Scheduling Loop-free Network Updates: It's Good to Relax!

Arne Ludwig[1], Jan Marcinkowski[2], Stefan Schmid[1,3]
[1] TU Berlin, Germany
[2] Institute of Computer Science, University of Wrocław, Poland
[3] Telekom Innovation Laboratories (T-Labs), Germany

ABSTRACT

We consider the problem of updating arbitrary routes in a software-defined network in a (transiently) loop-free manner. We are interested in fast network updates, i.e., in 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 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 prove that $O(\log n)$-round relaxed loop-free schedules always exist, and can also be computed efficiently.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; G.2.2 [**Graph Theory**]: Network Problems

General Terms

Algorithms

Keywords

Software-Defined Networking; Graph Algorithms; Scheduling; NP-hardness

1. INTRODUCTION

Computer networks are currently undergoing a phase transition. The paradigm of Software-Defined Networking (SDN) introduces 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 install arbitrary routes (i.e., forwarding rules) which may not necessarily be shortest paths or based on destination IP addresses only.

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3617-8 /15/07 ...$15.00.
http://dx.doi.org/10.1145/2767386.2767412.

However, exploiting the benefits of a more dynamic and logically centralized network management is non-trivial. A fundamental problem 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 [8], 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* [13].

One possible solution is to use a *2-Phase Commit Protocol (2PC)* and *(packet) tagging* [17]: 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* [17]: 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. [3]

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 schedules the next subset $V_2 \subseteq V$ of nodes to update, until the final subset V_k completes the route update. This protocol does not require packet tagging, and, as has been argued in [13], 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.

1.1 Our Contributions

This paper initiates the study of *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 [4]). We ask: *How many communication rounds k are needed to update a network in a (transiently) loop-free manner?*

We show that answering this question is difficult. In particular, we show that while deciding whether a k-round schedule exists is trivial for $k = 2$, it is already NP-complete for $k = 3$. Moreover, we show that there exist problem instances which require $\Omega(n)$ rounds, where n is the network size. In the Appendix, we will also show that just aiming to "greedily" update a *maximum* number of nodes in each round (as proposed in previous work [13], 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 prove that $O(\log n)$-round relaxed loop-free schedules always exist, and can also be computed efficiently, and we present an elegant algorithm accordingly.

1.2 Organization

The remainder of this paper is organized as follows. Section 2 introduces our formal model. Section 3 studies the strong consistency model for transient loop-freedom, and Section 4 studies relaxed loop-freedom. After reviewing related literature in Section 5, we conclude our work in Section 6.

2. 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 ($U = V$).

2.1 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 of U (i.e., $U = U_1 \uplus U_2 \uplus \ldots \uplus U_k$), 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 $U_{\le t}$, $U_{>t}$ 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*); 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.

2.2 Weak Loop-Freedom

In this paper, we will also propose a weaker notion of loop-freedom: *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. 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. 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.

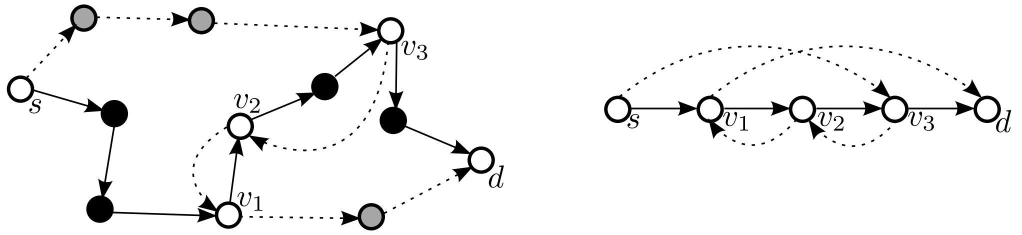

Figure 1: Overview of model and reduction. The network on the *left* is reduced to the line representation on the *right.* The solid lines show the old route r_1 and the dashed lines show the new route r_2. Nodes shown in white are the only ones which are part on both paths, and hence relevant for the problem.

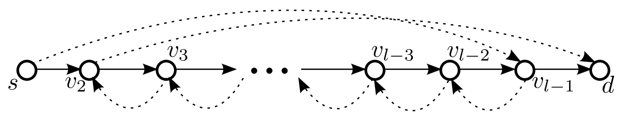

Figure 2: RLF vs SLF: An SLF schedule needs to update backward edges one by one from left to right, requiring $\Omega(n)$ rounds; for RLF, an $O(1)$-round schedule exists.

The difference is that we only care about the subset E'_t of E_t consisting of edges *reachable from the source s.*

Example. To highlight the difference between SLF and RLF, Fig. 2 presents an example where a relaxed 3-round loop-free update schedule exists: in round 1 all forward edges are updated, in round 2 all backward edges except for the last one (v_{l-1}, v_{l-2}) are updated, and in round 3, the last backward edge is updated. In contrast, a strong loop-free schedule needs to go through the backward edges one by one, $v_3, v_4, \cdots, v_{l-1}$: updating v_i before v_{i-1} results in a loop. Thus, $n-1$ rounds are required in this case: a factor $\Omega(n)$ more than RLF. This is worst possible.

3. FAST UPDATES ARE DIFFICULT

How many rounds are needed to update a network in a (strongly) loop-free manner? On the one hand, the problem seems difficult: the problem of breaking cycles even in a single round, is related to the well-known NP-hard Feedback Arc Set Problem. 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 problem instances allowing for 2-round schedules are trivial (Section 3.1), deciding whether 3-round schedules exist is NP-complete (Section 3.2). Also recall our example from Fig. 2 which shows that there exist problem instances which cannot be updated in less than $\Omega(n)$ rounds. In Appendix A, we will additionally prove that applying a greedy strategy which maximizes the number of node updates *in each round*, can lead—already after one round—to an undesirable configuration from which $\Omega(n)$ rounds are needed to complete the update, although the problem was initially $O(1)$-round solvable.

3.1 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. However, we can also classify edges from the other side, from the destination and "looking backward in time": as if we were updating edges from the dashed ("new" r_2) rules to the solid ("old" r_1) ones, starting with the last round. Given this backward 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. Now, it is easy to see that in the last round, we can update any subset of rules which are either **BF** or **FF**, just like in the first round where we can update any **FB** or **FF**. An example can be seen in Fig. 3 on the left.

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.

3.2 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

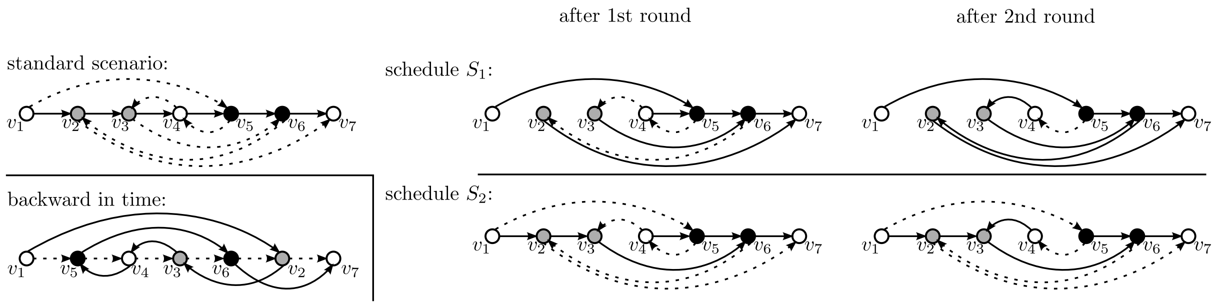

Figure 3: *Left:* "Looking backward in time", an example with reversed update pattern (from *dashed* to *solid* path). We obtain the following classification: v_1 is FF; v_2, v_3 are FB; v_4 is BB and v_5, v_6 are BF. *Right:* Intuition why node updates can be moved from round 2 to round 1 or 3. There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

network update problem (namely, two paths traversing the same set of nodes).

3.2.1 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* 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 S, for $t \in \{1, 2\}$. Since 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. $\square$

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** 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 ($U_1 = \{v_1, v_2, v_3\}, U_2 = \{v_4\}, U_3 = \{v_5, v_6\}$).

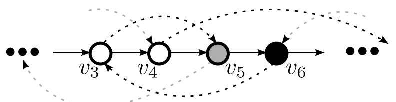

Figure 4: Choosing the right set of FF nodes is important. An update of only v_4 would enable the BB node v_6 to be updated in the second round. An additional update of v_3 would then lead to a loop (note that v_5 will definitely not be updated in the first round).

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**· and cannot be updated in the first round, and hence an update of v_6 would result in a loop ($v_3 \to v_5 \to v_6 \to v_3$).

3.2.2 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.

We create the following clauses:

1. For every variable x, we create variables
$$x_0, x_1, \dots, x_{p_x}, x_l, \overline{x}_0, \overline{x}_1, \dots, \overline{x}_{n_x}, \overline{x}_l,$$
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 $x_1, \dots, x_{p_x}$ and $\overline{x}_1, \dots, \overline{x}_{n_x}$). 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 $i = 0 \dots p_x - 1$ and $(\overline{x}_i \to \overline{x}_{i+1})$ for $i = 0 \dots n_x - 1$; 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 $\mathcal{C}'$.

3.2.3 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 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**· edge, and leftwards when it is a backward **B**· edge.

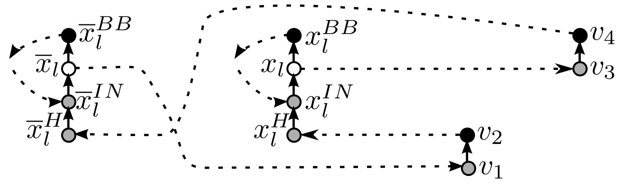

Figure 6: Gadget for exclusive clause. An update of either x_l or $\overline{x}_l$ prevents the other one from being updateable: the BB nodes v_2 and v_4 would form a cycle in the second round.

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*) assures 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 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 assured 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

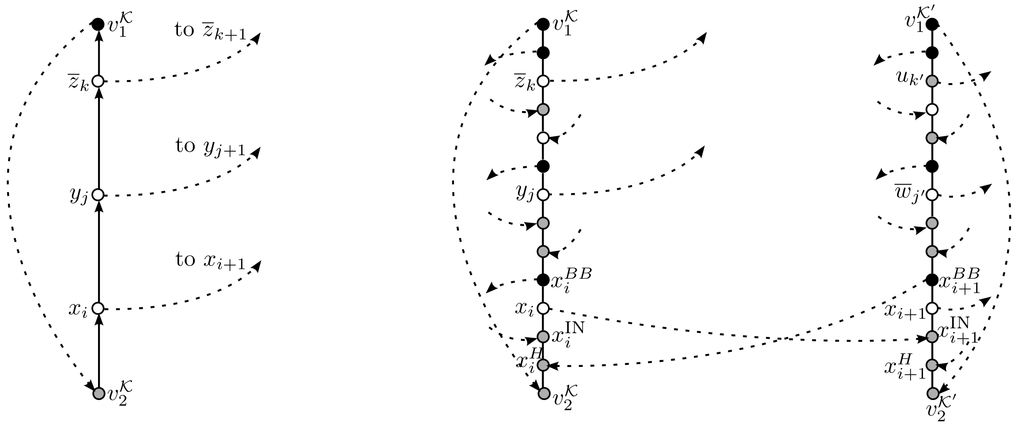

Figure 5: *Left:* Gadget for clause $x_i \vee y_j \vee \overline{z}_k$. At least one node needs to be updated to prevent the loop over $v_1^{\mathcal{K}}$ and $v_2^{\mathcal{K}}$. *Right:* More details about the gadget including also the implication clause $x_i \to x_{i+1}$ representation, and a second clause $x_{i+1} \vee \overline{w}_{j'} \vee u_{k'}$. It is assured 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 assured to be of type B·, to guarantee that they cannot be updated in the first round.

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 assures 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. $\square$

3.2.4 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) 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

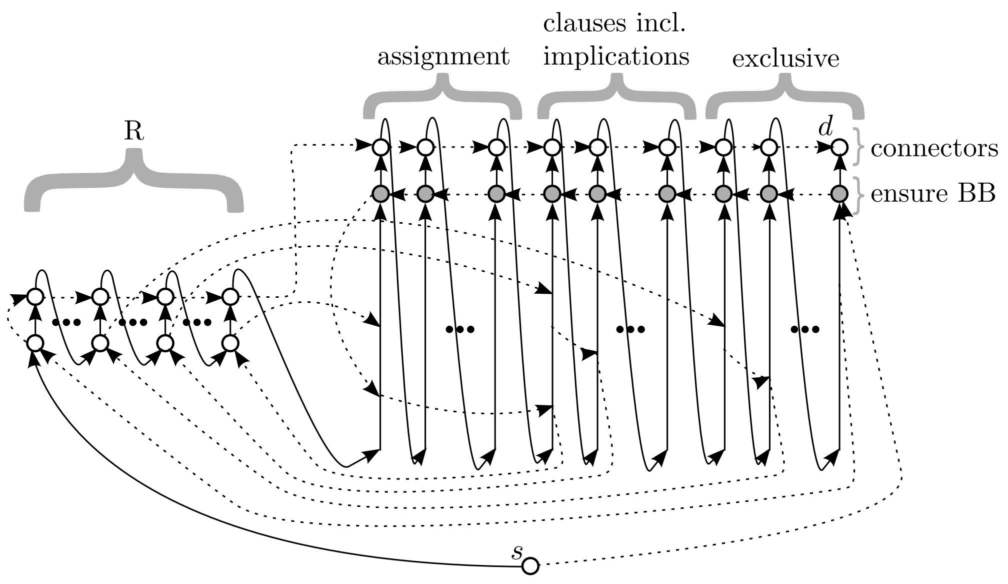

Figure 7: Overview of how the path is connected. The grey nodes are used to connect everything into one solid path. They also join the dashed path at the last nodes. This way, all nodes in R (*white cycles*) are of type FF.

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.

4. 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. (Recall also the example in Fig. 2 which permitted a 3-round solution for RLF while SLF required $n-1$ rounds.) 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.

Before presenting the 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). 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

Algorithm 1 *Peacock*

```
Input: initial network $G_0$, set of to-be-updated nodes $U$
Output: (relaxed) loop-free schedule $(U_1, U_2, \dots, U_k)$
 1: $G \leftarrow G_0$, $t \leftarrow 0$, for all $t$: $U_t \leftarrow \varnothing$
 2: while ($G$ contains more than one node) do
 3:     $t{+}{+}$
 4:     $X \leftarrow U \setminus U_{<t}$
 5:     if ($t$ odd) then
 6:         sort dashed forward edges in $out_2(X)$
 7:         for $u \in X$, starting with max forward distance do
 8:             if ($\nexists v \in U_t$ s.t. $(v < u < out_2(v)) \vee (v < out_2(u) < out_2(v))$) then
 9:                 add $u$ to $U_t$
10:     else
11:         add to $U_t$ all nodes not on the path from $s$ to $d$
12: return $(U_1, U_2, \dots, U_t)$
```

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*[1] 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 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

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

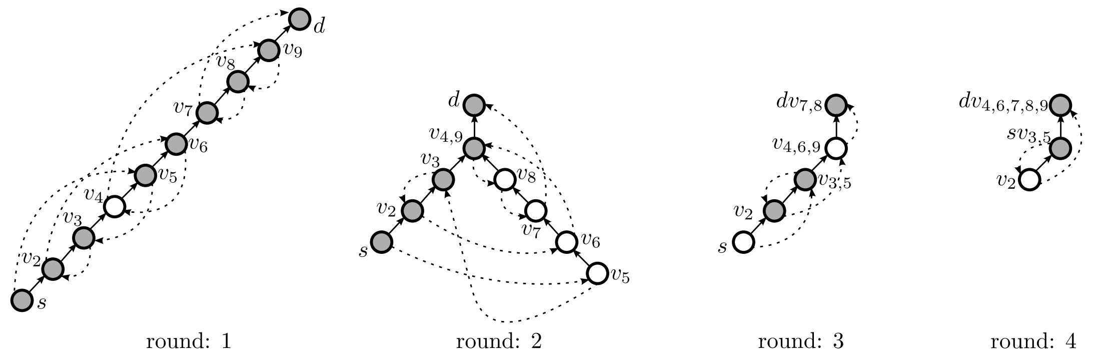

Figure 8: Example execution of *Peacock*. Updated nodes are shown in white. The initial network is a line (on the *left*). An update of the node with the largest distance v_4 and the merging of v_4 and v_9 leads to a tree shown for round 2. Here the nodes $v_5 - v_8$ can be updated since they are not on the $s - d$ path. This results in a line again, shown for round 3. In round 4, v_2 will be updated before the last node, which will be updated in round 5.

$O(\log n)$-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.

Algorithm 1 gives the formal listing for *Peacock* and Fig. 8 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* $O(\log n)$ *rounds.*

PROOF. 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. □

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. The number of rounds is hence logarithmic. □

5. RELATED WORK

Our work is motivated by the SDN paradigm, and especially its traffic engineering flexibilities and its support for a programmatic, dynamic, yet formally verifiable network management. [6] 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. [4, 7] 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, 8, 11, 12, 13, 17], synthesizing [15] and checking [9] policies [16] as well as routes [2] has also been studied intensively. In their seminal work, Reitblatt et al. [17] initiated the study of network updates providing strong, per-packet consistency guarantees, and the authors also presented a 2-phase commit protocol. This protocol also forms the basis of the distributed control plane implementation in [1].

Mahajan and Wattenhofer [13] started investigating weaker transient consistency properties—in particular also (strong) loop-freedom—for destination-based routing policies. Mahajan and Wattenhofer proposed an algorithm to "greedily" select a maximum number of edges which can be used early during the route installation process. Our work builds upon [13], but focuses on an alternative, round-based model to measure policy installation times, and also shows that a greedy strategy can lead to a large number of communication rounds—an observation which has also been made in [12]. The measurement studies in [8] and [10] provide empirical evidence for the non-negligible time and high variance of switch updates, further motivating our work.

Researchers have also started investigating consistent updates for networks which include (network function virtualized) middleboxes [14]. Ludwig et al. 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. A different standpoint is promoted by Ghorbani and Godfrey in their work [5]: the authors argue that in the context of network function virtualization, not weaker but rather stronger consistency properties are required.

6. CONCLUSION

We believe that our work opens interesting questions for future research. Most importantly, it would be interesting to derive $\omega(1)$-round lower bounds, or 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 1,000 nodes) we found requires 7 rounds.

Acknowledgments. We would like to thank Matthias Rost for conducting computational experiments which provided us with valuable insights into the structure of optimal schedules, and the anonymous reviewers for their detailed feedback which helped us to improve the presentation of this paper. This research was supported by the EU project UNIFY as well as by the German BMBF Software Campus grant 01IS12056.

7. REFERENCES

[1] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. 34th IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[2] 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.

[3] 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. ACM SIGCOMM HotSDN*, 2013.

[4] N. Feamster, J. Rexford, and E. Zegura. The road to sdn. *Queue*, 11(12):20:20–20:40, 2013.

[5] S. Ghorbani and B. Godfrey. Towards correct network virtualization. In *Proc. ACM HotSDN*, pages 109–114, 2014.

[6] 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.*, 35(5):41–54, 2005.

[7] 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. ACM SIGCOMM*, pages 551–562, 2014.

[8] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang. Dionysus: Dynamic Scheduling of Network Updates. In *Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Chicago, Illinois, USA*, August 2014.

[9] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 99–112, 2013.

[10] M. Kuzniar, P. Peresini, and D. Kostic. What you need to know about sdn flow tables. In *Proc. Passive and Active Measurements Conference (PAM)*, 2015.

[11] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Hong Kong*, August 2013.

[12] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2014.

[13] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proc. 12th ACM Workshop on Hot Topics in Networks (HotNets)*, 2013.

[14] 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. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 459–473, 2014.

[15] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[16] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.

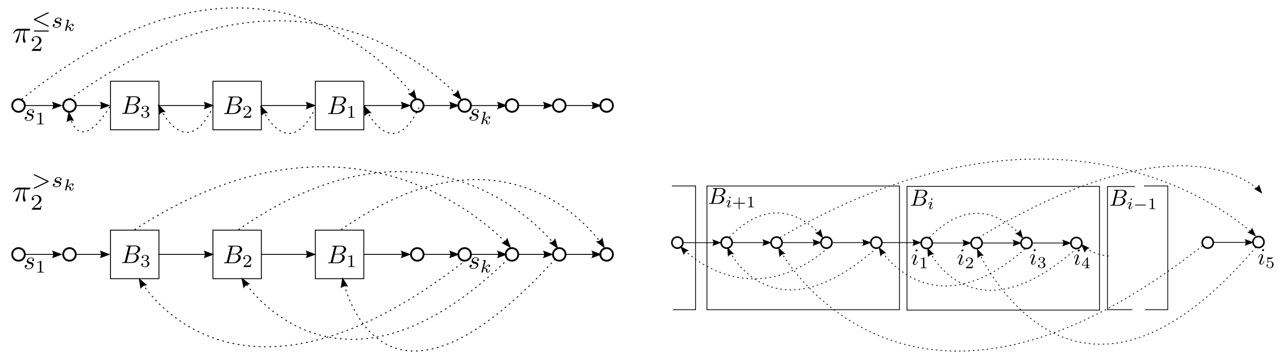

Figure 9: Pattern of a scenario where maximizing the number of updates per round will result in a $\Omega(n)$-round schedule, although a $O(1)$-round schedule would be possible. *Left:* An overview where $\pi_2^{\leq s_k}$ shows the edges of the new route before s_k and $\pi_2^{>s_k}$ those behind s_k. *Right:* A detailed representation of the blocks B_i.

[17] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. ACM SIGCOMM*, pages 323–334, 2012.

APPENDIX

A. 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 [13]) can fail miserably. In fact, a single greedy round may unrevokably change the required number of rounds from $O(1)$ to $\Omega(n)$.

Fig. 9 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.