Loop-Free Route Updates for Software-Defined Networks

Klaus-Tycho Foerster Arne Ludwig Jan Marcinkowski Stefan Schmid

***Abstract*—We consider the fundamental problem of updating arbitrary routes in a software-defined network in a (transiently) loop-free manner. Our objective is to compute fast network update schedules which minimize the number of interactions (i.e., rounds) between the controller and the network nodes. We first prove that this problem is difficult in general: The problem of deciding whether a k-round update schedule exists is NP-complete already for $k = 3$, and there are problem instances requiring $\Omega(n)$ rounds, where n is the network size. Given these negative results, we introduce an attractive, relaxed notion of loop-freedom. We show that relaxed loop-freedom admits for much shorter update schedules (up to a factor $\Omega(n)$ in the best case), and present a scheduling algorithm which requires at most $\Theta(\log n)$ rounds.**

***Index Terms*—Software-Defined Networking; Graph Algorithms; Scheduling; NP-hardness**

I. Introduction

Software-Defined Networks (SDNs) introduce interesting new flexibilities in terms of traffic-engineering and programmatic network control, by outsourcing and consolidating the control over a set of nodes (switches or routers) to a logically centralized (but potentially distributed) software controller. The controller can define and flexibly change arbitrary routes (i.e., forwarding rules): routes are not limited to shortest paths and are not necessarily based on IP destination addresses (only), but can depend on Layer-2, Layer-3 and Layer-4 header fields (e.g., TCP ports), and beyond.

While the logically centralized perspective offered by SDNs has the potential to significantly simplify network operations, an SDN still has to be regarded as a distributed system which comes with fundamental challenges. One such fundamental challenge regards the consistent implementation of *route updates*: In order to update a route r_1 to a route r_2, the controller needs to communicate the new forwarding rules to all nodes. However, as both the transmission as well as the installation of rules take time and are subject to variance [1], inconsistencies can be introduced during the update: For example, the same packet may still be forwarded according to the old rules (of r_1) at some nodes while it is forwarded already according to the new rules (of r_2) at others. The resulting *actual* routes may transiently violate basic consistency properties such as *loop-freedom* [2].

One possible solution is to use a *2-Phase Commit Protocol (2PC)* and *(packet) tagging* [3]: In a first round, the controller communicates the new rules of r_2 to all nodes. However, the rules only apply to packets with a certain tag (say, "*new*"), and hence existing packets without the *new* tag are still forwarded according to r_1. Once all nodes confirmed the successful installation of the new rules, in a second round, the controller instructs the ingress ports of the network to tag all packets with "*new*", forcing the packets to use the new route r_2. The 2PC protocol ensures a strong *per-packet consistency* [3]: each packet will be forwarded according to r_1 *(exclusive-)*or r_2, and loops are avoided. However, the use of tagging is undesirable, as it consumes header space in the packets and requires the installation of additional forwarding rules (matching the tagged packets), wasting precious switch memory; moreover, tagging can be problematic in the presence of middleboxes which change headers [4].

An alternative approach to ensure loop-free updates, without tagging, is to communicate updates to nodes in a staged manner: The controller first updates only a *safe subset* of nodes $V_1 \subseteq V$. After these nodes asynchronously installed the new rules, they send an acknowledgement to the controller, which then in turn schedules the next subset $V_2 \subseteq V$ of nodes to update, until the final subset V_k completes the route update. This protocol, which is based on a *node-ordering technique* [5], does not require packet tagging, and, as argued in [2], also has the advantage that some of the edges of r_2 become available earlier to packets: there is no need to wait for the full installation of r_2.

A. Our Contributions

This paper initiates the study of scheduling *fast* loop-free network updates, i.e., updates which require a minimal number of controller interactions while providing transient consistency guarantees. We consider a model where network routes can follow arbitrary paths and are not necessarily destination-based (arguably a key benefit of SDN [6]). We ask: *How many communication rounds k are needed to update a network in a (transiently) loop-free manner?*

Besides the inherent advantages of being faster, fewer rounds improve the agility of the network's control loop, also ensuring faster reactions to failures or changing workloads [1]. For example *SWAN* aims at upper bounding schedules to at approximately 3 rounds in their inter-datacenter Wide-Area Network (WAN) [7].

We show that answering this question is difficult. By leveraging an interesting problem symmetry, we first (constructively) show that deciding whether a k-round schedule exists can be decided efficiently for $k = 2$. However, the problem becomes NP-complete already for $k = 3$. Moreover, we show that there

Manuscript received February 10, 2017; revised October 6, 2017; accepted November 22, 2017. K.-T. Foerster is with Aalborg University, Denmark. A. Ludwig is with TU Berlin, Germany. J. Marcinkowski is with University of Wrocław, Poland. S. Schmid is with University of Vienna, Austria.

exist problem instances which require $\Omega(n)$ rounds, where n is the network size. We will also prove that just aiming to "greedily" update a *maximum* number of nodes in each round (as proposed in previous work [2], however, for a different model) may result in $\Omega(n)$-round schedules in instances which actually can be solved in $O(1)$ rounds; even worse, a *single* greedy round may inherently delay the schedule by a factor of $\Omega(n)$ more rounds.

Given these negative results, we propose an attractive alternative to the utterly strict loop-free requirement: *relaxed loop-freedom*. Relaxed loop-freedom is motivated by the observation that loops are only really problematic if they occur on the (changing) path between source and destination: topological loops in other parts of the network will never receive any new packets. We argue that relaxed loop-freedom not only expresses better the actually desired consistency in practice, but we also show that it comes with interesting benefits: We show that while relaxed and strong loop-freedom are equivalent for $k < 3$, in general, a relaxed loop-free update schedule can be $\Omega(n)$ times shorter than the best strong loop-free update schedule. More importantly, we prove that $O(\log n)$-round relaxed loop-free schedules always exist and can be computed efficiently: we present an elegant algorithm accordingly. We also provide an asymptotically matching lower bound, proving that our bound is tight: in a problem instance with heavily nested new routes, our algorithm needs $\Omega(\log n)$ rounds in the worst case.

Finally, we also establish a connection to a line of works by Wattenhofer et al. [2], [8], [9], which focuses on destination-based routing. In particular, we prove that our hardness results can be transferred to their model, providing new insights into their problem as well.

B. Organization

The remainder of this paper is organized as follows. Section II introduces our formal model. Section III studies the strong consistency model for transient loop-freedom, and Section IV studies relaxed loop-freedom. Section V complements our formal worst-case analysis by reporting on simulation results for different synthetic workloads. After reviewing related literature in Section VI, we conclude our work in Section VII.

II. Model

We are given a network and two routes r_1 (the *old route*) and r_2 (the *new route*). Both r_1 and r_2 are simple directed paths. Initially, packets are forwarded (using the *old rules*, henceforth also called *old edges*) along r_1, and eventually they should be forwarded according to the new rules of r_2. Packets should never be delayed or dropped at a node: whenever a packet arrives at a node, a matching forwarding rule should be present.

Without loss of generality, we assume that r_1 and r_2 lead from a source s to a destination d. Since nodes appearing only in one or none of the two paths are trivially updatable, we focus on the network G induced by the nodes V which are part of *both* routes r_1 *and* r_2, i.e., $V = \{v : v \in r_1 \wedge v \in r_2\}$. Thus, we can represent the routes as $r_1 = (s = v_1, v_2, \ldots, v_\ell = d)$ and $r_2 = (s = v_1, \pi(v_2), \ldots, \pi(v_{\ell-1}), v_\ell = d)$, for some permutation $\pi : V \setminus \{s, d\} \to V \setminus \{s, d\}$ and some number ℓ. In fact, we can represent routes in an even more compact way: we are actually only concerned about the nodes $U \subseteq V$ which need to be updated. Let, for each node $v \in V$, $out_1(v)$ (resp. $in_1(v)$) denote the outgoing (resp. incoming) edge according to route r_1, and $out_2(v)$ (resp. $in_2(v)$) denote the outgoing (resp. incoming) edge according to route r_2. Moreover, let us extend these definitions for entire node sets S, i.e., $out_i(S) = \bigcup_{v \in S} out_i(v)$, for $i \in \{1, 2\}$, and analogously, for in_i. We define s to be the first node (say, on r_1) with $out_1(v) \neq out_2(v)$, and d to be the last node with $in_1(v) \neq in_2(v)$. We are interested in the set of to-be-updated nodes $U = \{v \in V : out_1(v) \neq out_2(v)\}$, and define $n = |U|$. Given this reduction, in the following, we will assume that V only consists of interesting nodes ($U = V$).

A. Strong Loop-Freedom

We want to find a *schedule* $U_1, U_2, \ldots, U_k$ with minimum k, i.e., a sequence of subsets $U_t \subseteq U$ where the subsets form a partition[1] of U (i.e., $U = U_1 \cup U_2 \cup \ldots \cup U_k$), with the property that for any round t, given that the updates $U_{t'}$ for $t' < t$ have been made, all updates U_t can be performed "asynchronously", that is, in an arbitrary order without violating loop-freedom. That is, consistent paths will be maintained for any subset of updated nodes, independently of how long individual updates may take.

More formally, let $U_{<t} = \bigcup_{i=1,\ldots,t-1} U_i$ denote the set of nodes which have already been updated before round t, and let $U_{\leq t}$, $U_{>t}$ etc. be defined analogously. Since updates during round t occur asynchronously, an arbitrary subset of nodes $X \subseteq U_t$ may already have been updated while the nodes $\overline{X} = U_t \setminus X$ still use the old rules, resulting in a temporary forwarding graph $G_t(U, X, E_t)$ over nodes U, where $E_t = out_1(U_{>t} \cup \overline{X}) \cup out_2(U_{<t} \cup X)$. We require that the update schedule $U_1, U_2, \ldots, U_k$ fulfills the property that for all t and for any $X \subseteq U_t$, $G_t(U, X, E_t)$ is loop-free.

Later in this paper, we will sometimes refer to this definition of loop-freedom as the *Strong Loop-Freedom* (SLF), to distinguish it from *Relaxed Loop-Freedom* (RLF). By default, throughout this paper, the term loop-freedom without additional qualifier will refer to the strong variant.

Example. Fig. 1 illustrates our model: We are given two routes (the old rules of r_1 are *solid*, the new ones of r_2 are *dashed*), see Fig. 1 (*left*). We focus on the updateable nodes which are shared by the two routes. Thus, in our example, the update problem can be reduced to the 5-node chain graph in Fig. 1 (*right*). Throughout this paper, we will stick to this representation, and will indicate the old route r_1 using *solid lines*, and the new route r_2 using *dashed lines*. Moreover, we will depict the initial network configuration (before the update) such that the old route goes from left to right. In the following we will call an edge (u, v) of the new route r_2 *forward*, if v is closer (with respect to r_1) to the destination, resp. *backward*, if u is closer to the destination. It is also convenient to name nodes after their outgoing dashed edges (e.g., *forward* or *backward*);

[1] We can w.l.o.g. assume a partition of U, as a later identical update of an already updated node v does not change any forwarding behavior at v.

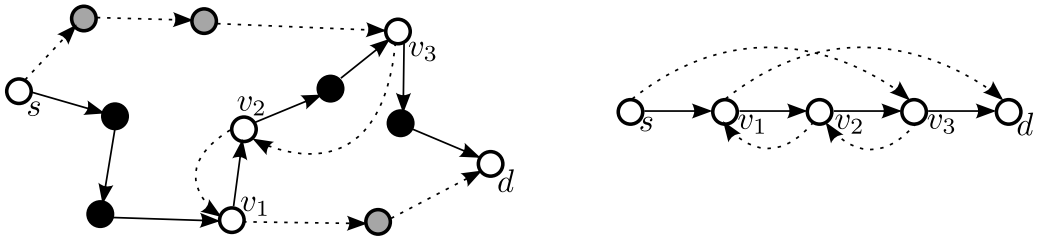


Fig. 1. Overview of model and reduction. The network on the *left* is reduced to the line representation on the *right*. The solid lines show the old route r_1 and the dashed lines show the new route r_2. Nodes shown in white are the only ones which are part on both paths, and hence relevant for the problem. Regarding the two-letter codes introduced in Section III-A, s is **FF**, v_1 is **FB**, v_2 is **BB**, and v_3 is **BF**, see Fig. 3 for a more detailed explanation of two-letter codes. Hence, the here shown problem does not permit a 2-round schedule, as it contains a **BB**, which cannot be updated in the first or last (second) round.

similarly, it is sometimes convenient to say that we *update an edge* when we update the corresponding node. Finally, we will treat the terms *edge* and *rule*, as synonyms in this paper.

B. Relaxed Loop-Freedom

In this paper, we will also propose a weaker notion of loop-freedom, denoted as *Relaxed Loop-Freedom* (RLF). Relaxed loop-freedom is motivated by the practical observation that transient loops are not very harmful if they do not occur between the source s and the destination d.

Example. In Fig. 2, any SLF schedule needs to update v_2 before updating v_3, which in turn requires to first update v_4, which requires to first update v_5, and so on, requiring $n-2$ rounds in total. However, by updating s in the first round, all new packets take the path s, v_{l-1}, d, skipping over v_2 to v_{l-2}. As such, when updating $v_2, \ldots, v_{l-2}$ in the second round, no new packets from s will enter a loop. After, we can update v_{l-1} in the third round. Recall that SLF required a factor $\Omega(n)$ more rounds, which is worst possible.

RLF Definition. As thus motivated, our following definition of relaxed loop-freedom will focus on the loop-freedom of current delivery path from the source s. The nodes not currently on the path from s to d may be updated even if they induce transient loops, until they eventually rejoin the path from s to d.

Concretely, and similar to the definition of SLF, we require the update schedule to fulfill the property that for all rounds t and for any subset X, the temporary forwarding graph $G_t(U, X, E'_t)$ is loop-free. The difference is that we only care about the subset E'_t of E_t consisting of edges *reachable from the source* s.

Impact of RLF. If relaxed loop-freedom is preserved, only a constant number of packets can loop: we will never push new packets into a loop "at line rate". In other words, even if switches acknowledge new updates late (or never), new packets will not enter loops.

In practice, the technique of relaxed loop-freedom can also be applied without any packets entering loops, by trading in a small amount of extra update time. Towards this end, we make the assumption that the latency, or time of flight, for any packet in the network is upper bounded by some ℓ, unless the packet enters a loop. Observe that loops may only be introduced in RLF for paths that are disconnected from the packet source. Hence, after a time of ℓ, each such path will be drained of packets. In other words, by waiting for ℓ after each round, no packets will loop – though the forwarding rules may still contain loops. As the update time of switches is commonly larger than the latency [1], the extra time spent will be small in comparison. Even if the switch update time were to be instantaneous, the latency of the packets used for controller-switch interaction will be similar to ℓ, inducing only a small constant-factor overhead.

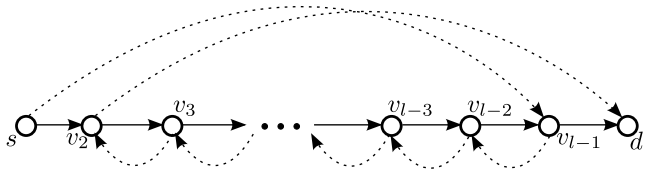


Fig. 2. RLF vs SLF: An SLF schedule needs to update backward edges one by one from left to right, requiring $\Omega(n)$ rounds; for RLF, an $O(1)$-round schedule exists, e.g., $U_1 = \{s\}$, $U_2 = \{v_2, \ldots, v_{l-2}\}$, $U_3 = \{v_{l-1}\}$.

In theory, we cannot assume a bound on ℓ, and the definition of RLF does not put a limit on how long the packets will loop.[2] However, the number of looping packets is still bounded by the packets on route of the affected path segments. To illustrate said number with an example, assume a rate of 1Gbit/s and a 10ms latency segment: at most 10Mbit of packets will loop.

III. Fast Updates Are Difficult

How many rounds are needed to update a network in a (strongly) loop-free manner? At first sight, the problem may seem difficult: the problem of breaking cycles even in a single round, is related to the well-known NP-hard Feedback Arc Set Problem [10]. On the other hand, our graphs have a very special structure, as they essentially only consist of two simple paths (namely the old and the new route).

In this section, we show that updating networks quickly is difficult, even for such simple graphs: while we can exploit an interesting symmetry property to efficiently compute 2-round schedules (Section III-A), it turns out that deciding whether 3-round schedules exist is already NP-complete (Section III-B). Also recall our example from Fig. 2 which shows that there exist problem instances which cannot be updated in less than $\Omega(n)$ rounds.

[2]Our *Peacock* algorithm presented later limits it to one round, see Section IV-B.

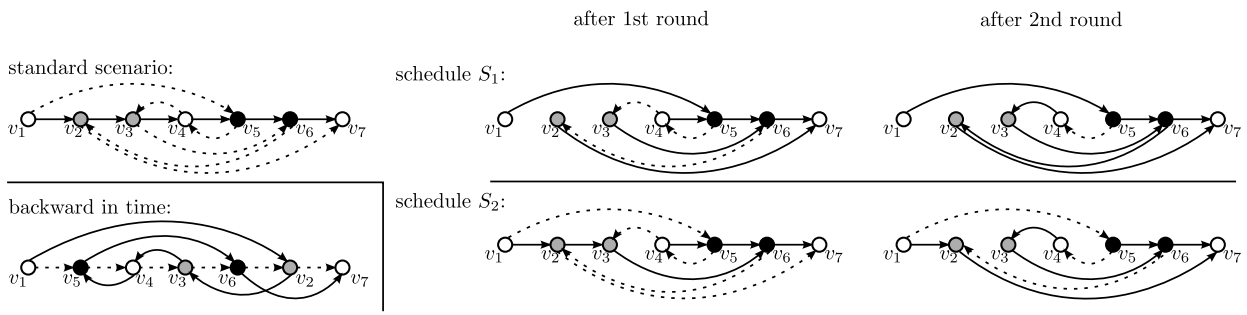


Fig. 3. *Left:* "Looking backward in time", an example with reversed update pattern (from *dashed* to *solid* path). We obtain the following classification: v_1 is **FF**; v_2, v_3 are **FB**; v_4 is **BB** and v_5, v_6 are **BF**. *Right:* Intuition why node updates can be moved from round 2 to round 1 or 3. There are two different valid update schedules for the standard scenario. Schedule S_1 is updating everything as early as possible with, e.g., **FB** node v_2 in round 1 and **BF** node v_6 in round 2, with $U_1 = \{v_1, v_2, v_3\}$, $U_2 = \{v_4, v_6\}$, and $U_3 = \{v_5\}$. Schedule S_2 is updating everything as late as possible, e.g., v_2 in round 2 and v_6 in round 3, with $U_1 = \{v_3\}$, $U_2 = \{v_2, v_4\}$, and $U_3 = \{v_1, v_5, v_6\}$. We depict updated nodes without their outgoing solid edges (no new packets will be sent this way), and dashed edges turn into solid edges.

A. 2-Round is Easy

Before we show how to find 2-round update schedules efficiently, let us introduce the following edge (resp. node) classification, which will be useful more generally. We already discussed the notion of *forward* and *backward* dashed edges (resp. nodes), indicating whether a dashed edge points in the same direction as the solid edge. This distinction is useful as, for example, it is always safe to update any number of forward-pointing edges: they can never introduce any loops.

A key insight is that the network update problem features a symmetry property: A legal update schedule leading from the old policy to the new policy *backward* must also be a legal update schedule leading from the new policy to the old policy. We introduce the following additional classification: we classify edges also with respect to an update schedule leading from the new policy to the old policy, "looking backward in time". That is, we consider updating edges from the dashed ("new" r_2) rules to the solid ("old" r_1) ones, starting with the last round. Given this perspective, we can classify the old (*solid*) rules as *backward* or *forward* relative to the new ones (*dashed*): we just need to draw the new route as a straight path and see, if the old rule points forward or backward.

Based on this classification, we propose two-letter codes to describe the nodes—the first letter will denote, whether the outgoing dashed edge points forward (**F**) or backward (**B**). Similarly, the second letter will describe the solid edge relative to the dashed path. We refer to Fig. 1 and Fig. 3 for examples.

Recall that any set of forward rules cannot introduce a loop: hence, 1) it is always safe to update any subset of **FB** and **FF** rules in the first round, and 2) as a schedule backward must also be legal, it is also always safe to update any **BF** and **FF** rules in the last round (which are then **FB** and **FF** rules).

Given this intuition, we can determine whether two rounds are sufficient: if there is any **BB** edge, it can neither be updated in the first round, nor in the last, so two rounds are not enough. Otherwise, we update **FB**s in the first round, **BF**s in the second round, and have complete freedom on when to update the **FF** nodes.

B. 3-Round is Hard

Unfortunately, it is already NP-complete to decide whether a problem instance has a 3-round update schedule.

Theorem 1: Deciding whether a $k = 3$-round schedule exists is NP-complete.

The k-round problem is certainly in NP: the correctness of a schedule can be verified easily. The hardness proof proceeds as follows. First we make a couple of observations which allow us to narrow the ground for choosing 3-round update schedules, reducing the problem to the selection of edge subsets. Second, we will present a slight modification of 3-SAT and—using gadgets—transform it into an instance of the edge selection problem. Finally, the graph built using the gadgets will be patched up to a proper instance of the network update problem (namely, two paths traversing the same set of nodes).

1) Classifying Nodes: When we aim for three rounds, the **FB** nodes can be updated in the first or second round. As we will observe in the following, it is however never *necessary* to update **FB** nodes in the second round: everything can just as well be done in the first round.

Lemma 1: If there exists a 3-round update schedule $\mathtt{S}$ which updates any nodes $V' \subseteq V$ of type **FB**, then there is also a 3-round update schedule which updates all nodes of V' in the first round. The same holds true for nodes of type **FF**.

Proof: Consider the temporary forwarding graph $G_t(X) = (U, X, E_t)$ during the t^{th} round update of $\mathtt{S}$, for $t \in \{1, 2\}$. Since $\mathtt{S}$ is correct, both $G_1(X) = (U, X, E_1)$ and $G_2(X) = (U, X, E_2)$ are loop-free, for any subset $X \subseteq U_t$. By moving updates of forwarding nodes **FB** and **FF** from round 2 to round 1, we will make G_2 only sparser, and will hence not introduce loops. However, also G_1 will remain loop-free, as the forwarding edges **F**· respect the topological order of r_1. ■

The same argument also holds in the other direction, using our "backward perspective": We can move **BF** (and **FF**) updates to the last round. Therefore, without loss of generality, we focus our analysis on schedules where all the **BB** nodes are updated in the middle (i.e., second) round, all **FB** nodes in the first round, and all the **BF** nodes in the last round. Thus, the problem boils down to finding a distribution of the **FF**

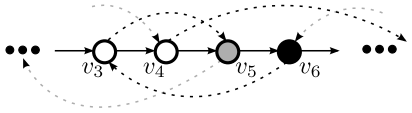


Fig. 4. Choosing the right set of **FF** nodes is important. An update of only v_4 would enable the **BB** node v_6 to be updated in the second round. An additional update of v_3 would then lead to a loop (note that v_5 will definitely not be updated in the first round).

updates to the first and the third round. As we will show in the following, finding such a distribution is NP-hard.

Fig. 3 provides intuition for why **FB** updates can be moved into the first round and **BF** updates in the third round. The right part shows two different 3-round schedules for a given scenario. The **FB** node v_3 needs to be updated in the first round in any valid 3-round schedule, since the only **BB** node v_4 needs to be updated in the second round. Schedule S_2 updates the **FB** node v_2 in round 2 and schedule S_1 shows that it would also be possible to update it in the first round. The **BF** node v_6 is updated in round 2 in S_1 and delayed to round 3 in S_3. According to Lemma 1, there also exists a schedule S_3 updating every **FB** node in the first round and every **BF** node in the third round ($U_1 = \{v_1, v_2, v_3\}, U_2 = \{v_4\}, U_3 = \{v_5, v_6\}$).

In order to be able to update every **BB** node in the second round, one needs to be careful which (of the **FF**) nodes to update in the first and which in the third round. Fig. 4 shows a snippet of a line where the **BB** node v_6 needs to be updated in the second round. An update of **FF** node v_4 in the first round would enable this update for the second round, but updating the **FF** node v_3 as well would render an update of v_6 impossible. Node v_5 is **B**$\cdot$ and cannot be updated in the first round, and hence an update of v_6 would result in a loop ($v_3 \to v_5 \to v_6 \to v_3$).

2) Modifying 3-CNF: For our reduction, we take an instance of the 3-SAT problem, $\mathcal{C}$, which we will eventually transform into an instance of a network update problem that is updatable in 3 rounds, if and only if the formula is satisfiable. However, we will first modify $\mathcal{C}$, using a standard construction, and replace each appearance of a variable in $\mathcal{C}$ using a new variable: concretely, a variable appearing λ times in $\mathcal{C}$ decays into $\lambda + 4$ new variables. By this trick, we will reduce the number of times any (new) variable appears in the (new) formula, allowing us to implement the low in- and out-degree requirements of our network update problem.[3] We create the following clauses:

1) For every variable x, we create variables
$$x_0, x_1, \ldots, x_{p_x}, x_l, \overline{x}_0, \overline{x}_1, \ldots, \overline{x}_{n_x}, \overline{x}_l,$$
where p_x is the number of positive appearances of x, and n_x the number of negative appearances. In every clause we replace the literals with the appropriate new variables (from the collections $x_1, \ldots, x_{p_x}$ and $\overline{x}_1, \ldots, \overline{x}_{n_x}$). Also, for every original variable x we add an "*assignment clause*" $(x_0 \vee \overline{x}_0)$.
2) For every original variable we add "*implication clauses*" $(x_i \to x_{i+1})$ for $i = 0 \ldots p_x - 1$ and $(\overline{x}_i \to \overline{x}_{i+1})$ for $i = 0 \ldots n_x - 1$; the last implications, for $i = p_x$ resp. $i = n_x$ must lead to x_l and $\overline{x}_l$ respectively ($(x_{p_x} \to x_l)$ and $(\overline{x}_{n_x} \to \overline{x}_l)$).
3) Finally, for every original variable x, we add an "*exclusive clause*" $(\neg x_l \vee \neg \overline{x}_l)$.

For each variable x, with the *assignment clause*, we ensure that at least one literal is true; with the *exclusive clause* we ensure that at most one literal is true; and with the *implication clause*, we ensure that the value is consistently preserved through all clones.

It is straightforward to translate any satisfying assignment of variables of one formula to the other, therefore the satisfiability problem for the new formula is equivalent to the original one. We will refer to the modified formula by $\mathcal{C}'$.

3) Creating and Connecting the Gadgets: For the reduction, we will create (network) gadgets representing the different clauses. Concretely, first, for every variable x_i in $\mathcal{C}'$, we create a node x_i, which will be of type **FF** (we will refer to the node using the variable's name). The idea is that updating the node in the first round will correspond to the positive valuation of the variable. In general, we will create for each gadget a path of solid edges pointing upward; eventually, we will connect these paths from left to right (using solid edges), to establish route r_1.

Every original clause $\mathcal{K}$ is encoded as a gadget in the graph using a separate solid path (drawn as a vertical line pointing upwards) with the variable-related (x_i) **FF** nodes on it. Above those nodes on the path, there is a **BB** node, $v_1^{\mathcal{K}}$, the starting point of a backward, dashed edge that will end just below the variables with a node $v_2^{\mathcal{K}}$ (Fig. 5 *left*). The backward edge and the solid path form a cycle, which needs to be disconnected in the first round. The only way to do this, is by updating at least one of the variable–related edges. Obviously, the dashed, forward edges starting at the **FF** nodes inside the clause must reach outside the clause-related backward edge $(v_1^{\mathcal{K}}, v_2^{\mathcal{K}})$. In fact, they will end just below the nodes representing the variables that are followed in the implications (see Fig. 5 on the *right*), so the dashed edge starting at the node x_i will point to the node x_{i+1} in a gadget representing another clause (actually it points to a special node x_{i+1}^{IN} that serves as a connecting point: we will present the details in the next paragraph; the last x_i will point to x_l situated in the exclusive gadget clause for x, which we describe later). For convenience, we order the clauses from left to right, and name the variables x_i, y_i, and z_i with increasing i from the left to the right according to this order. Thus, every dashed edge connecting two different gadgets points rightwards when it is a forward **F**$\cdot$ edge, and leftwards when it is a backward **B**$\cdot$ edge.

For each implication clause $\mathcal{K} = (x_i \to x_{i+1})$, we already have the nodes representing the two variables x_i and x_{i+1} (lying on two separate solid paths belonging to their respective gadgets) and a dashed edge from the antecedent, x_i, to a new node x_{i+1}^{IN} placed below the consequent one. The gadget (Fig. 5 *right*) ensures that if x_i is updated in the first round, then x_{i+1} must be updated as well, or there will be a cycle in the second round ($x_i \to x_{i+1}^{\text{IN}} \to x_{i+1} \to x_{i+1}^{BB} \to x_i^H \to x_i^{\text{IN}} \to x_i$): we draw a new node x_{i+1}^{BB} of type **BB** slightly above x_{i+1} (on its solid path) and a dashed edge pointing from it to another new

[3]The occurrence of each variable will be limited by 2, with some clauses having just 2 variables in them. The different case that all clauses have exactly 3 distinct variables, each occurring in at most 3 clauses, is in P [11].

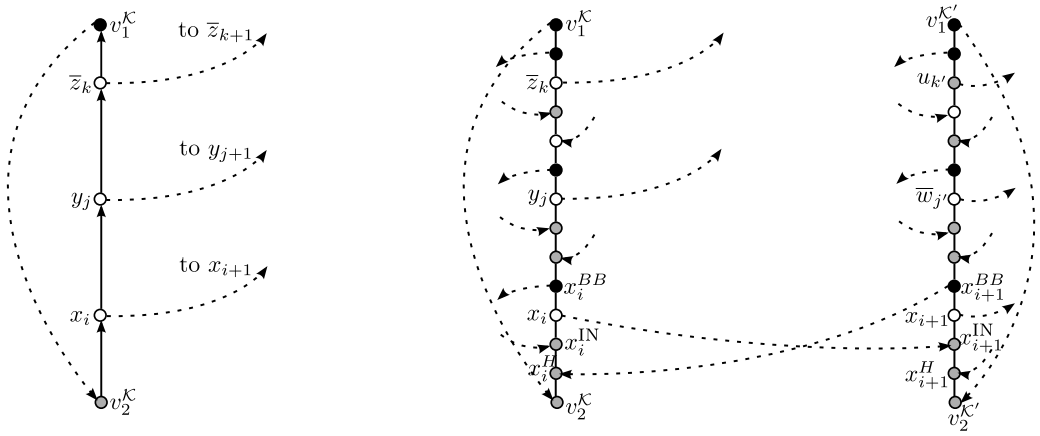


Fig. 5. *Left:* Gadget for clause $x_i \vee y_j \vee \overline{z}_k$. At least one node needs to be updated to prevent the loop over $v_1^{\mathcal{K}}$ and $v_2^{\mathcal{K}}$. *Right:* More details about the gadget including also the implication clause $x_i \to x_{i+1}$ representation, and a second clause $x_{i+1} \vee \overline{w}_{j'} \vee u_{k'}$. It is ensured that x_{i+1} is updated if x_1 is updated, otherwise the **BB** edge from x_{i+1}^{BB} would form a cycle. White nodes will eventually be **FF**, black nodes **BB**. The grey nodes will later be ensured to be of type **B**·, to guarantee that they cannot be updated in the first round.

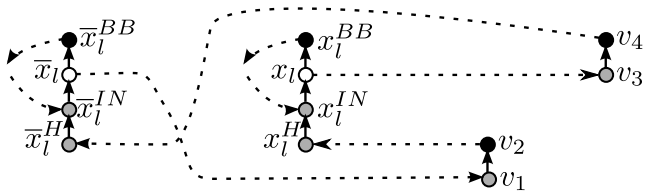


Fig. 6. Gadget for exclusive clause. An update of either x_l or $\overline{x}_l$ prevents the other one from being updateable: the **BB** nodes v_2 and v_4 would form a cycle in the second round.

helper node (to meet the in-degree constraint of the network update problem), x_i^H, slightly below x_i (in the figure we draw it below x_i^{IN} as well).

Then for every exclusive clause $\mathcal{K}_x = (\neg x_l \vee \neg \overline{x}_l)$ (shown in Fig. 6), we draw four solid paths. On the first, the **FF** node $\overline{x}_l$ is drawn and a dashed edge pointing from it to another helper node v_1 lying on the third solid path. Similarly, x_l on the second path points, with its forward dashed edge, towards v_3, which we place as the last of the four solid paths. Above v_1 and v_3 we draw another pair of **BB** nodes, v_2 and v_4 respectively. Then v_2 points back to the second solid path with its backward dashed edge, to another new node, x_l^H placed just below x_l on the first path. In the same manner, the backward edge starting at v_4 ends with $\overline{x}_l^H$ below $\overline{x}_l$. This way, updating both x_l and $\overline{x}_l$ in the first round will result in a cycle in the second round, since, as we know, all **BB**s *must* be updated in the second round. The cycle which can exist in the second round includes the following nodes $\overline{x}_l, v_1, v_2, x_l^H, x_l^{\text{IN}}, x_l, v_3, v_4, \overline{x}_l^H, \overline{x}_l^{\text{IN}}, \overline{x}_l$. x_l and $\overline{x}_l$ have been updated in the first round, the nodes v_2 and v_4 have been updated in the second round and the rest of the nodes in the cycle (the grey nodes) have not been updated yet. It will be later ensured that they are of type **B**· and therefore cannot be updated in the first round, hence making a scenario possible where they are delayed until the end of the second round. Therefore an update of both x_l and $\overline{x}_l$ is not possible in the first round.

While the composition of gadgets described so far is not yet a proper instance of a network update problem, we can already make some observations about the graph.

Theorem 2: If setting $V_{\mathcal{T}} \subset Var(\mathcal{C}')$ to `true` satisfies the formula, then there is no cycle ($\Rightarrow$). Moreover, a cycle-free update schedule gives us a satisfying variable assignment ($\Leftarrow$).

Proof: We prove the two directions $\Rightarrow$ and $\Leftarrow$ in turn.

$\Rightarrow$: Cycles are composed of: dashed edges starting at $V_{\mathcal{T}}$ nodes, solid edges starting at any other nodes to get somewhere, and any edges starting at **BB** nodes to get back. We will show that by following an arbitrary path consisting only of the listed edge types, we will never return to the starting point of the path. If the path ever chooses to take an **FF** updated dashed edge (starting at x_i), it will need to continue with edges starting at x_{i+1} up to x_l (this is ensured by the implications), and there is no way back from there: it cannot constitute a cycle. Conversely, a path which does not take any **FF** dashed edges would not be able to jump from one of the solid, vertical paths to another one more to the right, so if it returns to the starting point, it must use nodes lying on one of the solid paths. At the same time, a cycle on one of the solid paths would mean that one of the clauses is not satisfied, which contradicts the definition of $V_{\mathcal{T}}$.

$\Leftarrow$: Clearly, the construction ensures that if the formula $\mathcal{C}'$ is not satisfiable, when we have a selection of **FF** nodes which make the situation with *all* **BB** edges (which must be updated) acyclic then each clause must be true: it contains a true variable showing a path out of the cycle. ■

4) Connecting the Pieces: The presented gadgets leave us with a number of independent solid paths and many dashed edges starting at nodes of particular types (**FF** or **BB**). In order for the network to represent a valid problem instance, we need to connect the solid paths as well as the dashed paths. Our goal is to connect the solid path from left to right (and vertical lines are from bottom to top). The dashed path will be more complicated.

Let us first focus on connecting the *dashed* edges to a path. From the endpoint of each dashed edge, we will draw a backward dashed edge to a completely new node (one for each)

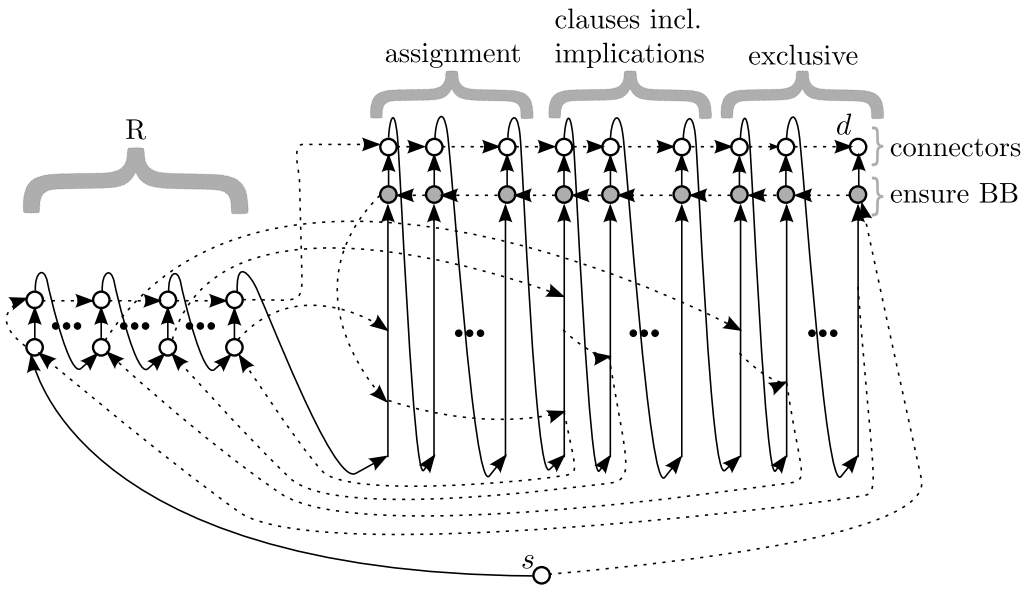


Fig. 7. Overview of how the path is connected. The grey nodes are used to connect everything into one solid path. They also join the dashed path at the last nodes. This way, all nodes in R (*white cycles*) are of type **FF**.

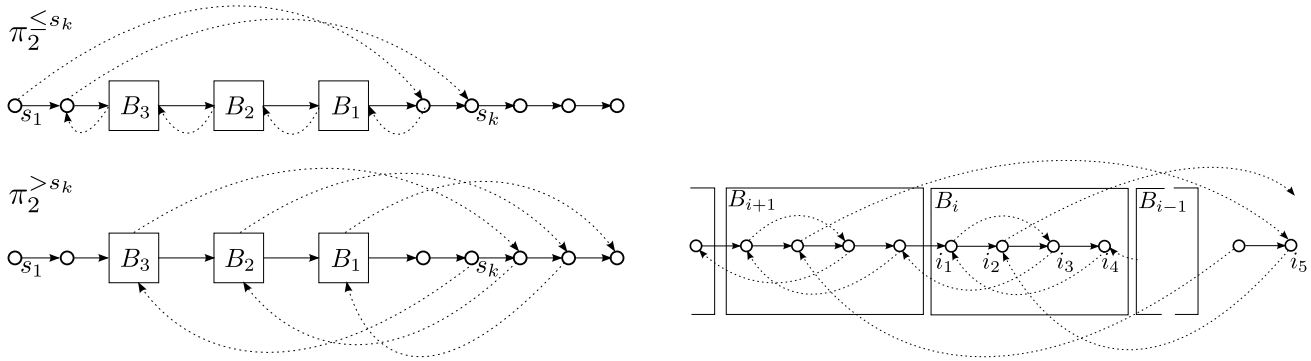


Fig. 8. Pattern of a scenario where maximizing the number of updates per round will result in a $\Omega(n)$-round schedule, although a $O(1)$-round schedule would be possible. *Left:* An overview where $\pi_2^{\leq s_k}$ shows the edges of the new route before s_k and $\pi_2^{>s_k}$ those behind s_k. *Right:* A detailed representation of the blocks B_i.

placed far left from our solid paths. Hence, all nodes in R — the set of new nodes — will appear earlier in the concluding solid path: edges pointing to R are backward, edges pointing away from R forward. Then we connect all the resulting 2-length dashed paths (including the previously constructed dashed ones, and the new ones pointing to R), using forward dashed edges starting at the new nodes, as described in the following.

Some of the nodes in our gadgets were of type **BB** while the others were **FF**. Recall that these type-properties are fairly local: we only need to look at the next node on the solid path and determine if it is preceding on the dashed path. To preserve the types of the nodes, we must therefore connect the 2-length paths in a correct order — first come the **FF** dashed edges, then the clause-related downward-pointing **BB** edges and in the end implication-related horizontal **BB** edges. In each of these groups the edges starting more to the left should precede those more to the right. Also – to ensure, that all the type assignment clause-related edges indeed start with a **BB** node – above each of those nodes $v_1^{\mathcal{C}}$, in their respective gadgets, we draw a new node $v_b^{\mathcal{C}}$. On each of the four solid paths used in the gadgets for the exclusive clauses $\mathcal{K}_x$, we do the same: we create nodes $v_b^{\mathcal{K}_x}{}_1, \ldots, v_b^{\mathcal{K}_x}{}_4$. Then we connect all the v_b's into a dashed path going from right to left. The path must be connected to the beginning of the dashed path we composed before, which will ensure the **BB** property of the previous nodes: the solid edge now points backwards relative to the dashed path. Each of the new $v_b^{\mathcal{C}}$ nodes will be of type **BF**. The nodes in R are ordered so that the dashed path ends at the leftmost node.

The nodes of R are positioned in a row, followed by our vertical solid paths. We draw a new node above each of them, connect it with a solid edge and connect the new node with what is next in the row, from the top of a vertical path to the bottom of the next one (Fig. 7). This way, we finally have one solid path. The new nodes are connected by a chain of forward dashed edges (so they can all be updated). In the end we add a starting node, which points with the solid edge to the leftmost R node, and with the dashed edge to the beginning of the dashed path which is the beginning of the path we constructed to ensure the **BB** properties (this point is **BF**).

It is important to note that in the last steps we have not

jeopardized the reduction by introducing disconnections of the gadget-**BB** edges, nor have we created any loops that cannot be easily broken (by updating all the empty nodes in Fig. 7). Therefore, the possibility of making the second round cycle-free in our instance is still equivalent to the satisfiability of $\mathcal{C}'$, which makes the 3-round network update problem NP-hard.

C. It's bad being greedy

Given the NP-hardness result of Theorem 1, one may wonder whether simple approximation algorithms exist. While we cannot prove the opposite, we conjecture that the problem is generally hard to approximate. To give some intuition, in the following, we show that a "greedy" approach which tries to maximize the number of updatable edges in each round (essentially the model studied in [2]) can fail miserably. In fact, a single greedy round may unrevokably change the required number of rounds from $O(1)$ to $\Omega(n)$.

Fig. 8 shows a scenario where a greedy update takes $\Omega(n)$ rounds even though an $O(1)$ round solution exists. The left side shows the general structure of the scenario which consists of several blocks B_i (more details on the right side). These blocks are connected via backward edges one by one, e.g., see the edge emerging from i_3. If a greedy algorithm picks all forward edges to be updated in a first round, it will include the nodes i_1 and i_2 as well as their representatives in the other blocks. The update of the i_1-type nodes essentially leads to a situation reminiscent of the one shown in Fig. 2, where many backward rules must be updated one after the other. Delaying the i_1-type nodes on the other hand will make it possible to update most of the backward edges in the next round, since the cycle is broken by the edges outgoing from the i_2-type nodes. This allows for an update in 4 rounds, independent of n. In case of the greedy algorithm, each additional block will increase the number of rounds by two. Each block consists of 4 nodes within the block and an additional node for connectivity to the right part of the line, resulting in $2n/5$ rounds: up to $n/10$ additional rounds are required.

IV. Relaxed Loop-Free Updates Are Tractable

Given the potentially large number of rounds required to update a network in a strongly loop-free manner, we now propose to relax loop-freedom to only include *actually used paths*, between source and destination. We believe that this is an attractive alternative: although some unlucky packets currently on transit on an edge may end up in a (temporary) loop, we will never route any packets entering the network at the source into a loop. Moreover, as we will see, relaxing the loop-freedom is also attractive because it enables fast and computationally tractable updates. In particular, we will present a fast and elegant algorithm which never requires more than $O(\log n)$ rounds: a potentially large gain given the $\Omega(n)$ lower bound for stronger models.

A. First Observations

We first observe that for $k < 3$, the relaxed problem variant does not help: in this case, relaxed and strong loop-freedom are equivalent. To see this, recall Section III-A where we showed that it is easy to decide if rules r_1, r_2 permit a 2-round update schedule in SLF: If no **BB** edge is present, update all **FB**s in round one, **BF**s in round two, and **FF**s in either round. Else, two rounds do not suffice.

We may ask: Are there networks that do not permit a 2-round update schedule in SLF, but in RLF? As it turns out, the answer is no:

Even using relaxed loop-freedom, a **BB** node cannot be updated in the first round. As every schedule must also be valid in reverse, a **BB** node cannot be updated in the last (second) round either under RLF. Furthermore, every schedule that satisfies SLF (recall Section III-A), also satisfies RLF.

Observation 1: The problem instances that permit 2-round update schedules are identical for strong and relaxed loop-freedom.

While there is not much we can gain from relaxing the notion of loop-freedom for $k < 3$, in general, the benefits can be significant. To see this, recall the example in Fig. 2: SLF required $\Omega(n)$ rounds, while RLF permitted a 3-round solution.

B. Algorithm and Upper Bound

Before presenting our scheduling algorithm in detail, let us introduce some concepts. During its execution, our algorithm will repeatedly perform *node merging*: when updating a node v, we will merge it with the node $out_2(v)$ it pointed to with its dashed edge. This can safely be done after each round, due to the irrelevance of already updated nodes (they will simply forward packets to the next node, without influencing the remaining problem at hand). I.e., after merging, we treat v and $out_2(v)$ as a single node,[4] with incoming rules from v and $out_2(v)$, but just outgoing rules from $out_2(v)$. This merging concept can be iterated: when v and $out_2(v)$ are updated, both nodes get merged into $out_2(out_2(v))$. We refer to Fig. 9 for further examples of node merging.

As we will see, while the initial network configuration consists of two paths, in later rounds, the already updated solid edges may no longer form a line from left to right, but rather an arbitrary directed tree, with tree edges directed towards the destination d; due to the node merging, the in-degree (from the solid edges) may also increase, while the out-degree and in-degree from the dashed edges remains one. We will use the terms *forward* and *backward* also in the context of the tree: they are defined with respect to the direction of the tree root. However, there also emerges a third kind of edges: *horizontal edges* in-between two different branches of the tree. Moreover, note that while the destination d will always be the root of the tree, the source s does not necessarily have to be at the leaf all the time (due to merging).

The proposed algorithm *Peacock*[5] is based on repeated node merging, and hence *tree shrinking*: starting from the line, it constructs various trees of decreasing sizes, until only a single node is left. At this point, the update is complete and the

[4]When eventually updating from $out_1(out_2(v))$ to $out_2(out_2(v))$ in some round t, we denote this by adding $out_2(v)$ to U_t.

[5]The name of the algorithm is due to its branch resp. "feather" spreading strategy.

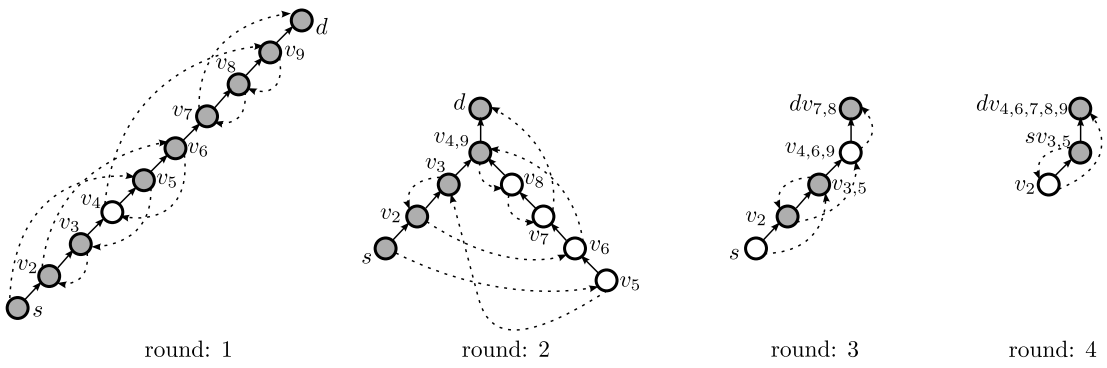


Fig. 9. Example execution of *Peacock*. Updated nodes are shown in white. The initial network is a line (on the *left*). An update of the node with the largest distance v_4 and the merging of v_4 and v_9 leads to a tree shown for round 2, where the node from the merger is denoted by $v_{4,9}$ for re-traceability. Observe that the incoming rules of $v_{4,9}$ are from v_4 and v_9, while the outgoing rules are from v_9. Next, in round 2, the nodes $v_5 - v_8$ can be updated since they are not on the $s-d$ path. E.g., v_7 merges with v_8 into $v_{7,8}$, which in turn gets merged with d into $dv_{7,8}$. This results in a line again, shown for round 3. In round 4, v_2 will be updated and merged into $dv_{2,3,6,7,8,9}$, before the last node, which will be updated in round 5, resulting in the single node $sv_{2,3,4,5,6,7,8,9}$.

Algorithm 1 *Peacock*

Input: initial network G, set of to-be-updated nodes U
Output: relaxed loop-free schedule $(U_1, U_2, \ldots, U_k)$

$t \leftarrow 0$, for all t: $U_t \leftarrow \varnothing$
while (G contains more than one node) **do**
  $t++$
  $X \leftarrow U \setminus U_{<t}$
  if (t *odd*) **then**
    sort dashed forward edges in $out_2(X)$
    for $u \in X$, starting with max forward distance **do**
      if ($\nexists v \in U_t$ s.t. $(v < u < out_2(v)) \vee (v < out_2(u) < out_2(v))$) **then**
        add u to U_t
    for $v \in U_t$ **do**
      merge v with $out_2(v)$ in G
  else
    add to U_t all nodes not on the path from s to d
    for $v \in U_t$ **do**
      merge v with $out_2(v)$ in G
return $(U_1, U_2, \ldots, U_t)$

algorithm terminates. As we will see, *Peacock* manages to decrease the remaining network size by at least a constant factor, for each *pair of consecutive rounds*, resulting in the $O(\log n)$-round upper bound.

Concretely, *Peacock* toggles between two simple strategies:

1) **Shortcut:** In odd rounds (i.e., in the 1^{st}, 3^{rd}, etc. round), *Peacock* tries to reduce the distance between source s and destination d as much as possible, by updating a disjoint set of "far-reaching" (dashed) forward edges: we define the *distance* of a dashed edge as the number of solid edges it skips on the current path from s to d. The idea is that by updating these far-reaching edges, we obtain a tree with many branches (of which only one contains the s-d path).
2) **Prune (and re-establish line):** In the even rounds (i.e., in the 2^{nd}, 4^{th}, etc. round), *Peacock* updates all nodes which are not on the current path from s to d. Since in the preceding odd round we shortened the length of the path from s to d, we can now update a significant number of nodes (namely a constant fraction of the still to-be-updated ones), and due to the subsequent merging operation, the resulting network size is significantly reduced. Intriguingly, the even round, after pruning and merging nodes, will always result in a simple line network again. Based on this line, we can easily determine the next set of far-reaching updatable edges again, enabling a subsequent "productive" even round. Furthermore, as *Peacock* re-establishes the line in every other round, packets disconnected from the destination in the odd rounds will loop for at most one round at a time.

Algorithm 1 gives the formal listing for *Peacock* and Fig. 9 illustrates an example. In the first round there is only one node (v_4) updated. *Peacock* is in the **Shortcut** phase and updates the "far reaching" edges. Once it adds node v_4 there is no other dashed forward edge remaining which is not interfering with the update of v_4. Hence *Peacock* switches to the **Prune** phase in round 2 and updates every node which is not on the $s-d$ path (v_5, v_6, v_7, v_8). *Peacock* then uses the **Shortcut** strategy again in round 3.

Theorem 3: Peacock solves any problem instance in at most $\lceil 6 \log n \rceil$ rounds.

We will make use of two helper lemmas, one targeting odd rounds (the extent to which the distance from s to d can be shortened) and one targeting even rounds (the number of nodes which can be pruned to produce a smaller resulting tree). We will see that after each pair of a consecutive odd and even round, only a constant fraction of nodes is left due to merging.

Lemma 2: In each odd round, *Peacock* reduces the number of nodes on the solid path from s to d by $n_t/3$, where n_t is the number of nodes on the path.

Proof: Peacock orders the nodes in decreasing order of distance, i.e., the number of solid edges they bridge. Including a node v (and its dashed edge), may block other nodes (resp. their intervals) from being scheduled in this round. However, due to the descending distance order, the set of blocked dashed edges span at most twice the distance from v to $out_2(v)$ on the current path: since we choose a maximal distance edge (say of

distance x), edges entering or exiting the corresponding interval may block at most an additional distance of $2x$. Assuming that these distances cannot be covered by any other updates, *Peacock* loses at most twice the distance which it covered. This leaves, in the worst case, at most $2n_t/3$ nodes on the path from s to d. ∎

Lemma 3: Peacock can simultaneously update all nodes which are not on the path from s to d. The subsequent merge operation, re-establishes the line topology.

Proof: First, we observe that by updating these nodes, we cannot introduce any loop, since we do not touch any outgoing dashed edges. Dashed edges, at any time, must form a simple path. Each branch which is currently not on the s-d path will therefore point with at least one new rule to the s-d branch. All nodes of the branch can hence be merged with the respective nodes of the new rules on the s-d branch: a line topology. Also note that the source s does not necessarily have to be at the leaf of a tree. But also in this case, it is possible to update everything on the branch below s. Imagine a node u' which is not on the (solid $s-d$) path. Due to *node merging*, this node will be merged with $out(u')$, which itself is now either part of the $s-d$ path, or will be updated together with another node. Thus, we will successively merge nodes until a node (necessarily) lies on the $s-d$ path and will not be updated. This leads to a line with s as a leaf. ∎

We can now prove Theorem 3:

Proof: Lemma 2 shows that *Peacock* reduces the number of nodes on the s-d path by $n_r/3$ if the underlying network is a line. All of these nodes are not part of the s-d path in the next round, and on different branches. This shows that an update of these nodes is possible in even rounds without introducing a (relaxed) loop. Since, according to Lemma 3, an update of every node but those on the s-d path leads to a line again, we have shown that the number of remaining nodes is reduced by a third every second round. Hence, as $n \cdot (2/3)^{3\log n} \leq 1$ for $n \geq 1$, $\lceil 2 \cdot 3 \log n \rceil$ rounds always suffice. ∎

C. A Matching Lower Bound

The above analysis is asymptotically tight:

Theorem 4: Peacock requires $\Omega(\log n)$ rounds in the worst case.

Proof: We prove the theorem by constructing problem instances for which *Peacock* requires many rounds. Concretely, we construct graphs $G_j, j \in \mathbb{N}$ with $8 \cdot 2^j$ nodes: to update a given G_j, *Peacock* requires $2j+5$ rounds.

In the constructed instance, when there are multiple forward edges which cover the same distance, we will fix the one to be picked by *Peacock*. This simplifies the presentation and comes without loss of generality.

We first present the general idea of the construction, before describing the technical details. By executing *Peacock* for two rounds (i.e., shortcut and prune), we reduce graph G_j into a graph isomorphic to G_{j-1}, requiring $2(j-1)+5$ rounds to complete. As $|V(G_j)| \in \Theta(2^j)$, the number of rounds used by *Peacock* is in $\Theta(\log n)$. An illustration is given in Fig. 10, starting with the 16-node graph G_1.

Let us now elaborate on the details of this construction.

Construction of G_j: For all $3 \leq i \leq 2^j+2$, we create the 8 nodes

$$v_{\frac{-1+2^i}{2^i}-7/8}, v_{\frac{-1+2^i}{2^i}-6/8}, \ldots, v_{\frac{-1+2^i}{2^i}-1/8}, v_{\frac{-1+2^i}{2^i}-0/8}$$

resulting in $8 \cdot 2^j$ nodes. E.g., for $j=0$, nodes $v_0, v_{1/8}, \ldots, v_{7/8}$ are created. The *old* edges are created by ordering the nodes by their index, defining an $s-d$ path with $-1+8 \cdot 2^j$ edges, with the node v_0 representing s and the node $v_{(-1+2^{2^j+2})/2^{2^j+2}}$ representing d. The *new* edges consist of $1/2 \cdot 8 \cdot 2^j$ forward edges and of $-1+1/2 \cdot 8 \cdot 2^j$ backward edges. All forward edges cover the same distance, namely half of the nodes, e.g., $(s = v_0, v_{4/8})$, $(v_{7/32}, v_{23/32})$, or

$$\left(v_{\frac{-1+2^{2^j+2}}{2^{2^j+2}}-4/8}, v_{\frac{-1+2^{2^j+2}}{2^{2^j+2}}} = d\right)$$

The construction of the backward edges is as follows, for every $3 \leq i \leq 2^j+2$, except for d:

$$\left(v_{\frac{-1+2^i}{2^i}-3/8}, v_{\frac{-1+2^i}{2^i}-5/8}\right), \left(v_{\frac{-1+2^i}{2^i}-1/8}, v_{\frac{-1+2^i}{2^i}-6/8}\right),$$
$$\left(v_{\frac{-1+2^i}{2^i}-2/8}, v_{\frac{-1+2^i}{2^i}-4/8}\right), \left(v_{\frac{-1+2^i}{2^i}-0/8}, v_{\frac{-1+2^{i+1}}{2^{i+1}}-7/8}\right)$$

i.e., $-1+1/2 \cdot 8 \cdot 2^j$ backward edges. The $8 \cdot 2^j$ nodes are created in 2^j layers, denoted by the index $3, 4, \ldots 2^j+2$, with s being in layer 3 and d being in layer 2^j+2. The new forward and backward edges thus form an $s-d$ path as follows: First, layer 3 is traversed, then layer 4, then 5, ..., until lastly, layer 2^j+2 is traversed, ending at $d = v_{(-1+2^{2^j+2})/2^{2^j+2}}$.

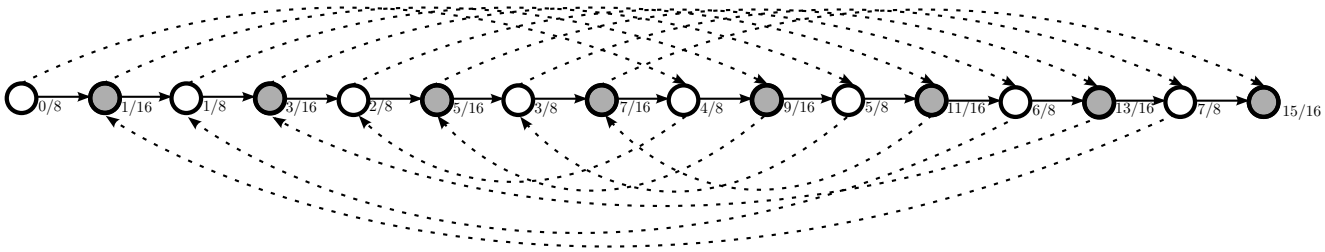
To prove the logarithmic runtime, we proceed by induction. **Base case**: For G_0 with 8 nodes, *Peacock* will in the first round update the edge $(v_0, v_{4/8})$ (shortcut), then prune nodes, resulting in the 4 nodes $v_{0,4/8}, v_{1/8,5/8}, v_{2/8,6/8}, v_{3/8,7/8}$. Next, $(v_{0,1/4}, v_{2/4,6/8})$ is chosen as the shortcut, where pruning results in the two nodes $v_{0,4/8,2/8,6/8}, v_{1/8,5/8,3/8,7/8}$. The last shortcut is the only forward edge left, with no more pruning needed. As such, *Peacock* uses $2j+5 = 5$ rounds to update G_0, cf. Fig. 10.

Inductive step: We next show that one iteration of *Peacock* turns G_{j+1} into a graph (isomorphic) to G_j. Let $(v_0, v_{4/8})$ be the shortcut edge. Then, in the pruning phase, all nodes with an index $k < 4/8$ will be updated, merging with $v_{k+4/8}$, with $s = v_0$ being merged with $v_{4/8}$ already in the shortcut update. The old edges between $v_{4/8}$ and d in G_{j+1} thus form all the old edges in G_j.

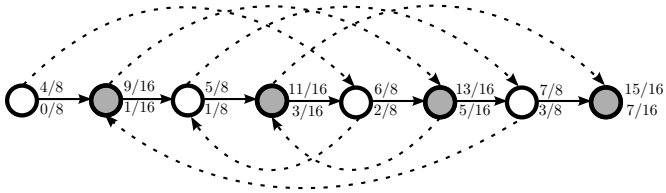
The correctness of the proof now relies on the technical construction of the backward edges in G_{j+1}, as they will be responsible for all new edges in G_j. The idea is as follows: G_{j+1} has $2^{j+1} = 2 \cdot 2^j$ layers, twice as many as G_j with 2^j layers $3, 4, \ldots, 2^j+2$. The layers 3 and 4 from G_{j+1} with 8+8 nodes will form the 8 nodes in layer 3 in G_j, layers 5 and 6 from G_{j+1} will form layer 4 in G_j, etc., and layers $2^{j+1}+1$ and $2^{j+1}+2$ from G_{j+1} will form the layer 2^j+2 in G_j.

Towards this end, consider any $3 < i < 2^j+2$: The 16 nodes from layers $2(i-2)+1$ and $2(i-2)+2$ in G_{j+1} have to be pruned into layer i in G_j. We start with layer $2(i-2)+1$ in G_{j+1}, consisting of the 8 nodes

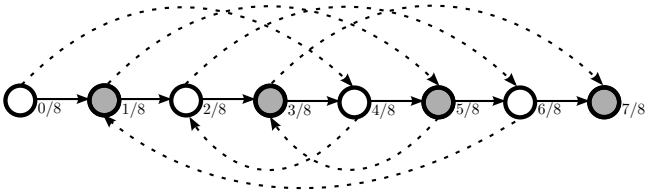
$$v_{\frac{-1+2^{2(i-2)+1}}{2^{2(i-2)+1}}-7/8}, \ldots, v_{\frac{-1+2^{2(i-2)+1}}{2^{2(i-2)+1}}-0/8}$$



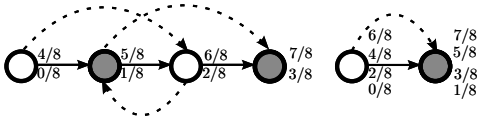
(a) The graph G_1 with 16 nodes. When *Peacock* selects the edge from $0/8$ to $4/8$ as a shortcut, pruning results in the graph in Fig. 10b.



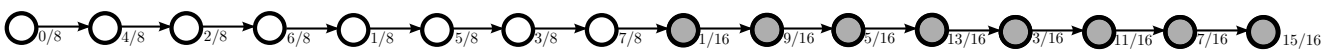
(b) After two rounds with *Peacock*, isomorphic to G_0 in Fig. 10c.



(c) The graph G_0 with 8 nodes. $0/8$ to $4/8$ is the next shortcut.



(d) To the left, the output of *Peacock* on G_0 after two rounds. To the right, after two more rounds, selecting the first forward edge as a shortcut each time.



(e) The resulting updated graph, expanded into 16 nodes again.

Fig. 10. Example of the operation of *Peacock* on G_1. The v is omitted in node names for better readability. Starting on G_1 in Fig. 10a, *Peacock* performs two updates (shortcut+prune), resulting in the graph in Fig. 10b – which is isomorphic to the graph G_0 in Fig. 10c. Then, five further updates are needed: Resulting, first in the left graph in Fig. 10d, second, in the right one in the same figure, and third, the completed update schedule, expanded in Fig. 10e.

which are pruned to just 4 nodes

$$v_{\frac{-1+2^{2(i-2)+1}}{2^{2(i-2)+1}}-7/8,\cdots-3/8},\ldots,v_{\frac{-1+2^{2(i-2)+1}}{2^{2(i-2)+1}}-4/8,\cdots-0/8}.$$

The same happens at layer $2(i-2)+2$ in G_{j+1}, resulting in the 4 nodes

$$v_{\frac{-1+2^{2(i-2)+2}}{2^{2(i-2)+2}}-7/8,\cdots-3/8},\ldots,v_{\frac{-1+2^{2(i-2)+2}}{2^{2(i-2)+2}}-4/8,\cdots-0/8}$$

We now iterate through all corresponding 8 backward egdes in G_{j+1}, the 4 from layer $2(i-2)+1$ and the 4 from layer $2(i-2)+2$. For ease of readability, we will refer to nodes just by their (combined) ending index, e.g., $-1/8$ or $-2/8, -6/8$, as long as the layer is clear from the context. Note that in layer 3 in G_j, the node $-7/8$ (representing s) will have no incoming edges, and in the highest layer 2^j+2, the node $-0/8$ (representing d) will have no outgoing edges.

1) **Layer** $2(i-2)+1$: The node with index $-1/8$ points to $-6/8$, resulting in the new backward edge from the index $-5/8, -1/8$ to $-6/8, -2/8$, which we identify as from $-3/8$ to $-5/8$ (backward) in layer i in G_j. Similarly:
   a) $-2/8 \rightarrow -4/8 \Rightarrow -2/8, -6/8 \rightarrow -4/8, -0/8 \Rightarrow -5/8 \rightarrow -1/8$ (forward) in layer i in G_j.
   b) $-3/8 \rightarrow -5/8 \Rightarrow -3/8, -7/8 \rightarrow -5/8, -1/8 \Rightarrow -7/8 \rightarrow -3/8$ (forward) in layer i in G_j.
   c) $-0/8 \rightarrow -7/8$ $(2(i-2)+2) \Rightarrow -0/8, -4/8 \rightarrow -7/8, -3/8$ $(2(i-2)+2) \Rightarrow -1/8 \rightarrow -6/8$ (backward) in layer i in G_j.
2) **Layer** $2(i-2)+2$: Using analogous arguments, we identify the resulting backward edge from index $-5/8, -1/8$ to $-6/8, -2/8$ as from $-2/8$ to $-4/8$ (backward) in layer i in G_j. Again, similarly:
   a) $-3/8, -7/8 \rightarrow -5/8, -1/8 \Rightarrow -6/8 \rightarrow -2/8$ (forward) in layer i in G_j.
   b) $-2/8, -6/8 \rightarrow -4/8, -0/8 \Rightarrow -4/8 \rightarrow -0/8$ (forward) in layer i in G_j.
   c) $-0/8, -4/8 \rightarrow -7/8, -3/8$ $(2(i-2)+3) \Rightarrow -0/8$ $(i) \rightarrow -7/8$ $(i+1)$ (backward) in G_j, does not exist for $i = 2^j+2$ in G_j.

   As *Peacock* leaves the old edges beyond the index $-3/8$ intact, we have shown the inductive step to be correct, i.e., one iteration of *Peacock* turns G_{j+1} into G_j.

It thus follows that *Peacock* needs $2j+5$ rounds to finish updating G_j. As G_j has $n = 8 \cdot 2^j$ nodes, *Peacock* needs $\Theta(\log n)$ rounds on G_j, which concludes the proof. ■

V. Simulation Study

We established so far that strong and relaxed loop-freedom differ heavily in their worst case behavior, but are identical for extremely short schedules. Specifically, recall that there are instances where strong loop-freedom requires $\Omega(n)$ rounds, but relaxed loop-freedom always takes at most $\Theta(\log n)$ rounds. On the other hand, the instances solvable in two rounds are exactly the same for both strong and relaxed loop-freedom.

In order to complement our formal analysis and evaluate the number of update rounds required in "on average" and in more realistic settings, we conducted a series of experiments, in different settings. In particular, we also implemented a Mixed

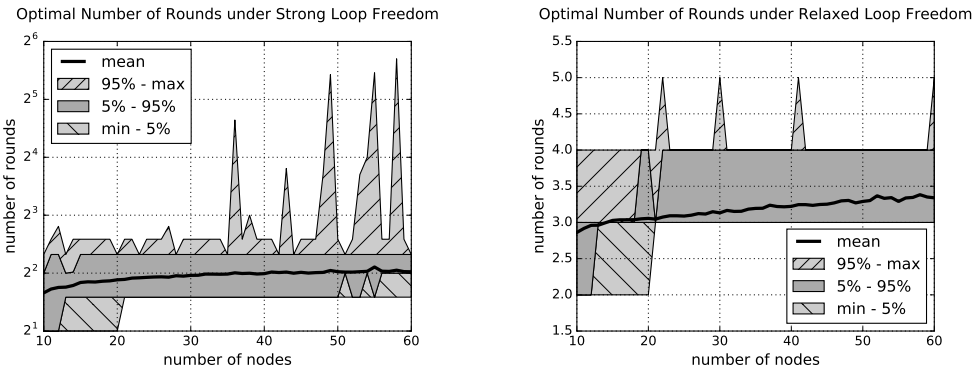


Fig. 11. Plots of the experiments conducted to compare SLF (left) and RLF (right), with about 500 instances per number of nodes, ranging from 10 to 60 (x-axis). The number of rounds is displayed on the y-axis, but note that the scale is *logarithmic* for the left SLF. We depict the mean, bottom 5%, 5% to 95%, and the top 5% in terms of numbers of rounds needed. On average, RLF is roughly between a half and a full round faster than SLF, but starting at around 35 nodes, SLF heavily deviates in the top 5%, taking over 10 times more rounds in extreme cases. For the bottom 5%, both SLF and RLF behave similarly.

Integer Program (MIP) in order to have an optimal baseline to which we can compare the quality of the produced update schedules. As the runtime of the mixed integer program is high in large networks, in the following, we focus on networks of up to size 60. To enable other researchers to reproduce our results, we release the code of our implementation together with this paper.

We now first describe the experimental methodology in Section V-A, followed by our results and discussion in Section V-B, where we also briefly study optimal solutions for the *Peacock* lower bound construction of the last section.

A. Comparison Methodology

In order to compare the strong and relaxed loop-freedom, we consider an optimal baseline, adapting the Mixed Integer Program presented in [12] by removing waypoints and not considering any flow extensions.

We generate random instances ranging from 10 to 60 nodes by permuting the order of the intermediate nodes in the new routes uniformly at random, as in [12], not permitting new rules of the type (v_i, v_{i+1}). On each number of nodes 500 experiments are performed, $25{,}500$ in total. Each experiment is allocated 30 minutes of computing time on a single thread of an Intel Xeon E5-4627 v3 at 2.60GHz, with 6GB of memory per experiment. We use Gurobi 7.5.1 [13] to solve the MIP formulations. In total, less than ten experiments did not finish in the time limits and were terminated.

B. Comparison Results

We present our experiment results in Figure 11. As can be seen in the lower left corner of both plots, the 2-round instances are identical, as expected. Until around 35 nodes, RLF is around one round faster in average than SLF. This speed-up matches the experiments (*with* waypoint-enforcement) performed in [12]. Beyond 35 nodes however, SLF experiences peaks in the number of rounds required, taking up to roughly 10 times longer than RLF in the top 5% of experiments performed.

Summarizing, in over $25{,}000$ experiments, RLF and SLF did not deviate much in average, but the tail heavily differs: while RLF peaked at just 5 rounds, SLF took up to around 50 rounds. As the duration of the update process "*determines the agility of the control loop*" [1], SLF can thus cause heavy delays for the network operation, whereas the performance of RLF is dependable. Furthermore, as pointed out in Section II-B, the downside of some packets entering loops in RLF can be negated in practice.

Lastly, we also evaluated instances of the lower bound graphs G_j from Section IV-C. Optimal relaxed loop-free schedules for networks of size 8 require 3 rounds, for size 16 to 256, 4 rounds, and for networks of size 512, 7 rounds. Thus, we conclude that while the problems can in principle be solved faster than with Peacock, no algorithm exists to compute very short schedules for these instances.

VI. Related Work

Although some of our insights are of more general nature, our work is mainly motivated by the SDN paradigm, and especially its traffic engineering flexibilities and its support for a programmatic, dynamic, yet formally verifiable network management [14]. Indeed, a more flexible traffic engineering, that is, selection of forwarding routes, is considered one of the main motivations for SDN, and has been studied intensively over the last years. [6], [15]. Our paper is orthogonal to this line of research, in the sense that in our model, the routes are *given* and can be arbitrary.

The problem of updating [1], [2], [3], [16], [17], [18], synthesizing [19] and checking [20] policies [21] as well as routes [22] has also been studied intensively. In their seminal work, Reitblatt et al. [3] initiated the study of network updates providing strong, per-packet consistency guarantees, and the authors also presented a 2-phase commit protocol. The paper sparked much research, and the protocol also forms the basis of the distributed control plane implementation [16]. For an overview of the network update literature and a discussion of how the problem in the SDN context related to similar problems in traditional networks, we refer the reader to the recent survey [5].

Our work is also motivated by the measurement studies in [1] and [23] providing empirical evidence for the non-negligible time and high variance of switch updates, as well as by the work

by Mahajan and Wattenhofer [2], later extended in [8], which started investigating weaker transient consistency properties—in particular also (strong) loop-freedom—for destination-based routing policies (DLF). This line of research focuses on the problem of maximizing the number of links which can be updated simultaneously at a *given* moment in time. This problem is known to be NP-hard as well [9], both for SLF and RLF [24], but there also exist approximation algorithms and optimal algorithms for special instances [24].

Our model is different in two main respects:

1) *Objective function:* Rather than maximizing the number of links which can be updated concurrently [2], we study the natural problem of computing *short* schedules: schedules which minimize the number of controller interactions. We have shown in this paper that a greedy update of links can significantly delay a route update.
2) *Arbitrary paths:* In contrast to prior work on loop-free updates [2], our model is not limited to destination-based routing. Rather, routes from a source s to a destination d can be arbitrary paths. Given the traffic engineering flexibilities introduced by SDNs, this is an important use case.

Despite these differences, interestingly, some of the results presented in this paper also provide new insights for the DLF model. In particular, our problem instances can be transformed into destination-based instances (on special graphs), and hence our NP-hardness proofs (in particular Theorem 1) also apply in the model by Wattenhofer et al. Conversely, all algorithms (maximizing the number of links) in the DLF-model are also consistent in SLF and RLF. It is however an open problem if finding a $k = 2$-round update schedule is NP-hard in DLF. However, analogously to the approach presented in this paper, we can argue that any $k = 2$ instance for DLF may not contain any **BB** nodes, due to symmetry properties. However, it is not clear if **BB**s completely characterize $k = 2$ instances in DLF, as further so-called horizontal edges can be present.

Besides these closely related works, there exists a number of additional results in the field. Inspired by [25], a local proof-labeling scheme for DLF was presented in [26], where nodes decide to update based on their direct neighborhood. This way, the controller needs to send out only a single update, with the nodes themselves updating asynchronously, but still in a provably consistent fashion. The work in [26] draws upon ideas of François et al. [27], [28], who studied transient loops in link-state routing protocols.

Wattenhofer et al. [2], [8] showed that multi-destination-based rules (without considering the source) can be handled by temporary splitting and joining them afterwards. Vanbever et al. [29] showed in this context that if a router has to update all its rules for a linear number of destinations at once, finding a strongly loop-free router ordering is NP-hard. Furthermore, already for two destinations without rule-splitting, sublinear scheduling of updates in DLF is NP-hard as well [9].

Dudycz et al. [30] considered the joint optimization of multiple routes, aiming to minimize the number of times a router has to be updated via the controller. The authors showed how to optimally combine consistent update schedules, for example from [2], [8] or from our work.

Researchers have also started investigating consistent updates for networks which include (network function virtualized) middleboxes [31]. Ludwig et al. [18] presented update protocols which maintain security critical properties such as waypoint enforcement via a firewall, in a transiently consistent manner; the authors also showed that the loop-freedom and waypoint enforcement properties may even conflict, deciding if both are possible is NP-hard to decide [12]. Vissicchio and Cittadini [32] propose to jointly use update ordering and packet tagging in this context. Further ideas can be found in [33]. Cerný et al. [34] investigate when update ordering without tagging allows for per-packet consistency, i.e., every packet may only use the old or new route.

Considering congestion-freedom adds another layer of complexity, a hierarchy first explored in [2], [8]: Using a 2-phase commit and tagging, deciding if a congestion-free update exists can be performed in polynomial time for splittable flows [35], or is even always possible in anycast scenarios [36], but is NP-hard for unsplittable flows [9], [35]. If just node ordering is allowed, the latter problem becomes already NP-hard for just two flows [37].

A further standpoint is promoted by Ghorbani and Godfrey in their work [38]: the authors argued that in the context of network function virtualization, not weaker but rather stronger consistency properties are required.

Bibliographic note. Preliminary versions of this paper were presented at ACM HotNets 2014 [18] (introducing the round-based model) as well as at ACM PODC 2015 [39].

VII. Conclusion

This paper initiated the study of fast and loop-free network update algorithms requiring a minimum number of update rounds. We have shown that the existing, strict definitions of transient loop-freedom are problematic, as short schedules are hard to compute and may not even exist. We have proposed a weaker notion of loop-freedom which we believe expresses the actually required consistency.

We believe that our work opens interesting directions for future research. Most importantly, it would be interesting to derive $\omega(1)$-round lower bounds, or to show that $O(1)$-round schedules for relaxed loop-free problems always exist. Our computational experiments (using mixed integer programs) indicate that larger problem instances require more rounds. So far, the worst problem instance (consisting of 512 nodes, based on our lower bound topology in the proof of Theorem 4) requires 7 rounds.

Another important direction for future research regards the development of benchmarks and rigorous methodologies to evaluate and compare the performance of different network update algorithms. While we in this paper have focused on random as well as "hard" update instances, as SDN is moving intro production, it will be interesting to better understand the types of route updates arising in emerging SDN deployments.

For reproducibility purposes and to ease comparisons in future works, we made the code of our evaluation publicly available to the community at https://github.com/MatthiasRost/NetworkUpdateScheduler.

Acknowledgments. We would like to thank Matthias Rost for helping with the experimental evaluation. Klaus-Tycho Foerster is supported by the Danish Villum Foundation.

REFERENCES

[1] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang, "Dionysus: Dynamic Scheduling of Network Updates," in *Proc. SIGCOMM*, August 2014.
[2] R. Mahajan and R. Wattenhofer, "On Consistent Updates in Software Defined Networks," in *Proc. HotNets*, 2013.
[3] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, 2012.
[4] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. HotSDN*, 2013.
[5] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent network updates," *CoRR*, vol. abs/1609.02305, 2016.
[6] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, pp. 20:20–20:40, 2013.
[7] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proc. SIGCOMM*, 2013.
[8] K.-T. Foerster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *Proc. Networking*, 2016.
[9] K.-T. Foerster and R. Wattenhofer, "The power of two in consistent network updates: Hard loop freedom, easy flow migration," in *Proc. ICCCN*, 2016.
[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[11] C. A. Tovey, "A simplified np-complete satisfiability problem," *Discrete Applied Mathematics*, vol. 8, no. 1, pp. 85–89, 1984.
[12] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," in *Proc. SIGMETRICS*, 2016.
[13] Gurobi optimizer 7.5.1. http://www.gurobi.com/.
[14] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4d approach to network control and management," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 5, pp. 41–54, 2005.
[15] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "Sdx: A software defined internet exchange," in *Proc. SIGCOMM*, 2014.
[16] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust sdn control plane for transactional network updates," in *Proc. INFOCOM*, 2015.
[17] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, "zUpdate: Updating Data Center Networks with Zero Loss," in *Proc. SIGCOMM*, August 2013.
[18] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proc. HotNets*, 2014.
[19] J. McClurg, H. Hojjat, P. Cerny, and N. Foster, "Efficient synthesis of network updates," in *Proc. PLDI*, 2015.
[20] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. NSDI*, 2013.
[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software Defined Networks," in *Proc. NSDI*, 2013.
[22] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proc. ACM SIGCOMM*, 2007.
[23] M. Kuzniar, P. Peresini, and D. Kostic, "What you need to know about sdn flow tables," in *Proc. PAM*, 2015.
[24] S. A. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid, "Transiently consistent SDN updates: Being greedy is hard," in *Proc. SIROCCO*, 2016.
[25] S. Schmid and J. Suomela, "Exploiting locality in distributed SDN control," in *Proc. HotSDN*, 2013.
[26] K.-T. Foerster, T. Luedi, J. Seidel, and R. Wattenhofer, "Local checkability, no strings attached: (a)cyclicity, reachability, loop free updates in SDNs," *Theoretical Computer Science*, 2017 (to appear).
[27] P. François, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving subsecond IGP convergence in large IP networks," *Computer Communication Review*, vol. 35, no. 3, pp. 35–44, 2005.
[28] P. François and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Trans. Netw.*, vol. 15, no. 6, pp. 1280–1292, 2007.
[29] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, and O. Bonaventure, "Lossless migrations of link-state igps," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1842–1855, 2012.
[30] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *Proc. DSN*, 2016.
[31] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proc. NSDI*, 2014.
[32] S. Vissicchio and L. Cittadini, "FLIP the (flow) table: Fast lightweight policy-preserving SDN updates," in *Proc. INFOCOM*, 2016.
[33] S. Vissicchio, L. Vanbever, L. Cittadini, G. G. Xie, and O. Bonaventure, "Safe update of hybrid SDN networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1649–1662, 2017.
[34] P. Cerný, N. Foster, N. Jagnik, and J. McClurg, "Optimal consistent network updates in polynomial time," in *Proc. DISC*, 2016.
[35] S. Brandt, K.-T. Foerster, and R. Wattenhofer, "On consistent migration of flows in sdns," in *Proc. INFOCOM*, 2016.
[36] ——, "Augmenting flows for the consistent migration of multi-commodity single-destination flows in sdns," *Pervasive and Mobile Computing*, vol. 36, pp. 134–150, 2017.
[37] S. A. Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht, "Congestion-free rerouting of flows on dags," *CoRR*, vol. abs/1611.09296, 2016.
[38] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proc. HotSDN*, 2014.
[39] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *PODC*, 2015.

Klaus-Tycho Foerster is a Postdoctoral Researcher at Aalborg University, Denmark. He received his Diplomas in Mathematics (2007) & Computer Science (2011) from Braunschweig University of Technology and his PhD degree (2016) from ETH Zurich, Switzerland, advised by Roger Wattenhofer. He spent autumn 2016 as a Visiting Researcher at Microsoft Research Redmond with Ratul Mahajan, before joining Stefan Schmid in 2017. His research interests revolve around algorithms and complexity in the areas of networking and distributed computing.

Arne Ludwig received his Diploma and PhD in Computer Science from the Technical University of Berlin, Germany in 2011 and 2016. He is currently working at the SAP Innovation Center Potsdam. His research interests include (Online-)Algorithms, SDN, Network Virtualization and Network Economics.

Jan Marcinkowski is a PhD student at the University of Wrocław, Poland. He received his MSc degree in Computer Science from the same university (2016). He spent autumn 2014 in TU Berlin, as a guest of Stefan Schmid's, working on SDN updates. In 2015-2017 he lived in Zurich, Switzerland where he worked as a Software Engineer at Google.

Stefan Schmid is Professor at the Faculty of Computer Science at the University of Vienna, Austria. He received his MSc (2004) and PhD degrees (2008) from ETH Zurich, Switzerland. In 2009, Stefan Schmid was a postdoc at TU Munich and the University of Paderborn, between 2009 and 2015, a senior research scientist at the Telekom Innovations Laboratories (T-Labs) in Berlin, Germany, and from the end of 2015 till early 2018, an Associate Professor at Aalborg University, Denmark. His research interests revolve around fundamental and algorithmic problems arising in networked and distributed systems.