IFIP Networking 2019 1570514789

On Polynomial-Time Congestion-Free Software-Defined Network Updates

Saeed A. Amiri
MPI
Germany

Szymon Dudycz
University of Wroclaw
Poland

Mahmoud Parham
University of Vienna
Faculty of Computer Science
Austria

Stefan Schmid
University of Vienna
Faculty of Computer Science
Austria

Sebastian Wiederrecht
TU Berlin
Germany

***Abstract*—We consider the SDN network update problem in which a controller wants to update the routes of k (unsplittable) flows from their old paths to the new paths, consistently, i.e., without temporary congestion. As updates communicated by the controller take effect asynchronously, the challenge is to perform these updates fast, i.e., using a minimal number of rounds (controller interactions). We present the first fast, i.e., polynomial-time solution for scheduling such congestion-free network updates, for two flows and in the node ordering model. We also show that the problem is already NP-hard for six flows. We complement our formal results with simulations.**

I. Introduction

Emerging software-defined communication networks provide *direct* and "*algorithmic*" control over the forwarding rules of nodes (i.e., routers and switches) and hence the network *routes*. The resulting routes are not restricted to follow only shortest paths and moreover, they can be flexibly *adapted over time*, e.g., depending on certain events in the dataplane. Indeed, there are many reasons why flows may need to be *rerouted* [14], including security and policy changes (e.g., suspicious traffic is rerouted via a firewall), traffic engineering optimizations, reactions to changes in the demand, maintenance work, failures, etc.

Implementing route changes however is challenging, since updating a route usually involves the distribution of new (forwarding) rules across the asynchronous communication network, and since even *during* such route changes, it is important to maintain certain safety properties. In particular, the routes of flows should be changed without causing any congestion, without using packet tagging, and without introducing temporary forwarding loops. For example, in a Software Defined Network (SDN), rules are communicated by the remote software controller. Therefore, updates have to be distributed in *rounds*, in which switches acknowledge the next batch of updates [26, 27, 32]. This approach is known as the *node-ordering approach* [14], which is attractive as it does not require extra packet header and router memory space.

This introduces a *scheduling problem*: In which order to update the different forwarding rules for the different flows and switches over time, such that these safety properties are maintained at any time? And how to schedule these updates such that the rerouting time (and number of controller interactions) is minimized?

ISBN 978-3-903176-16-4

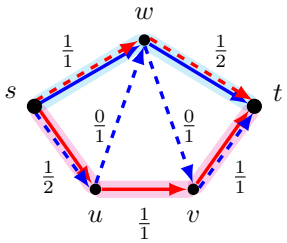

Fig. 1: The flow rerouting problem: Example.

A. A Simple Example

Figure 1 gives an example of the flow rerouting problem. We want to schedule the rerouting of 2 flows in a 5-node network, connecting nodes $\{s, u, v, w, t\}$ with 7 edges $\{\{s, u\}, \{s, w\}, \{u, w\}, \{u, v\}, \{v, w\}, \{v, t\}, \{w, t\}\}$. In this example, both flows originate at s and end at t: The first flow is indicated in red and the second flow in blue.

Each of the two flows has an *original ("old") route* and a *new route*, which it should be updated to. We indicate the original route with a *solid* line and the new route with a *dashed* line. For example, the original route of the red flow is (s, u, v, t) and needs to be updated to (s, w, t). The original route of the blue flow is (s, w, t) and needs to be updated to (s, u, w, v, t).

In other words, each flow defines an *update pair*, consisting of two routes (the original and the new one): Acordingly, updates are denoted using tuples, i.e., (v, B) means that we activate all inactive (dashed) outgoing blue edges (the new *forwarding rules*) of vertex v and deactivate all of its active (solid) outgoing blue edges (the old *forwarding rules*).

In this example, we assume that both flows consume 1 unit of bandwidth on each link they traverse. Both flows are unsplittable. Accordingly, we annotate the network edges in the figure with two numbers, $\frac{x}{y}$, where x denotes the bandwidth consumed by the two flows on the corresponding edge *before* rerouting and y denotes the edge capacity.

How to reroute the two flows from their old paths to their new paths in a congestion-free manner? In this example, initially, we cannot perform the update (s, R), since the red

flow combined with the existing blue flow would violate the capacity on edge (s, w) (the flow would also be invalid because the forwarding rule for the red flow at w is not ready yet and hence it cannot reach t anymore).

So the first part of an update schedule should look like this, where the updates in this sequence are performed one-by-one:

$$(u, B), (s, B), (w, R), (s, R), \ldots$$

One valid sequence is the following:

$$(u, B), (s, B), (w, R), (s, R), (u, R), (v, R), (v, B), (w, B)$$

But this schedule requires 8 rounds, updating only one vertex for one flow at a time.

A faster update sequence schedules multiple updates in a single round, if possible without introducing congestion: Updates that are scheduled for the same round are asynchronous and can occur in any order, and hence, need to be performed carefully. The following schedule requires 4 rounds and is the shortest valid congestion-free flow rerouting solution for our example:

$$(u, B), \{(s, B), (v, B), (w, R)\},$$
$$(s, R), \{(u, R), (v, R), (w, B)\}$$

A rigorous formal model for this problem will be given later in this paper.

B. Our Contributions

This paper initiates the study of polynomial-time scheduling algoritms to reroute flows in a congestion-free manner and *fast*. In particular, we contribute the *first* polynomial-time algorithm to compute shortest rerouting schedules *for two flows*, requiring that for each flow the union of the new and the old paths is acyclic. It is already known that for general flow graphs, even for two flows, the problem is NP-hard [1]. In fact, our algorithm runs in (deterministic) *linear time*; its runtime is hence asymptotically optimal.

We show that this is almost as good as one can hope for when investing only polynomial time algorithms: we rigorously prove that even *deciding* whether a congestion-free reroute schedule exists is NP-hard, already for six flows. In other words, we provide an almost tight characterization of the polynomial-time solvability of the problem.

C. Paper Organization

The remainder of this paper is organized as follows. Section II presents a formal model for the problem studied in this paper. Section III describes and analyzes a polynomial-time update scheduling algorithm for two flows and Section IV presents the hardness proof for six flows. We present simulation results in Section V. After reviewing related work in Section VI, we conclude our contribution in Section VII.

II. A Rigorous Formal Model

This section presents a rigorous formal model for the fast congestion-free flow rerouting problem introduced intuitively in Figure 1. The problem can be described in terms of edge capacitated directed graphs. In what follows, we will assume basic familiarity with directed graphs and we refer the reader to [4] for more details. We denote a directed edge e with head v and tail u by $e = (u, v)$. For an undirected edge e between vertices u, v, we write $e = \{u, v\}$; u, v are called endpoints of e.

For ease of presentation and without loss of generality, we consider directed graphs with only one source vertex (where flows will originate) and one terminal vertex (the flows' sink). We call this graph a *flow network*. The forwarding rules that define the paths considered in our problem, are best seen as flows in a network. We will be interested in rerouting flows such that natural notions of consistency are preserved, such as loop-freedom and congestion-freedom. In particular, we will say that a set of flows is valid if the edge capacities of the underlying network are respected.

Definition 1 (Flow Network, Flow, Valid Flow Sets)**.** *A **flow network** is a directed capacitated graph $G = (V, E, s, t, c)$, where s is the* source*, t the* terminal*, V is the set of vertices with $s, t \in V$, $E \subseteq V \times V$ is a set of ordered pairs known as edges, and $c\colon E \to \mathbb{N}$ a capacity function assigning a capacity $c(e)$ to every edge $e \in E$. An (s, t)-*flow *F of demand $d \in \mathbb{N}$ is a* directed path *from s to t in a flow network such that $d \leq c(e)$ for all $e \in E(F)$. Given a set of (s, t)-flows $\mathcal{F} = \{F_1, \ldots, F_k\}$ with demands $d_1, \ldots, d_k$ respectively, we call $\mathcal{F}$ a **valid flow set**, or simply **valid**, if $c(e) \geq \sum_{i\colon e \in E(F_i)} d_i$.*

Recall that we consider the problem of how to reroute a current (old) flow to a new flow, and hence we will consider such flows in "update pairs":

Definition 2 (Update Flow Pair)**.** *An **update flow pair** $P = (F^o, F^u)$ consists of two (s, t)-flows F^o, the* old flow*, and F^u, the* update (or new) flow*, both having demand d. Accordingly, by $\mathcal{P} = \{P_1, \ldots P_k\}$, we denote a family of update flow pairs.*

The update flow network is a flow network (the underlying edge capacitated graph) together with a **valid** family of flow pairs. For an illustration, recall the initial network in Figure 1: The old flows are presented as the directed paths made of solid edges and the new ones are represented by the dashed edges.

A flow can be rerouted by updating the outgoing edges of the vertices along its path (the forwarding rules), i.e., by blocking the outgoing edge of the old flow and by allowing traffic along the outgoing edge of the new flow (if either of them exists). If these two edges coincide, there are no changes. In order to ensure transient consistency, the updates of these outgoing edges need to be scheduled over time: this results in a sequence which can be partitioned into update rounds.

Definition 3 (Resolving Updates, Update Sequence)**.** *Given $G = (V, E, \mathcal{P}, s, t, c)$ and an update flow pair $P = (F^o, F^u) \in \mathcal{P}$ of demand d, we consider the **activation label***

$\alpha_P \colon E(F^o \cup F^u) \times 2^{V \times \mathcal{P}} \to \{\text{active}, \text{inactive}\}$. *For an edge* $(u,v) \in E(F^o \cup F^u)$ *and a set of updates* $U \subseteq V \times \mathcal{P}$, α_P *is defined as follows:*

$$\alpha_P((u,v),U) = \begin{cases} \text{active}, & \text{if } (u,P) \notin U, (u,v) \in E(F^o), \\ \text{active}, & \text{if } (u,P) \in U, (u,v) \in E(F^u), \\ \text{inactive}, & \text{otherwise.} \end{cases}$$

The graph:

$$\alpha(U,G) = (V, \{e \in E \,|\exists i \in [k] \text{ s.t. } \alpha_{P_i}(e,U) = \text{active}\})$$

is called the U***-state*** *of* G *and we call any update in* U ***resolved***.

An ***update sequence*** $\mathfrak{R} = (\mathfrak{r}_1, \ldots, \mathfrak{r}_\ell)$ *is an ordered partition of* $V \times \mathcal{P}$. *For every* $i \in [\ell]$ *we define* $U_i = \bigcup_{j=1}^{i} \mathfrak{r}_j$ *and consider the activation label* $\alpha_P^i(e) = \alpha_P(e, U_i)$ *for every update flow pair* $P = (F^o, F^u) \in \mathcal{P}$ *of demand* d *and edge* $e \in E(F^o \cup F^u)$.

Let (u,P) be some update. When we say that we want to *resolve* (u,P), we mean that we target a state of G in which (u,P) is resolved. In most cases this will mean to add (u,P) to the set of already resolved updates.

With a slight abuse of notation, let define $\alpha_P(U,G) = (V(F^o) \cup V(F^u), (E(F^o) \cup E(F^u))$.

In the definition of an update sequence, $\mathfrak{r}_i$ for $i \in [\ell]$ is a ***round***. We define the *initial round* $\mathfrak{r}_0 = \emptyset$. Recall that we consider unsplittable flows which travel along a single path. The following will clarify how active edges are to be used.

Definition 4 (Transient Flow, Transient Family)**.** *Given a valid family* $\mathcal{P} = \{P_1, \ldots P_k\}$ *of update flow pairs with demands* $d_1, \ldots, d_k$ *respectively, and a set of updates* $U \subseteq V \times \mathcal{P}$. *A flow pair* $P \in \mathcal{P}$ *is* ***transient*** *for* U, *if* $\alpha_P(U,G)$ *contains a unique valid* (s,t)*-flow* $T_{P,U}$. *Similarly,* $\mathcal{P}$ *is a* ***transient family*** *for a set of updates* $U \subseteq V \times \mathcal{P}$, *if and only if every* $P \in \mathcal{P}$ *is transient for* U.

In short, the transient flows look like a path of active edges for flow F, which starts at the source vertex and ends at the terminal vertex. Note that there may be some active edges connected to this path, but they cannot be used to route the flow since $T_{P,U}$ is unique after resolving U. The collection of the transient flows corresponding to the transient family is a snapshot of a valid updating scenario. Whenever we say a path p "routes" a flow F, we mean that all edges of path p are active for flow F.

In each round $\mathfrak{r}_i$, any subset of updates of $\mathfrak{r}_i$ resolved without considering the remaining updates of $\mathfrak{r}_i$ should allow a transient flow for every flow pair. This models the asynchronous nature of the implementation of the update commands in each round.

Definition 5 (Consistency Rule)**.** *Let* $\mathfrak{R} = (\mathfrak{r}_1, \ldots, \mathfrak{r}_\ell)$ *be an update sequence and* $i \in [\ell]$. *We require that for any* $S \subseteq \mathfrak{r}_i, \mathcal{U}_i^S := S \cup \bigcup_{i-1} \mathfrak{r}_j$, *there is a family of transient flow pairs.*

Definition 6 (Valid Update)**.** *An update sequence* $\mathfrak{R}$ *is* ***valid****, or* ***feasible****, if every round* $\mathfrak{r}_i \in \mathfrak{R}$ *obeys the consistency rule.*

Note that we do not forbid any edge $e \in E(F_i^o \cap F_i^u)$ and we never activate or deactivate such an edge. Starting with an initial update flow network, these edges will be active and remain so until all updates are resolved. Hence there are vertices $v \in V$ with either no outgoing edge for a given flow pair F at all; or v has an outgoing edge, but this edge is used by both the old and the update flow of F. We will call such updates (v,P) *empty*.

Empty updates do not have any impact on the actual problem since they never change any transient flow. Hence they can always be scheduled in the first round and thus w.l.o.g. we can ignore them in the following. Let us now define the main problem which we consider in this paper.

Definition 7 (k-NETWORK FLOW UPDATE PROBLEM)**.** *Given an update flow network* G *with* k *update flow pairs, is there a feasible update sequence* $\mathfrak{R}$*? The corresponding optimization problem is: What is the minimum* ℓ *such that there exists a valid update sequence* $\mathfrak{R}$ *using exactly* ℓ *rounds?*

Finally, we introduce some **preliminaries**. Let $G = (V, E, \mathcal{P}, s, t, c)$ be an update flow network consisting of two flow pairs P^1, P^2, such that each flow pair is an acyclic graph.

Let $P_i = (F_i^o, F_i^u)$ be an update flow pair of demand d. Let $v_1, \ldots, v_k$ be the topological order of vertices in the graph $G[F^o \cup F^u]$, denoted by $\prec$. Let $\{z_1, \ldots, z_t\}$ be the intersection of F^o and F^u w.r.t. the above topological order. The subgraph of $F_i^o \cup F_i^u$ induced by the set $\{v \in V(F_i^o \cup F_i^u) \mid z_j \prec v \prec z_{j+1}\}$, $j \in [t-1]$, is called the jth *block* of the update flow pair F_i, or simply the jth i*-block*. We will denote this block by b_j^i.

For a block b, we define $\mathscr{S}(b)$ to be the *start of the block*, i.e., the smallest vertex w.r.t. $\prec$; similarly, $\mathscr{E}(b)$ is the *end of the block*: the largest vertex w.r.t. $\prec$.

Let $G = (V, E, \mathcal{P}, s, t, c)$ be an update flow network with $\mathcal{P} = \{P_1, \ldots, P_k\}$ and let $\mathcal{B}$ be the set of its blocks. Let b be an i-block and P_i the corresponding update flow pair. For a feasible update sequence $\mathfrak{R}$, we will denote the round $\mathfrak{R}(\mathscr{S}(b), P_i)$ by $\mathfrak{R}(b)$. We say that i-block b is *updated*, if all edges in $b \cap F_i^u$ are active and all edges in $b \cap F_i^o \setminus F_i^u$ are inactive.

III. A FAST SCHEDULING ALGORITHM

This section presents an elegant, *linear-time* and deterministic algorithm to compute shortest update schedules for two flows.

Let $G = (V, E, \mathcal{P}, s, t, c)$ be an update flow network where (V, E) is the union of the DAGs implied by the flow pairs. Let $\mathcal{P} = \{B, R\}$ be the two update flow pairs with $B = (B^o, B^u)$ and $R = (R^o, R^u)$ of demands d_B and d_R. As in the previous section, we identify B with blue and R with red.

We say that an I-block b_1 is *dependent* on a J-block b_2, $I, J \in \{B, R\}$, $I \neq J$, if there is an edge $e \in (E(b_1) \cap E(I^u)) \cap (E(b_2) \cap E(J^o))$, but $c(e) < d_I + d_J$. In fact, to update b_1, we either violate capacity constraints, or we update b_2 first in order to prevent congestion. In this case, we write $b_1 \to b_2$ and say that b_1 *requires* b_2.

We say a block b is a *free block*, if it is not dependent on any other block. A *dependency graph* of G is a graph $D = (V_D, E_D)$ for which there exists a bijective mapping $\mu\colon V(D) \leftrightarrow B(G)$, and there is an edge $(v_b, v_{b'})$ in D if $b \to b'$. Clearly, a block b is free if and only if it corresponds to a sink in D.

We propose the following algorithm to check the feasibility of the flow rerouting problem.

Algorithm 1. **Feasible 2-Flow DAG Update**
Input: Update Flow Network G

1) Compute the dependency graph D of G.
2) If there is a cycle in D, return *impossible to update*.
3) While $D \neq \emptyset$ repeat:
   i Update all blocks which correspond to the sink vertices of D, in the number of rounds from Corollary 1.
   ii Delete all of the current sink vertices from D.

Recall that empty updates can always be scheduled in the first round, even for infeasible problem instances. So for Algorithm 1 and all following algorithms, we simply assume these updates to be scheduled together with the non-empty updates of round 1.

Figure 2 gives an example of an update flow network on a DAG and illustrates the block decomposition and its value to finding a feasible update sequence.

Suppose $\mathfrak{R}$ is a feasible update sequence for G. We say that a c-block b w.r.t. $\mathfrak{R} = (\mathfrak{r}_1, \ldots, \mathfrak{r}_\ell)$ is *updated in consecutive rounds*, if the following holds: if some of the edges of b are activated/deactivated in round i and some others in round j, then for every $i < k < j$, there is an edge of b which is activated/deactivated.

We reiterate the following two lemmas from [1] (Lemmata 4 and 5).

Lemma 1. *Let b be a c-block. Then in a feasible update sequence $\mathfrak{R}$, all vertices (resp. their outgoing c flow edges) in $F_c^u \cap b - \mathscr{S}(b)$ are updated strictly before $\mathscr{S}(b)$. Moreover, all vertices in $b - F_c^u$ are updated strictly after $\mathscr{S}(b)$ is updated.*

Lemma 2. *Given any feasible (not necessarily shortest) update sequence $\mathfrak{R}$, there is a feasible update sequence $\mathfrak{R}'$ which updates every block in at most 3 consecutive rounds.*

From the above lemmas, we immediately derive a corollary regarding the optimality in terms of the number of rounds: the 3 rounds feasible update sequence.

Corollary 1. *Let b be any c-block with $|E(b \cap F_c^o)| \geq 2$ and $|E(b \cap F_c^u)| \geq 2$. Then it is not possible to update b in less than 3 rounds: otherwise it is not possible to update b in less than 2 rounds.*

Next we show that if there is a cycle in the dependency graph, then it is impossible to update any flow.

Lemma 3. *If there is a cycle in the dependency graph, then there is no feasible update sequence.*

Proof. Suppose that there exists a cycle in the dependency graph. Without loss of generality, we can assume that this is the only cycle in the dependency graph as we can always remove vertices without creating new dependencies. Then it is not possible to update the cycle. For the sake of contradiction, suppose that there is a feasible update order; then there is a feasible update order in which blocks are updated in consecutive (distinct) rounds. But in this order, one of the vertices in the dependency graph (a block) should be earlier than the others. This is impossible due to dependency on other vertices. □

We will now slightly modify Algorithm 1 to create a new algorithm which not only computes a feasible sequence $\mathfrak{R}$ for a given update flow network in polynomial time, whenever it exists, but which also ensures that $\mathfrak{R}$ is as short as possible (in terms of number of rounds). For any block b, let $c(b)$ denote its corresponding flow pair.

Algorithm 2. **Optimal 2-Flow DAG Update**
Input: Update Flow Network G

1) Compute the dependency graph D of G.
2) If there is a cycle in D, return *impossible to update*.
3) If there is any block b corresponding to a sink vertex of D with $(b \cap F^u_{c(b)}) - \mathscr{S}(b) \neq \emptyset$ set $i := 2$, otherwise set $i := 1$.
4) While $D \neq \emptyset$ repeat:
   i Schedule the update of all blocks b which correspond to the sink vertices of D for the rounds $i-1$, i, $i+1$, such that $\mathscr{S}(b)$ is updated in round i.
   ii Delete all of the current sink vertices from D.
   iii Set $i := i + 1$.

Theorem 1. *An optimal (feasible) update sequence on acyclic update flow networks with exactly 2 update flow pairs can be found in linear time.*

Proof. Let G denote the given update flow network. In the following, for ease of presentation, we will slightly abuse terminology and say that "a block is updated in some round", meaning that the block is updated in the corresponding consecutive rounds as in the proof of Lemma 2.

We proceed as follows. First, we find a block decomposition and create the dependency graph of the input instance. This takes linear time only. If there is a cycle in that graph, we output *impossible* (cf Lemma 3). Otherwise, we apply Algorithm 2. As there is no cycle in the dependency graph (a property that stays invariant), in each round, either there exists a free block which is not updated yet, or everything is already updated or is in the process of being updated. Hence,

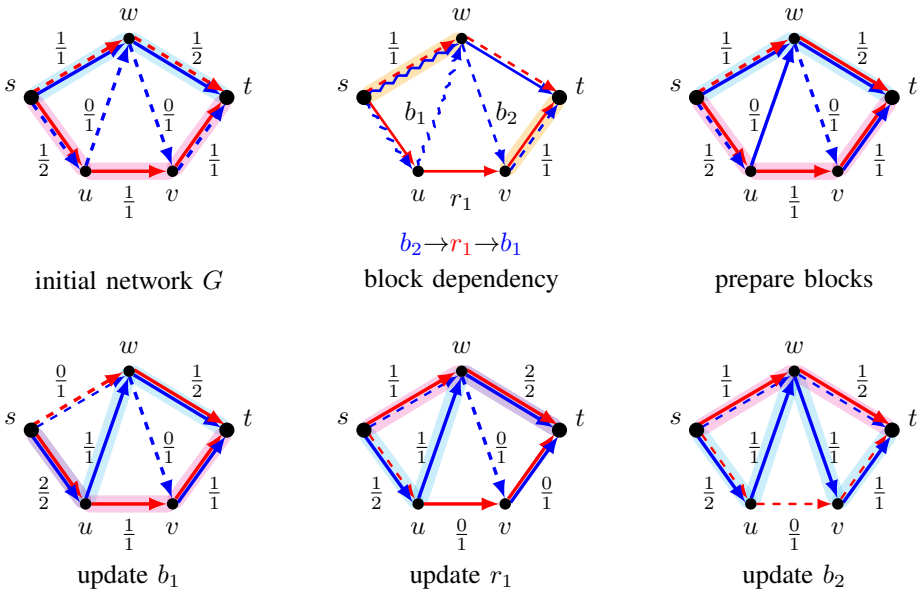

Fig. 2: *Example* for Algorithm 1. The 2 update flow pairs are red and blue, each of demand 1. The active edges of the respective colors are indicated as *solid lines* and the inactive edges are *dashed*. Each edge in the flow graph is annotated with its current load (*top*) and its capacity (*bottom*). We start by identifying the blue and red blocks. For red there is exactly one such block r_1, since R^o and R^u only coincide in s and t. The blue flow pair on the other hand omits two blocks b_1 and b_2: B^o and B^u meet again at w and at t. We observe that b_2 can only be updated after r_1 has been updated; similarly, r_1 can only be updated after b_1 has been updated. An update sequence respecting these dependencies can be constructed as follows. We can first prepare the blocks by updating the following two out-edges which currently do not carry any flow: (w, red), $(u, blue)$, and $(v, blue)$. Subsequently, the three blocks can be updated in a congestion-free manner in the following order: Prepare the update for all blocks in the first round. Then, update b_1 in the second round, r_1 in the third round, b_2 in the fourth round.

if there is a feasible solution (it may not be unique), we can find one in time $O(|G|)$.

For the optimality in terms of the number of rounds, consider two feasible update sequences. Let $\mathfrak{R}_{\text{ALG}}$ be the update sequence produced by Algorithm 2 and let $\mathfrak{R}_{\text{OPT}}$ be a feasible update sequence that realizes the minimum number of rounds. According to Lemma 1, any block b is updated only in round $\mathscr{S}(b)$.

Suppose there is a block b' such that $\mathfrak{r}_{\text{OPT}}(b') < \mathfrak{r}_{\text{ALG}}(b')$. Then let b be the block with earliest update round $\mathfrak{r}_{\text{ALG}}(b) > \mathfrak{r}_{\text{OPT}}(b)$. That is, for every block b'' with $\mathfrak{r}_{\text{OPT}}(b'') \leq \mathfrak{r}_{\text{OPT}}(b)$, $\mathfrak{r}_{\text{OPT}}(b'') \geq \mathfrak{r}_{\text{ALG}}(b'')$ holds. Since $\mathscr{S}(b)$ is updated in round $\mathfrak{r}_{\text{OPT}}(b)$, there are no dependencies for b that are still in place in this round. Thus, according to the sequence $\mathfrak{R}_{\text{OPT}}$, b is a sink vertex of the dependency graph after round $\mathfrak{r}_{\text{OPT}}(b) - 1$. Furthermore, by our previous observation, every start of some block has been updated up to this round in the optimal sequence, and hence it is also already updated in the same round in $\mathfrak{R}_{\text{ALG}}$. This means that after round $\mathfrak{r}_{\text{OPT}}(b) - 1 < \mathfrak{r}_{\text{ALG}}(b) - 1$, b is a sink vertex of the dependency graph of $\mathfrak{R}_{\text{ALG}}$ as well. Thus, Algorithm 2 would have scheduled the update of block b in the rounds $\mathfrak{r}_{\text{OPT}}(b) - 1$, $\mathfrak{r}_{\text{OPT}}(b)$ and $\mathfrak{r}_{\text{OPT}}(b) + 1$. Contradiction.

Thus $\mathfrak{r}_{\text{ALG}}(b) \leq \mathfrak{r}_{\text{OPT}}(b)$ for all blocks b. Now let $b_1, \ldots, b_\ell$ be the last blocks whose starts are updated the latest under $\mathfrak{R}_{\text{ALG}}$. If there is some $i \in [\ell]$ such that $\left|E^o_{b_i}\right| \geq 2$ and $\left|E^u_{b_i}\right| \geq 2$, $\mathfrak{R}_{\text{ALG}}$ uses exactly $\mathfrak{r}_{\text{ALG}}(b_i) + 1$ rounds; otherwise it is one round less, by Corollary 1. By our previous observation, none of these blocks can start later than $\mathfrak{r}_{\text{ALG}}(b_i)$ and thus $\mathfrak{r}_{\text{OPT}}$ uses at least as many rounds as Algorithm 2. Hence the algorithm is optimal in the number of rounds. □

IV. NP-hardness for More Flows

This section shows that the polynomial-time result derived above cannot be generalized much further: it is NP-hard to compute a shortest schedule already for six flows, and even if the pair of old and new path *forms a DAG*. In fact, we show that already the decision problem, i.e., whether a feasible schedule exists, is NP-hard.

Theorem 2. *Deciding whether a feasible network update schedule exists for a given update flow network in which each flow pair forms a DAG is NP-hard for six flows.*

We use a reduction from 3-SAT. Let C be any 3-SAT formula with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. The resulting update flow network is denoted as $G(C)$.

We will create 6 flow pairs: X, $\overline{X}$, D_1, D_2, D_3 and B, each having demand 1. B is the blocking pair: it can by updated only if all clauses are satisfied. Flows X and $\overline{X}$ contain gadgets for all literals, X for positive ones and $\overline{X}$ for negative ones. Updating a variable gadget in X corresponds to assigning the variable value 1 in C. Flow B prevents the

variable gadget to be updated in both X and $\overline{X}$, unless all clauses are satisfied.

Flows D_1, D_2 and D_3 encode clauses of C. Each of these flows contains a clause gadget linking a clause to one of its literals. This gadget can be updated only if the literal is satisfied. Updating a clause gadget in one of those flows will allow B to be updated.

Now we proceed with the detailed description of the reduction.

1) **Clause gadgets:** For every clause $i \in [m]$ we introduce eight vertices: u^i, v^i and for $j \in \{1,2,3\}$ u^i_j and v^i_j. For $j \in \{1,2,3\}$ we add edge (u^i, v^i) to D^o_j and edges (u^i, u^i_j), (u^i_j, v^i_j) and (v^i_j, v^i) to D^u_j.
2) **Variable Gadgets:** For every $j \in [n]$, we introduce two vertices: w^j_1 and w^j_2. Let $P_j = \left\{p^j_1, \ldots, p^j_{k_j}\right\}$ denote the set of indices of the clauses containing the literal x_j and $\overline{P}_j = \left\{\overline{p}^j_1, \ldots, \overline{p}^j_{k'_j}\right\}$ the set of indices of the clauses containing the literal $\overline{x}_j$. Furthermore, let $\pi(i,j)$ denote the position of x_j in the clause C_i, $i \in P_j$. Similarly, $\overline{\pi}(i',j)$ denotes the position of $\overline{x_j}$ in $C_{i'}$ where $i' \in \overline{P}_j$.
   To X^u and $\overline{X}^u$ we add edge (w^j_1, w^j_2).
   To X^o we add the following edges:
   - $(u^{p^j_i}_{\pi(p^j_i,j)}, v^{p^j_i}_{\pi(p^j_i,j)})$ for all $i \in \{1, \ldots, k_j\}$,
   - $(v^{p^j_i}_{\pi(p^j_i,j)}, u^{p^j_{i+1}}_{\pi(p^j_{i+1},j)})$ for all $i \in \{1, \ldots, k_j - 1\}$,
   - $(w^j_1, u^{p^j_1}_{\pi(p^j_1,j)})$ and $(v^{p^j_k}_{\pi(p^j_k,j)}, w^j_2)$.

   We proceed similarly with $\overline{X}^o$ and clauses containing $\overline{x}_j$.
3) **Blocking flow:** The goal of flow B is to block update of w^j_1, for any $j \in [n]$, in both X and $\overline{X}$.
   To do that we add to B^o the following edges:
   - (w^j_1, w^j_2), for all $j \in [n]$,
   - (w^j_2, w^{j+1}_1), for all $j \in [n-1]$.

   We also add the following edges to B^u:
   - (u^i, v^i), for all $i \in [m]$,
   - (v^i, u^{i+1}), for all $i \in [m-1]$.
4) **Source and Terminal:** Now we need to connect all the gadgets in the flows. The source and the terminal of all flows will be s and t.
   To D^o_j and D^u_j, for $j \in \{1,2,3\}$, we add the following edges:
   - (v^i, u^{i+1}), for all $i \in [m-1]$,
   - (s, u^1) and (v^m, t).

   To X^o, X^u, $\overline{X}^o$ and $\overline{X}^u$ we add the following edges:
   - (w^j_2, w^{j+1}_1), for all $j \in [n-1]$
   - (s, w^1_1) and (w^n_2, t)

   We also add edges (s, w^1_1) and (w^n_2, t) to B^o and edges (s, u^1) and (v^m, t) to B^u.
5) **Edges capacity:** For all $j \in [n]$ we set the capacity of edge (w^j_1, w^j_2) to be 2. Also for all $i \in [m]$ we set capacity of edge (u^i, v^i) to be 3 and capacity of edge (u^i_j, v^i_j), for $j \in \{1,2,3\}$, to be 1.
   For all the other edges, we set their capacity to be 6, that is, to the number of flows. Therefore they cannot violate any capacity constraint.

Lemma 4. *Given any valid update sequence $\mathfrak{R}$ for the above constructed update flow network $G(C)$, the following conditions hold.*

1) *For every $r < \mathfrak{R}(s,B)$ and $j \in [n]$ $\mathfrak{R}(w^j_1, X) > r$ or $\mathfrak{R}(w^j_1, \overline{X}) > r$.*
2) *For every $r \geq \mathfrak{R}(s,B)$ and $i \in [m]$ $\mathfrak{R}(u^i, D_1) < r$, $\mathfrak{R}(u^i, D_2) < r$ or $\mathfrak{R}(u^i, D_3) < r$.*

Proof. Note that B^o and B^u have no common nodes apart from s and t. That means that for any r either $T_{B,U_r} = B^o$ or $T_{B,U_r} = B^u$. Now we prove both conditions.

1) Let us consider any $j \in [n]$. As $r < \mathfrak{R}(s,B)$, then $T_{B,U_r} = B^o$. The capacity of edge (w^j_1, w^j_2) is 2 and it belongs to B^o. Therefore it can be in at most one other transient flow, so the condition holds.
2) Let us consider any $i \in [m]$. As $r \geq \mathfrak{R}(s,B)$, then $T_{B,U_r} = B^u$. The capacity of edge (u^i, v^i) is 3 and it belongs to B^u. Therefore it can be in at most two other transient flows, so the condition holds.

□

Proof. Now we are ready to prove Theorem 2. First, let us assume that C is satisfiable and we will construct valid update sequence for $G(C)$. Let σ be an assignment satisfying C. Then the update sequence for $G(C)$ is as follows.

1) For every $j \in [n]$, if $\sigma(x_j) = 1$ then resolve (w^j_1, X), otherwise resolve $(w^j_1, \overline{X})$.
2) For every clause C_i at least one of the edges (u^i_1, v^i_1), (u^i_2, v^i_2) and (u^i_3, v^i_3) is neither in $T_{X, r^f_2 - 1}$ nor $T_{\overline{X}, r^f_2 - 1}$. So the update of u^i can be resolved in the corresponding flow D_1, D_2 or D_3 (this follows from σ being satisfying assignment).
3) As every $i \in [m]$ edge (u^i, v^i) is used by at most 2 flows, we can resolve every update in B^u, resolving (s,B) as the last one.
4) For every $j \in [n]$, resolve either (w^j_1, X) or $(w^j_1, \overline{X})$, depending on which one was not resolved in step 1.
5) For every $i \in [m]$ resolve updates of u^i in flows D_1, D_2 and D_3 (those that have not been resolved in step 2).
6) Resolve the remaining updates in all flows.

Now let us assume that there is a valid update sequence σ for $G(C)$. We will show that C is satisfiable by constructing satisfying assignment σ.

Let us consider round $r = \sigma(s,B)$. We assign values in the following way. For $j \in [n]$, if $\sigma(w_1, X) < r$ then $\sigma(x_j) := 1$ and if $\sigma(w_1, \overline{X}) < r$ then $\sigma(x_j) := 0$. If both $\sigma(w_1, X) > r$ and $\sigma(w_1, \overline{X}) > r$ we assign to x_j an arbitrary value. By Condition 1 of Lemma 4 this is a correct assignment, that is no variable is assigned two values.

We want to prove that this assignment satisfies σ. Let us consider any clause C_i. By Condition 2 of Lemma 4 at least one of (u^i, D_1), (u_i, D_2) or (u_i, D_3) is updated before round

r. That means that at least for one of variables x_j in C_i $\sigma(w_1^j, X) < r$, if C_i contains literal x_j, or $\sigma(w_1^j, \overline{X}) < r$, if C_i contains literal $\overline{x}_j$. This means that C_i is satisfied by x_j in σ.

□

V. Empirical Results

In order to gain insights into the actual number of rounds needed to reroute flows in real networks, we conducted a simulation study on real network topologies. We also want to compare the length of the schedules produced by our algorithm (which provably provides *shortest* schedules) to the state-of-the-art algorithm presented in [1] (which only computes *feasible* schedules).[1] In order to study the need for fast algorithms, we compare the runtime of our algorithm to the mathematical programming approach, as it is frequently used in the literature [14].

We implemented Algorithm 2 using standard C++ libraries and performed an exhaustive evaluation on over 100 topologies provided by the Topology Zoo project [22]. We produced ≈ 136 million random update problems in total. In a preprocessing step, the evaluation takes the raw graph and allocates minimal capacities: set the capacity to 2 for links that carry the old/update flow paths of both pairs, otherwise set the capacity to 1.

The program, for every pair of source and destination (s, t), first computes all the paths from s to t. Next, it iterates over all possible path pairs (i.e. old and update paths) chosen independently for each of the two flows (dismissing repeated path pairs). Each iteration does the following.

1) Performs Line 1 on the path pairs and generates a block dependency graph D.
2) Enumerates all paths in D and each path P is weighted as follows.
   a) Initialize $w(P) = |P|$.
   b) Let b_1 be the block that corresponds to the last vertex of P. Set $w(P) = w(P) + 1$ if $|E[b_1 \cap F^u(b_1)]| > 1$.
   c) Let b_2 be the block that corresponds to the first vertex of P. Set $w(P) = w(P) + 1$ if $|E[b_1 \cap F^o(b_1)]| > 1$.
3) Find the path $P_{max} = \max_{P'} w(P')$ (ties broken arbitrarily). Let $\ell = w(P_{max})$.
4) For each block b corresponding to a vertex in D apply $\ell = \max(\ell, |E[b \cap F^o(b)]| + |E[b \cap F^u(b)]| + 1)$.

At the end, ℓ will hold the actual number of rounds it takes in the optimal schedule produced by Algorithm 2. The case 2b accounts for the preparation (i.e. adding new flow rules) round of the block scheduled earliest in a chain of dependent blocks (i.e. current path P). Similarly, 2c accounts for the cleanup round (i.e. removal of old flow rules) of the block scheduled the latest in that chain.

[1]More specifically, we observe that our Algorithm 1 is a simpler feasibility algorithm than [1], *for two flows*: it employs basic batching, which leads to shorter schedules compared to [1]. We hence use Algorithm 1 as a baseline and lower bound on the number of rounds needed by the more complex algorithm in [1].

Eventually, ℓ is determined either by the chain of dependent blocks that corresponds to the longest weighted path in D, or by some single block at Line 4 due to extra preparation/cleanup rounds consumed by that block.

$$\textbf{Minimize } R \tag{1}$$

$$\mathit{ROUNDS} = \{1, .., (|V| - 1).|P|\}$$

$$\sum_{r \in \mathit{ROUNDS}} x^r_{v,i} = 1 \qquad \forall i \in |P|, v \in P_i \tag{2}$$

$$y^0_{(u,v),i} = 1 \qquad \forall (u,v) \in F^o_i \tag{3}$$

$$y^0_{(u,v),i} = 0 \qquad \forall (u,v) \in F^u_i \tag{4}$$

$$\forall i \in [|P|], r \in \mathit{ROUNDS}\ \{ \tag{5}$$

$$x^r_{v,i}, \mathit{fork}^r_{v,i}, \mathit{join}^r_{v,i} \in \{0,1\} \qquad \forall v \in P_i \tag{6}$$

$$y^r_{(u,v),i}, f^r_{(u,v),i} \in [0,1] \qquad \forall (u,v) \in P_i \tag{7}$$

$$R \geq r.x^r_{v,i} \qquad \forall v \in P_i \tag{8}$$

$$y^r_{(u,v),i} = \sum_{r' \leq r} x^{r'}_{u,i} \qquad \forall (u,v) \in F^u_i \tag{9}$$

$$y^r_{(u,v),i} = 1 - \sum_{r' \leq r} x^{r'}_{u,i} \qquad \forall (u,v) \in F^o_i \tag{10}$$

$$\mathit{fork}^r_{v,i} = \begin{cases} x^r_{v,i} \ \exists w, w' \in P_i : \begin{cases} (v,w) \in F^o_i \\ (v,w') \in F^u_i \end{cases} & \forall v \in P_i \\ 0 \quad \text{else} \end{cases} \tag{11}$$

$$f^r_{(u,v),i} \leq y^{r-1}_{(u,v),i} + \mathit{fork}^r_{u,i} \qquad \forall (u,v) \in P_i \tag{12}$$

$$f^r_{(u,v),i} \leq y^r_{(u,v),i} + \mathit{fork}^r_{u,i} \qquad \forall (u,v) \in P_i \tag{13}$$

$$\mathit{join}^r_{v,i} \leq f^r_{(u,v),i}, f^r_{(u',v),i} \forall v, u, u' \in P_i : \begin{cases} (u,v) \in F^o_i \\ (u',v) \in F^u_i \end{cases} \tag{14}$$

$$\sum_{(v,w) \in P_i} f^r_{(v,w),i} - \sum_{(u,v) \in P_i} f^r_{(u,v),i} = \begin{cases} 1 + \mathit{fork}^r_{s,i} & v = s \\ -(1 + \mathit{join}^r_{t,i}) & v = t \\ \mathit{fork}^r_{v,i} - \mathit{join}^r_{v,i} & \text{else} \end{cases} \qquad \forall v \in P_i \tag{15}$$

$$\}$$

$$\sum_{i \in [|P|]} f^r_{(u,v),i} \leq C_{(u,v)} \qquad \forall r \in \mathit{ROUNDS}, (u,v) \in E \tag{16}$$

Fig. 3: Mixed Integer Program for k flow pairs

Our results (see Figure 4a) show that the optimal number of rounds on the networks vary between 2 and 6. When evaluating Algorithm 1 on the same input data, we see that the number of rounds is higher (especially the tail), and spread between 2 and 14 (see Figure 4b).

We also implemented a mixed integer program of this problem for comparison (Figure 3), as this is the usual approach in the literature [14]. The runtime, even for two flows, is usually a few seconds, much longer compared to the results from the first implementation (≈ 100 microseconds, see Figure 4c).

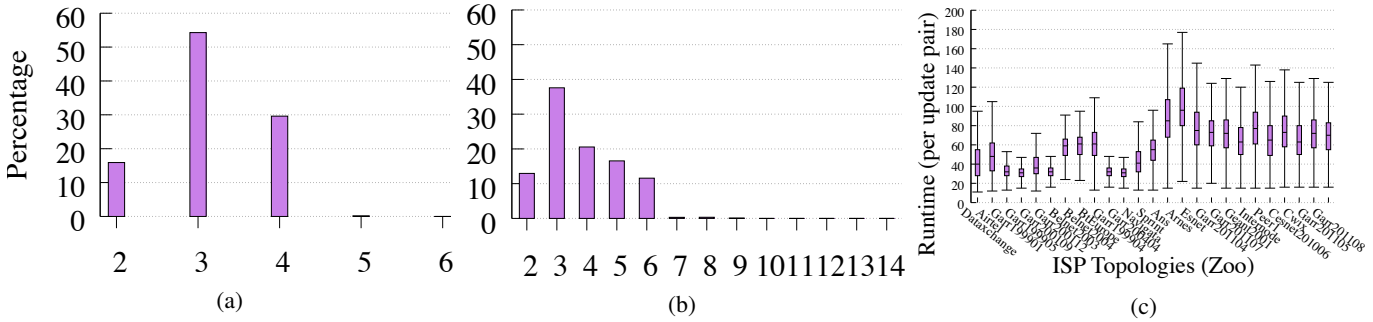

Fig. 4: (a) The frequency of each possible number of rounds (in %) in optimal schedules from Algorithm 2, (b) in arbitrary feasible schedules from Algorithm 1. (c) Distribution of runtime (in microsecond) over all the problem instances from some of the evaluated graphs.

Next, we describe the formulation in detail. (2): each schedule variable $x^r_{v,i}$ indicates whether a node v is scheduled to update flow i in round r. (5): repeat the embraced lines for every pairs P_i and each round $r \in$ *ROUNDS*. (9),(10): $y^r_{(u,v),i}$ indicates whether the link (u, v) is active for pair i immediately after round r (i.e. active graph). (11): *fork nodes* are the nodes at which old and update paths split. A fork node v acts as a source, doubling its incoming transient flow i, when it updates the flow during round r (i.e. if $fork^r_{v,i}$ is 1). (14): *join nodes* are nodes at which the old and update paths meet once again. A join node acts as a sink (if $join^r_{v,i}$ is 1) when the two in-links both carry the transient flow i in the transient state of round r. (12),(13): $f^r_{u,v,i}$ specifies the transient flow i on a link (u, v). The first terms on the r.h.s. constraints together state that the link is allowed to be utilized in the transient state of round r, if it is active before round r and it remains active during the round. Alternatively, if the link is deactivating in round r due to the updating fork node u, then the second term allows the link to be usable in the transient state. (This, along with Constraint (15) guarantees there will be no loops on the old out-branch of any updating fork node.) (15): runs a variable-size transient flow i from s to t in order to impose st-connectivity in the worst-case transient state. The flow produced at s is of size 1 and it arrives at t with the same size. In the meanwhile, any active fork node (including possibly s) adds one unit to this flow and splits it into two unit-size flows along both its out-links. Later, a join node consumes this extra flow by taking away the 1 unit. (16): the capacity constraints.

Because of a possible cleanup round after a fork node updates, it is necessary to maintain st-connectivity via both (old and update) out-links of the fork node, which is ensured by (15). In other words, no cleanup (i.e. removal of old flow rules) should occur on the old branch of the fork node in the same round it reroutes to the new branch.

VI. Related Work

The fundamental problem of how to reroute flows has recently received much attention in the networking community and we refer the reader to the recent survey by Foerster et al. [14] for an overview of the field. Yet, today, and in contrast to the classic problem of how to *route* flows [2, 9, 20, 21, 24, 37], we still know surprisingly little about useful *algorithmic* techniques for efficient flow *rerouting*.

There exist several empirical studies motivating our model [19, 23], however, this literature is orthogonal to ours. Moreover, many existing consistent network update algorithms such as [7, 19, 25, 30, 32, 35] require packet tagging and additional forwarding rules, which render the problem different in nature. Mahajan and Wattenhofer [32] initiated the study of flow rerouting algorithms which schedule updates over time. The authors also presented first algorithms to quickly updates routes in a transiently *loop-free* manner [3, 13, 16], by maximizing *the number of updates per round*. A second line of research focuses on *minimizing the number of rounds* of loop-free updates [8, 12, 26–28].

As congestion is known to negatively affect application performance and user experience, it has also been studied intensively in the context of flow rerouting problems. The seminal work by Hongqiang et al. [25] on congestion-free rerouting has already been extended in several papers, using static [5, 6, 10, 17, 29, 31, 36, 41], dynamic [19], or time-based [11, 15, 33, 34, 39, 40] approaches. Vissicchio et al. presented FLIP [38], which combines per-packet consistent updates with order-based rule replacements, in order to reduce memory overhead: additional rules are used only when necessary. Moreover, Hua et al. [18] recently initiated the study of adversarial settings, and presented FOUM, a flow-ordered update mechanism that is robust to packet-tampering and packet dropping attacks. However, none of these papers present polynomial-time algorithms for rerouting flows without requiring packet tagging.

Our work on polynomial-time algorithms is motivated in particular by the negative result by Ludwig et al. [27] who showed that deciding whether a loop-free 3-round update schedule exists is NP-hard, even in the absence of capacity constraints. Given this negative result, much prior work typically resorts to heuristics [39], which however do not come with any formal guarantees on the quality of the computed schedule, or to algorithms which have a super-polynomial

runtime [26]. The only exception is the polynomial-time algorithm by Amiri et al. [1] for acyclic flow graphs, which however is limited to computing *feasible* (possibly very long) update schedules. There are various differences between this paper and the recent work of Amiri et al.: (1) In contrast to our work where only flow pairs need to form a DAG, [1] considers a much more restricted model where the union of all flows must be acyclic. This restriction allows the authors to design an FPT algorithm for k flows, whereas in our model the problem is NP-complete already for six flows. Very different techniques are required to show hardness. (2) Similarly to [1], our algorithm relies on a dependency graph that explains the relation between flows. However, since we aim to compute schedules only for two flows, we do not require the big machinery introduced for k flows but can provide a more elegant algorithm. (3) At the same time, since in contrast to prior work, we focus on an optimal solution, our model is more chalelnging and requires new algorithmic ideas.

VII. Future Work

The main open question of our work concerns the polynomial-time tractability for $2 < k < 6$ flows. These cases might be very challenging, and currently, we do not have any insights on how to deal even with three flows. Another direction for future work regards the study of relevant more specific networks such as sparse networks.

References

[1] Saeed Akhoondian Amiri, Szymon Dudycz, Stefan Schmid, and Sebastian Wiederrecht. Congestion-free rerouting of flows on dags. In *Proc. ICALP*, 2018.

[2] Saeed Akhoondian Amiri, Stephan Kreutzer, Dániel Marx, and Roman Rabinovich. Routing with congestion in acyclic digraphs. In *MFCS*, pages 7:1–7:11, 2016.

[3] Saeed Akhoondian Amiri, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Transiently consistent sdn updates: Being greedy is hard. In *SIROCCO*, 2016.

[4] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs - theory, algorithms and applications*. Springer, 2002.

[5] Sebastian Brandt, Klaus-Tycho Foerster, and Roger Wattenhofer. Augmenting flows for the consistent migration of multi-commodity single-destination flows in sdns. *Pervasive Mob. Comput.*, 36:134–150, 2017.

[6] Sebastian Brandt, Klaus-Tycho Förster, and Roger Wattenhofer. On Consistent Migration of Flows in SDNs. In *Proc. IEEE INFOCOM*, 2016.

[7] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. IEEE INFOCOM*, 2015.

[8] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. In *Proc. IEEE IFIP DSN*, 2016.

[9] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5(4):691–703, 1976.

[10] Klaus-Tycho Foerster. On the consistent migration of unsplittable flows: Upper and lower complexity bounds. In *IEEE NCA*, 2017.

[11] Klaus-Tycho Foerster. On the consistent migration of splittable flows: Latency-awareness and complexities. In *IEEE NCA*, 2018.

[12] Klaus-Tycho Foerster, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Loop-free route updates for software-defined networks. *IEEE/ACM Trans. Netw.*, 26(1):328–341, 2018.

[13] Klaus-Tycho Foerster, Thomas Luedi, Jochen Seidel, and Roger Wattenhofer. Local checkability, no strings attached: (a)cyclicity, reachability, loop free updates in sdns. *Theor. Comput. Sci.*, 709:48–63, 2018.

[14] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. Survey of consistent software-defined network updates. *IEEE Commun. Surveys Tuts.*, 2018.

[15] Klaus-Tycho Foerster, Laurent Vanbever, and Roger Wattenhofer. Latency and consistent flow migration: Relax for lossless updates. In *Proc. IFIP Networking*, 2019.

[16] Klaus-Tycho Förster, Ratul Mahajan, and Roger Wattenhofer. Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes. In *Proc. 15th IFIP Networking*, 2016.

[17] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26, 2013.

[18] Jingyu Hua, Xin Ge, and Sheng Zhong. FOUM: A Flow-Ordered Consistent Update Mechanism for Software-Defined Networking in Adversarial Settings. 2016.

[19] Xin Jin, Hongqiang Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Jennifer Rexford, Roger Wattenhofer, and Ming Zhang. Dynamic scheduling of network updates. In *Proc. ACM SIGCOMM*, 2014.

[20] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Stephan Kreutzer. An excluded half-integral grid theorem for digraphs and the directed disjoint paths problem. In *STOC*, pages 70–78, 2014.

[21] Jon M. Kleinberg. Decision algorithms for unsplittable flow and the half-disjoint paths problem. In *Proc. STOC*, pages 530–539, 1998.

[22] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[23] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about sdn flow tables. In *Proc. PAM*, pages 347–359. Springer, 2015.

[24] Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.

[25] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David A. Maltz. Update: Updating Data Center Networks with Zero Loss. In *Proc. ACM SIGCOMM*, 2013.

[26] Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. Transiently secure network updates. In *Proc. ACM SIGMETRICS*, 2016.

[27] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM PODC*, 2015.

[28] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proc. HotNets*, 2014.

[29] Long Luo, Hongfang Yu, Shouxi Luo, and Mingui Zhang. Fast lossless traffic migration for SDN updates. In *ICC*, 2015.

[30] Long Luo, Hongfang Yu, Shouxi Luo, Mingui Zhang, and Shui Yu. Achieving fast and lightweight sdn updates with segment routing. In *2016 IEEE Global Communications Conference (GLOBECOM)*, 2016.

[31] Shouxi Luo, Hongfang Yu, Long Luo, and Lemin Li. Arrange your network updates as you wish. In *IFIP Networking*, 2016.

[32] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proc. ACM HotNets*, 2013.

[33] Tal Mizrahi and Yoram Moses. Time-based updates in software defined networks. In *Proc. ACM HotSDN*, pages 163–164, 2013.

[34] Tal Mizrahi, Ori Rottenstreich, and Yoram Moses. Timeflip: Scheduling network updates with timestamp-based tcam ranges. In *Proc. IEEE INFOCOM*, pages 2551–2559, 2015.

[35] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proc. ACM SIGCOMM*, pages 323–334, 2012.

[36] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. ACM HotNets*, 2011.

[37] Martin Skutella. Approximating the single source unsplittable min-cost flow problem. In *Proc. IEEE FOCS*, pages 136–145, 2000.

[38] Stefano Vissicchio and Luca Cittadini. FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates. In *Proc. IEEE INFOCOM*, 2016.

[39] Jiaqi Zheng, Guihai Chen, Stefan Schmid, Haipeng Dai, Jie Wu, and Qiang Ni. Scheduling congestion- and loop-free network update in timed sdns. *IEEE J. Sel. Areas Commun.*, 35(11):2542–2552, 2017.

[40] Jiaqi Zheng, Bo Li, Chen Tian, Klaus-Tycho Foerster, Stefan Schmid, Guihai Chen, and Jie Wu. Congestion-free rerouting of multiple flows in timed sdns. *IEEE Journal on Selected Areas in Communications*, 2019.

[41] Jiaqi Zheng, Hong Xu, Guihai Chen, and Haipeng Dai. Minimizing transient congestion during network update in data centers. In *Proc. ICNP*, 2015.