of this path is $2a$, the increased capacity is $2a + a = 3a \leq 4a - 2\epsilon a$, as the initial capacity on it has demand $2a$, and the updated F_{true} or F_{false} has demand a.

In step two, update in each clause gadget C^i, the flow F_1, F_2 or F_3 that corresponds to the variable x_j that satisfies this clause. The updated flow is legal, as the edges (u^i, u_j^i) and (v_j^i, v^i) have enough capacity (a) and no flow on them, and the edge (u_j^i, v_j^i) has capacity a as well, and after update step one, it has no flow on it. If there is more than one variable satisfying C^i, we can choose any non-empty set of such variables.

Next, update the blocking flow $F(\sqrt{a})$. Each of the edges (u^i, v^i) has capacity $3a$, increased capacity $(2-\epsilon)3a$, and after update step two, there are at most two flows of demand a each on it, making this update valid, as $5a < (2-\epsilon)3a$.

In the next $\sqrt{a}$ update rounds update the flows $F(k)$ for $k = \sqrt{a} - 1, \ldots, 0$, one per round. We will show below that these flows cannot be updated in the same round due to capacity constraints. These updates do not affect the flows between the gadgets. Inside each clause gadget, the capacity of the path $w_j^1, w(k+1), w_j^2$ is always greater than the flow $F(k)$, and this path is always unused when we update $F(k)$, since this update comes after the update of $F(k+1)$.

In update round $\sqrt{a} + 4$ update the flows F_{true} and F_{false} in each variable gadget where it was not updated in the first round. This is now valid since in the last round we updated $F(0)$, so $(w_j^1, w(0), w_j^2)$ has capacity $2a$ and no flow on it.

Finally, in the last update round we update the flows F_1, F_2 and F_3 in the clauses where they are not updated yet. This is possible as the last update round made the edges of the form (u_i^j, v_i^j) free of flow. This shows the existence of a $(\times(2-\epsilon))$-valid update schedule if a satisfying assignment exists.

For the other direction, assume the graph has a $(\times(2-\epsilon))$-valid update schedule. The construction of the flow pairs and the edge capacities implies that the update sequence we define above is the only one possible, which implies a satisfying assignment. We now prove this claim.

We start by showing that only the flows F_{true} and F_{false} can be updated in the initial configuration, and only one of them in each variable gadget. Consider the variable gadget for x_j: Each path $(w_j^1, w_j(k), w_j^2)$ has capacity and flow $2a + k\sqrt{a}$; its excess capacity is thus

$$\begin{aligned}(1-\epsilon)(2a + k\sqrt{a}) &< 2a + k\sqrt{a} - 2\epsilon a \\ &= 2a + (k-1)\sqrt{a} + \sqrt{a}(1 - 2\epsilon\sqrt{a}) \\ &< 2a + (k-1)\sqrt{a}\end{aligned}$$

where the last inequality uses the choice of a. Hence, for all $1 \leq k \leq \sqrt{a}$, the flow $F(k-1)$ cannot be updated. For the flows F_{true} and F_{false}, their updated path $(w_j^1, w_j(0), w_j^2)$ has excess capacity $(1-\epsilon)2a < 2a$, so they cannot both be updated simultaneously.

Consider the clause gadget for C^i. The flows F_1, F_2 and F_3 have a demand of a each, and their updated paths are of the form (u^i, u_j^i, v_j^i, v^i); the central edge (u_j^i, v_j^i) is saturated, and has excess capacity $(1-\epsilon)a$, smaller than the demand of F_1, F_2, and F_3. Finally, the updated blocking flow $F(\sqrt{a})$ uses edges of the form (u^i, v^i); each such edge has capacity $3a$, 3 flows of demand a each, and its excess capacity $(1-\epsilon)3a$ is not enough to accommodate the $3a$ demand of $F(\sqrt{a})$.

By this discussion, we see that the first update round can contain at most one F_{true} or F_{false} update in each variable gadget, and only these updates. Moreover, before any other type of update is made, there must be enough such updates to guarantee that all the clauses are satisfied: if this is not the case, there is a clause C^i such that all its three edges of the form (u_j^i, v_j^i) cannot take any of the flows F_1, F_2 and F_3. This, in turn, implies that the blocking flow $F(\sqrt{a})$ can never be updated. Hence, before the blocking flow is updated, one of F_{true} and F_{false} must be updated in each variable gadget, and then one of F_1, F_2 and F_3 must be updated in each clause gadget. So, a $(\times(2-\epsilon))$-valid update schedule implies as satisfying assignment, as claimed. □

Theorem 5. *For every constant $1 > \epsilon > 0$ deciding if a $(+(C_{\max}/3 - \epsilon))$-valid update schedule exists is NP-hard.*

Proof. Given a 3-CNF formula, consider a graph construction as above, with $a = 0$. ; see Figure 11. That is, for each x_j

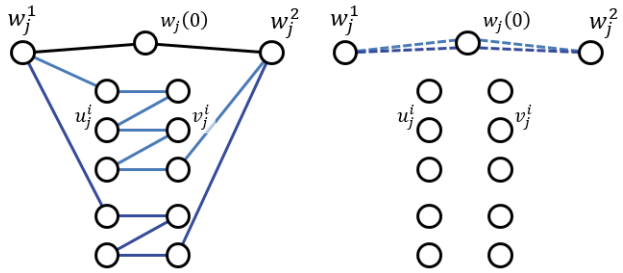


Figure 11: The variable gadget for variable x_j in the additive case.

the only node of the form $w_j(k)$ is $w_j(0)$, and the blocking flow is $F(0) = F(\sqrt{a})$. For F_{true}, F_{false}, their updated flows in an x_j gadget are set to be through the path $(w_j^1, w(0), w_j^2)$.

We now have only six flow pairs $F_{true}, F_{false}, F_1, F_2, F_3$ and $F(0)$, and set all their demands to 1. All the edges in the gadgets have capacity 1, except for the edges of the paths $(w_j^1, w(0), w_j^2)$ with capacity 2, and the edges of the form (u^i, v^i) with capacity 3. Thus $C_{\max} = 3$.

For the edges connecting s and t to the gadgets, and between the gadgets, set the capacity to 3. In addition, the flows F_1, F_2, F_3 traverse the clause gadget in increasing order of identifiers, while the updated blocking flow $F(0)$ traverses them in the opposite order. This ensures that the edges between the gadgets never form a bottleneck or prevent updates. Given a satisfying assignment, the construction of a valid (and thus, a $(+(C_{\max}/3 - \epsilon))$-valid) update schedule is similar to the one in the proof of Theorem 4. In the first update round, in each variable gadget x_j, update either F_{true} or F_{false}, by the assignment. This is possible since the path $(w_j^1, w(0), w_j^2)$ has capacity 2 but only a single flow through it. In the second update round in each clause C^i, update one of the flows F_1, F_2, F_3 which goes through the edge (u_j^i, v_j^i) corresponding to the variable x_j that satisfies the clause. This is possible since the edge (u_j^i, v_j^i) now carries no flow. In the third update round update the blocking flow $F(0)$, which uses the edges (u^i, v^i) that were just now cleared out from one of the flows on them. Thus it has flow 2 and excess capacity $2 - \epsilon$, which is sufficient for the demand of flow $F(0)$. In the forth update round, complete the update of F_{true} and F_{false}, and in the fifth update round complete the update of F_1, F_2 and F_3, all without any additional augmentation in the edges of the graph. For the converse direction, note that $C_{\max}/3 - \epsilon = 1 - \epsilon$. In the initial configuration, paths of the form $(w_j^1, w(0), w_j^2)$ have excess capacity of $1+(1-\epsilon) < 2$, so only one of F_{true} and F_{false} can be updated before $F(0)$. The edges (u_j^i, v_j^i) have just $1-\epsilon$ excess capacity, so none of F_1, F_2 and F_3 can be updated in a clause gadget in which none of the variables was updated. Hence, a $(+(C_{\max}/3 - \epsilon))$-valid update schedule must first update one of F_{true} and F_{false} in each variable clause, in a way that induces a satisfying assignment, as claimed. □