Transiently Consistent SDN Updates: Being Greedy is Hard[*]

Saeed Akhoondian Amiri[1] Arne Ludwig[1] Jan Marcinkowski[2] Stefan Schmid[3]
[1] Technical University Berlin, Germany [2] University of Wroclaw, Poland [3] Aalborg University, Denmark
saeed.amiri@tu-berlin.de, arne@inet.tu-berlin.de jasiekmarc@gmail.com schmiste@cs.aau.dk

Abstract

The software-defined networking paradigm introduces interesting opportunities to operate networks in a more flexible yet formally verifiable manner. Despite the logically centralized control, however, a Software-Defined Network (SDN) is still a distributed system, with inherent delays between the switches and the controller. Especially the problem of changing network configurations in a consistent manner, also known as the consistent network update problem, has received much attention over the last years. This paper revisits the problem of how to update an SDN in a transiently consistent, loop-free manner. First, we rigorously prove that computing a maximum ("greedy") loop-free network update is generally NP-hard; this result has implications for the classic maximum acyclic subgraph problem (the dual feedback arc set problem) as well. Second, we show that for special problem instances, fast and good approximation algorithms exist.

1 Introduction

By outsourcing and consolidating the control over multiple data-plane elements to a centralized software program, Software-Defined Networks (SDNs) introduce flexibilities and optimization opportunities. However, while a logically centralized control is appealing, an SDN still needs to be regarded as a distributed system, posing non-trivial challenges [3, 10, 18, 19, 20, 21, 22]. In particular, the communication channel between switches and controller exhibits non-negligible and varying delays [11, 21], which may introduce inconsistencies during *network updates*.

Over the last years, the problem of how to consistently update routes in a (software-defined) network has received much attention, both in the systems as well as in the theory community [10, 17, 19, 21, 24]. While in the seminal work by Reitblatt et al. [21], protocols providing strong, per-packet consistency guarantees were presented (using some kind of 2-phase commit technique), it was later observed that weaker ("relaxed"), but transiently consistent guarantees can be implemented more efficiently. In particular, Mahajan and Wattenhofer [19] proposed a first algorithm to update routes in a network in a transiently loop-free manner. Their approach is appealing as it does not require packet tagging (which comes with overheads in terms of header space and also introduces challenges in the presence of middleboxes [23] or multiple controllers [3]) or additional TCAM entries [3, 21] (which is problematic given the fast growth of forwarding tables both in the Internet as well as in the virtualized datacenters [2]). Moreover, the relaxed notion of consistency also allows (parts of the) paths to become available sooner [19].

Concretely, to update a network in a transiently loop-free manner, an algorithm can proceed *in rounds* [17, 19]: in each round, a "safe subset" of (so-called OpenFlow) switches is updated, such that, independently of the times and order in which the updates of this round take effect, the network is always consistent. The scheme can be implemented as follows: After the switches of round t have confirmed the successful update (e.g., using acknowledgements [14]), the next subset of switches for round $t + 1$ is scheduled.

[*]The research of Saeed Akhoondian Amiri has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC consolidator grant DISTRUCT, agreement No 648527). The research of Arne Ludwig and Stefan Schmid was supported by the EU project UNIFY.

It is easy to see that a simple update schedule always exists: we can update switches one-by-one, proceeding from the destination toward the source of a route. In practice, however, it is desirable that updates are fast and new routes become available quickly: Ideally, in order to be able to use as many new links as possible, one aims to maximize the number of concurrently updated switches [19]. We will refer to this approach as the *greedy approach*.

This paper revisits the problem of updating a maximum number of switches in a transiently loop-free manner. In particular, we consider the two different notions of loop-freedom introduced in [17]: *strong loop-freedom* and *relaxed loop-freedom*. The first variant guarantees loop-freedom in a very strict, topological sense: no single packet will ever loop. The second variant is less strict, and allows for a small bounded number of packets to loop during the update; however, at no point in time should newly arriving packets be pushed into a loop. It is known that by relaxing loop-freedom, in principle many more switches can be updated simultaneously.

Our Contributions. We rigorously prove that computing the maximum set of switches which can be updated simultaneously, without introducing a loop, is NP-hard, both regarding strong and relaxed loop-freedom. This result may be somewhat suprising, given the very simple graph induced by our network update problem. The result also has implications for the classic Maximum Acyclic Subgraph Problem (MASP), a.k.a. the dual Feedback Arc Set Problem (dFASP): The problem of computing a maximum set of switches which can be updated simultaneously, corresponds to the dFASP, on special graphs essentially describing two routes (the old and the new one). Our NP-hardness result shows that MASP/dFASP is hard even on such graphs. On the positive side, we identify network update problems which allow for optimal or almost optimal (with a provable approximation factor less than 2) polynomial-time algorithms, e.g., problem instances where the number of leaves is bounded or problem instances with bounded underlying undirected tree-width.

Organization. The remainder of this paper is organized as follows. Section 2 introduces preliminaries and presents our formal model. In Section 3, we prove that computing greedy updates is NP-hard, both for strong and for relaxed loop-freedom. Section 4 describes polynomial-time (approximation) algorithms. After reviewing related work in Section 5, we conclude and discuss future work in Section 6. Some technical details and longer discussions appear in the *arXiv Report 1605.03158*.

2 Model

We are given a network and two policies resp. *routes* π_1 (the *old policy*) and π_2 (the *new policy*). Both π_1 and π_2 are simple directed paths (*digraphs*). Initially, packets are forwarded (using the *old rules*, henceforth also called *old edges*) along π_1, and eventually they should be forwarded according to the new rules of π_2. Packets should never be delayed or dropped at a switch, henceforth also called *node*: whenever a packet arrives at a node, a matching forwarding rule should be present. Without loss of generality, we assume that π_1 and π_2 lead from a source s to a destination d.

We assume that the network is managed by a controller which sends out forwarding rule updates to the nodes. As the individual node updates occur in an asynchronous manner, we require the controller to send out simultaneous updates only to a "safe" subset of nodes. Only after these updates have been confirmed (*ack*ed), the next subset is updated.

We observe that nodes appearing only in one or none of the two paths are trivially updateable, therefore we focus on the network G induced by the nodes V which are part of *both* policies π_1 *and* π_2, i.e., $V = \{v : v \in \pi_1 \wedge v \in \pi_2\}$. We can represent the policies as $\pi_1 = (s = v_1, v_2, \dots, v_\ell = d)$ and $\pi_2 = (s = v_1, \pi(v_2), \dots, \pi(v_{\ell-1}), v_\ell = d)$, for some permutation $\pi : V \setminus \{s,d\} \to V \setminus \{s,d\}$ and some number ℓ. In fact, we can represent policies 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 policy π_1, and $out_2(v)$ (resp. $in_2(v)$) denote the outgoing (resp. incoming) edge according to policy π_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 π_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 ($U = V$).

We require that paths be loop-free [19], and distinguish between *Strong Loop-Freedom* (SLF) and *Relaxed Loop-Freedom* (RLF) [17].

Strong Loop-Freedom. We want to find an *update schedule* $U_1, U_2, \ldots, U_k$, 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. Thus, 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.

In the following we will call an edge (u, v) of the new policy π_2 *forward*, if v is closer (with respect to π_1) to the destination, resp. *backward*, if u is closer to the destination. It is also convenient to name nodes after their outgoing edges w.r.t. policy π_2 (e.g., *forward* or *backward*); similarly, it is sometimes convenient to say that we *update an edge* when we update the corresponding node.

While the initial network configuration consists of two paths, in later rounds, the already updated 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. 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.

Relaxed Loop-Freedom. *Relaxed Loop-Freedom* (RLF) 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 bounded 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. The difference is that we only care about the subset E'_t of E_t consisting of edges *reachable from the source s*.

The Greedy Approach. Our objective is to update simultaneously as many nodes (or equivalently, edges) as possible: an objective initially studied in [19], which may also be seen as a greedy approach to minimize the number of rounds[1]. Note that in the first round, computing a maximum update set is trivial: All forward edges can be updated simultaneously, as they will never introduce a cycle, but no backward edge can be updated in the first round, as it can always induce a cycle; horizontal edges do not exist in the first round. Also observe that since all nodes lie on the path from source to destination, this holds for both strong and relaxed loop-freedom. However, as we will show in this paper, already in the second round, a computationally hard problem can arise.

3 Being Greedy is Hard

Interestingly, although the underlying graphs are very simple, and originate from just two (legal) paths, we now prove that the loop-free network update problem is NP-hard.

Theorem 1. *The greedy network update problem, the problem of selecting a maximum set of nodes which can be updated simultaneously, is NP-hard.*

Our reduction is from the NP-hard *Minimum Hitting Set* problem. This proof is similar for both consistency models: strong and relaxed loop-freedom, and we can present the two variants together. The inputs to the hitting set problem are:

[1]It is known however that in the worst case, a greedy approach can lead to an unnecessarily large number of rounds [17]

1. A universe of m elements $\mathcal{E} = \{\varepsilon_1, \varepsilon_2, \dots, \varepsilon_m\}$.
2. A set $S = \{S_1, S_2, S_3, \dots, S_k\}$ of k subsets $S_i \subseteq \mathcal{E}$.

The objective is to find a subset $\mathcal{E}' \subseteq \mathcal{E}$ of minimal size, such that each set S_i includes at least one element from $\mathcal{E}'$: $\forall S_i \in S : S_i \cap \mathcal{E}' \neq \varnothing$. In the following, we will assume that elements are unique and can be ordered $\varepsilon_1 < \varepsilon_2 \dots < \varepsilon_m$. The idea of the reduction is to create, in polynomial time, a legal network update instance where the problem of choosing a maximum set of nodes which can be updated concurrently is equivalent to choosing a minimum hitting set. While in the initial network configuration, essentially describing two paths from s to d, a maximum update set can be chosen in polynomial time (simply update all forwarding edges but no backward edges), we show in the following that already in the second round, the problem can be computationally hard.

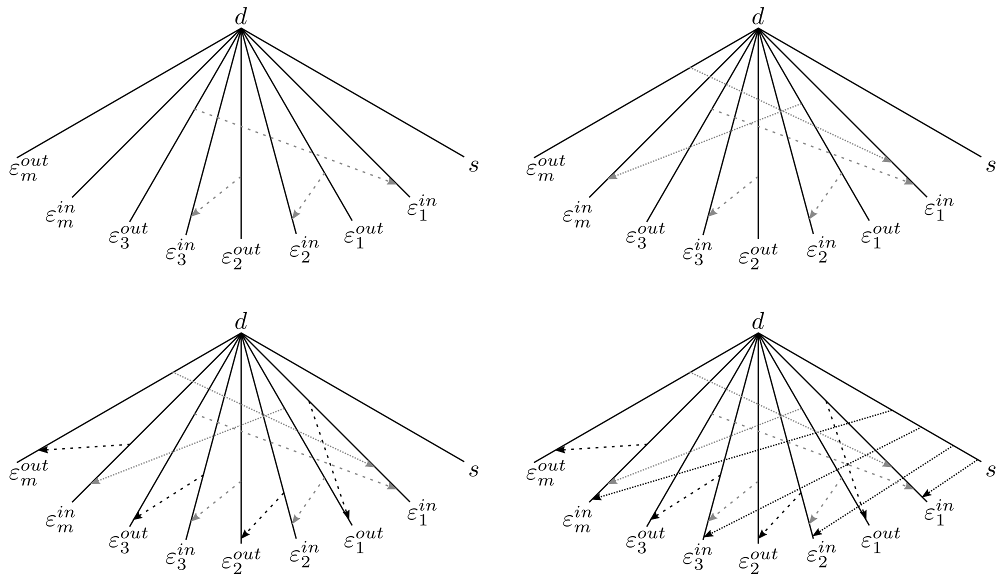

Figure 1: Example: Construction of network update instance given a hitting set instance with $\mathcal{E} = \{1, 2, 3, \dots, m\}$ and $S = \{\{1, 2, 3\}, \{1, m\}\}$. Each element $\varepsilon \in \mathcal{E}$ is represented by a pair of branches, one called outgoing (*out*) and one incoming (*in*). Moreover, we add a branch representing the $s - d$ path on the very right. The black branches represent already installed rules (either old or updated in the first round), and new rules (*dashed*) are situated between the branches. There are three types of to-be-updated, dashed edges: one type represents the sets (loosely and densely dashed grey), one type represents element selector edges (between in and out branch, loosely dashed black), and one type is required to connect the $s - d$ path to the elements (densely dashed grey). We prove that such a scenario can be reached after one update round where all (and only) forward edges are updated. ***Top-left:*** Each loosely dashed grey edge represents $m + 1$ edges, and is used to describe the set $\{1, 2, 3\}$:$(1, 2), (2, 3), (3, 1)$. ***Top-right:*** Each densely dashed grey edge represents $m + 1$ edges and is used for the set $\{1, m\}$: $(1, m), (m, 1)$. ***Bottom-left:*** The loosely dashed black edges are single edges and are the element selector edges, representing the decision if an element is part of $\mathcal{E}'$ or not. ***Bottom-right:*** Each densely dashed edge visualizes $m \cdot (m + 1)$ edges from the s-branch to the incoming branches of every $\varepsilon \in \mathcal{E}$.

More concretely, based on a hitting set instance, we aim to construct a network update instance of the following form, see Figure 1. For each element $\varepsilon \in \mathcal{E}$, we create a pair of branches ε^{in} and ε^{out}, i.e., $2m$ branches in total. To model the relaxed loop-free case, in addition to the $\mathcal{E}$ branches, we add a source-destination branch, from s to d, depicted on the right in the figure. We will introduce the following to-be-updated new edges:

1. **Set Edges (SEs):** The first type of edges models sets. Let us refer to the (ordered) elements in a given set S_i by $\varepsilon_1^{(i)} < \varepsilon_2^{(i)} < \varepsilon_3^{(i)} \ldots$. For each set $S_i \in S$, we now create $m+1$ edges from each $\varepsilon_j^{(i)}$ to $\varepsilon_{j+1}^{(i)}$, in a modulo fashion. That is, we also introduce $m+1$ edges from the last element to the first element of the set. These edges start at the *out* branch of the smaller index and end at the *in* branch of the larger index. There are no requirements on how the edges of different sets are placed with respect to each other, as long as they are not mixed. Moreover, only one instance of multiple equivalent SEs arising in multiple sets must be kept.

2. **Anti-selector Edges (AEs):** These m edges constitute the decision problem of whether an element should be included in the minimum hitting set. AEs are created as follows: From the top of each *in* branch we create a *single* edge to the bottom of the corresponding *out* branch. That is, we ensure that an update of the edge from ε_i^{in} to ε_i^{out} is equivalent to $\varepsilon_i \notin \mathcal{E}'$, or, equivalently, every $\varepsilon_i \in \mathcal{E}'$ will not be included in the update set.

3. **Relaxed Edges (WEs):** These edges are only needed for the relaxed loop-free case. They connect the *s-d* branch to the other branches in such a way that no loops are missed. In other words, the edges aim to emulate a strong loop-free scenario by introducing artificial sources at the bottom of each branch. To achieve this, we create a certain number of edges from the *s*-branch to the bottom of every *in* branch. The precise amount will be explained at the detailed construction part of creating parallel edges. See Figure 1 *bottom-left* for an example.

The rationale is as follows. If no *Anti-selector Edges* (AEs) are updated, all *Relaxed Edges* (WEs) as well as all *Set Edges* (SEs) can be updated simultaneously, without introducing a loop. However, since there are in total exactly m AEs but each set of SEs are $m+1$ edges (hence they will all be updated), we can conclude that the problem boils down to selecting a maximum number of element AEs which do not introduce a loop. The set of non-updated AEs constitutes the selected sets, the hitting set: There must be at least one element for which there is an AE, preventing the loop. By maximizing the number of chosen AEs (maximum update set) we minimize the hitting set.

Let us consider an example: In Figure 1 *bottom-right*, if for a set S_i every AE of $\varepsilon_i \in S_i$ is updated, a cycle is created: updating edges ε_1^{in} and ε_m^{in} results in a cycle with the $m+1$ edges from ε_1^{out} and ε_m^{out}. Note that the resulting network update instance is of polynomial size (and can also be derived in polynomial time). In the remainder of the proof, we show that the described network update instance is indeed legal, e.g., we have a single path from source to destination, and this instance can actually be obtained after one update round.

3.1 Concepts and Gadgets

Before we describe the details of the construction, we first make some fundamental observations regarding greedy updates.

Introducing Forwarding Edges and Branches: First, a delayer concept is required to establish forwarding edges for the second round. Observe that every forwarding edge (a,b), with $a < b$, is always updated by a greedy algorithm in the first round. A delayer is used to construct a forward edge (a,b), with $a < b$, that is created in the second round. A *delayer* for edge (a,b) consists of two edges: an edge pointing backwards to a' from a with $a' < a$, plus an edge pointing from there to b. The forward edge (a',b) will be updated in the first round, which yields an edge (a,b) due to merging (see Figure 2).

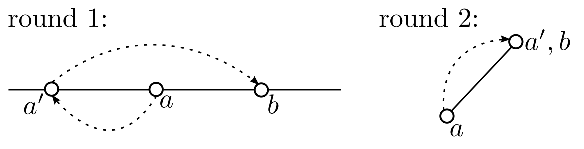

Figure 2: Delayer concept: A forwarding edge $(a, a'b)$ can be created in round 2 using a helper node a'.

We next describe how to create the *in* and *out* branches as well as the s branch pointing to the destination d (recall Figure 1). This can be achieved as follows: From a node close to the source s, we create a path of forward edges which ends at the destination. Each of these forward edges will be updated in the first round, and hence merged with its respective successor, which will be the destination for the very last forward edge. The nodes belonging to these forward edges will be called *branching nodes*. Every node in-between two *branching nodes* will be part of a new branch pointing to the destination. See Figure 3 for an example. The rightmost node before the *branching node* on the line will also be the topmost node on the branch after the first round update (as long as it has an outgoing backward edge, hence not being updated in the first round). We will use the terms right and high (rightmost-topmost) and left-low for the first and second round interchangeably.

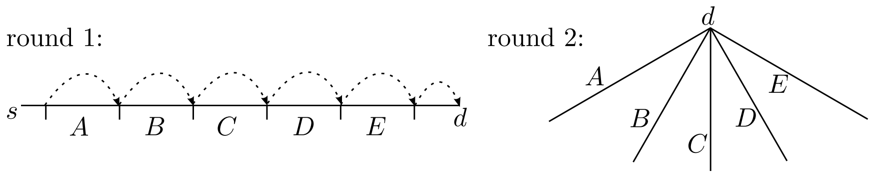

Figure 3: Creating branches after a greedy update of forward edges.

Introducing Special Segments: In our construction, we split the line (old path) into disjoint segments which will become independent branches at the beginning of the second round. In addition to these segments, there will be two special segments, one at the beginning and one at the end. The first will not even become an independent branch at the beginning of the second round, but is merely used to realize the delayer edges. Behind the very last segment (ε_1^{in}) and just before d, there is a second special segment, which we call *relaxed*: it is needed to create the branch with the source s at the bottom and its connections to the other ε_i^{in} branches.

In our construction, SEs come in groups of $m+1$ edges. These edges must eventually be part of a legal network update path, and must be connected in a loop-free manner. In other words, to create the desired problem instance, we need to find a way to connect two branches b_1 and b_2 with $m+1$ edges, such that there is a single complete path from s to d. Furthermore, these edges should not form a loop.

In the *arXiv Report 1605.03158*, we describe how parallel edges can be constructed.

3.2 Connecting the Pieces

Given these gadgets, we are able to complete the construction of our problem instance.

Realizing the Delayer: The first created segment, *temp*, serves for edges that are created using the *delayer* concept. This is due to our construction: every node that will be created in this interval in our construction will be a forward node and therefore updated in the first greedy round. The *temp* segment will be located right after the source s on the line.

Realizing the Branches: We create two segments for each $\varepsilon \in \mathcal{E}$, one *out* and one *in*, and sort them in descending global order (and depict them from left to right) w.r.t. $\varepsilon \in \mathcal{E}$, with the *out* segment closer to s than the *in* segment for each ε, i.e. $\varepsilon_m^{out}, \varepsilon_m^{in}, \ldots, \varepsilon_2^{out}, \varepsilon_2^{in}, \varepsilon_1^{out}, \varepsilon_1^{in}$.

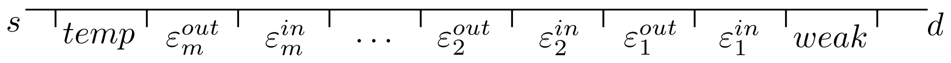

Figure 4: Illustration of how to split the old line into segments according to the amount of needed branches in the second round.

Connecting the Path: We will now create the new path from the source s to the destination d through all the different segments. This path requires additional edges. We will ensure that these edges can always be updated and hence do not violate the selector properties. Moreover, we ensure that they do not introduce a loop. In order to create a branch with s at the bottom (to ensure that the proof will also hold for relaxed loop-freedom), we start our path from the source s to a node $relaxed-bot$ on the very left part of the *relaxed* segment. From here we need to create the $m\cdot\ (m+1)$ connections to every other ε_i^{in} branch, more precisely to the very left of the top part of this branch ε_{i-t}^{in}: the relaxed Edges (WEs). Starting from $relaxed-bot$, we create the $m\cdot\ (m+1)$ zigzag edges (described in detail in the technical report only) to the ε_1^{in} segment. Once this is done, we repeat this process for the remaining ε_i^{in} connecting them in the same order blockwise, as they are ordered on the line. See Figure 5.

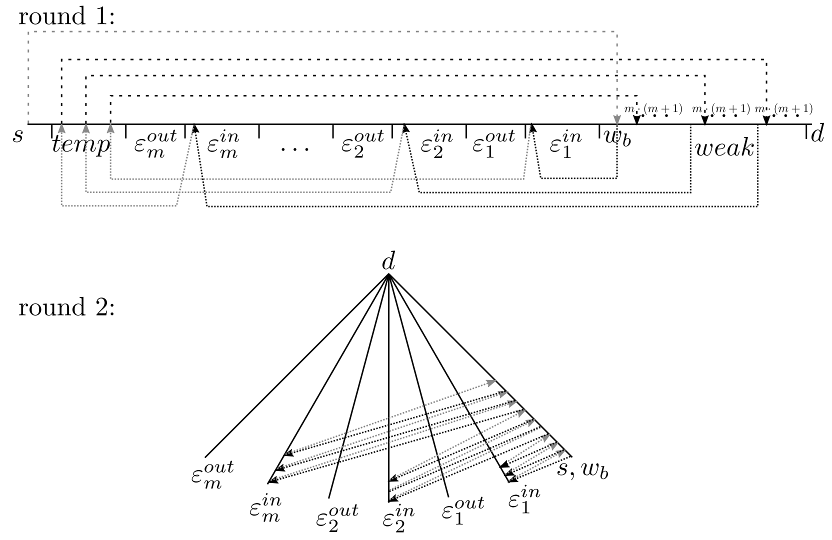

Figure 5: Creating the branch with the source at the bottom and $m\cdot\ (m+1)$ connections to each ε_i^{in} segment of the line, as shown in Section 3.1. The $m\cdot\ (m+1)$ connections are visualized as a single edge in the first round to enhance visibility.

At the beginning of the second round, we will now have a branch with the source s at the bottom and $m+1$ edges to each of the ε_i^{in} branches. The next step is to connect the out branches with the in branches (the Set Edges). For each set $S_j \in S$ and each pair $\varepsilon_i, \varepsilon_l \in S_j$ with no $\varepsilon' \in S_j, \varepsilon_i < \varepsilon' < \varepsilon_l$, we create $m+1$ edges from ε_i^{out} to ε_l^{in}, more precisely to the top part ε_{l-t}^{in} somewhere above the WEs. Each pair $\varepsilon_i, \varepsilon_l$ only needs to connect once with the $m+1$ edges, even if it occurs in several different sets of S. The last element ε_i of a set S_j will additionally need to be connected to the first element of the set (the modulo edges).

After the $m+1$ connections to ε_m^{in}, the path returns at the right most (or highest in the (s,d)-branch) node in the *relaxed* segment. From here we create a backward edge to the left part of ε_1^{out}. Here, we create $m+1$ connections to every ε_i^{in}, which is the next larger element in any of the sets. An example is shown in Figure 6.

To complete the $m+1$ connections for every pair, we proceed as follows: we connect the ε_1^{out} branch to all required in-branches, then add the edge from ε_1^{out} to the ε_2^{out} branch, then add the edges from the ε_2^{out} branch to all required in-branches, etc. Generally, we interleave adding the edges from the ε_i^{out} branch to all required in-branches and then add the i-out to $(i+1)$-out edge. Until the path arrives at the end of the last out branch, ε_m^{out}:

- *Step A - Create the $m+1$ set specific edges:* Here we create $m+1$ connections to every successor in the respective sets (at most once per pair). If this element is the largest element in a set, it needs to be connected to the in part of the smallest element of this set again. Here the delayer concept needs to be used for the modulo edges.

- *Step B - Connecting the out branches:* In order to create the next $m+1$ connections from the next out segment ε_{i+1}^{out}, we need to connect it from our current out segment ε_i^{out}. The edge therefore needs to

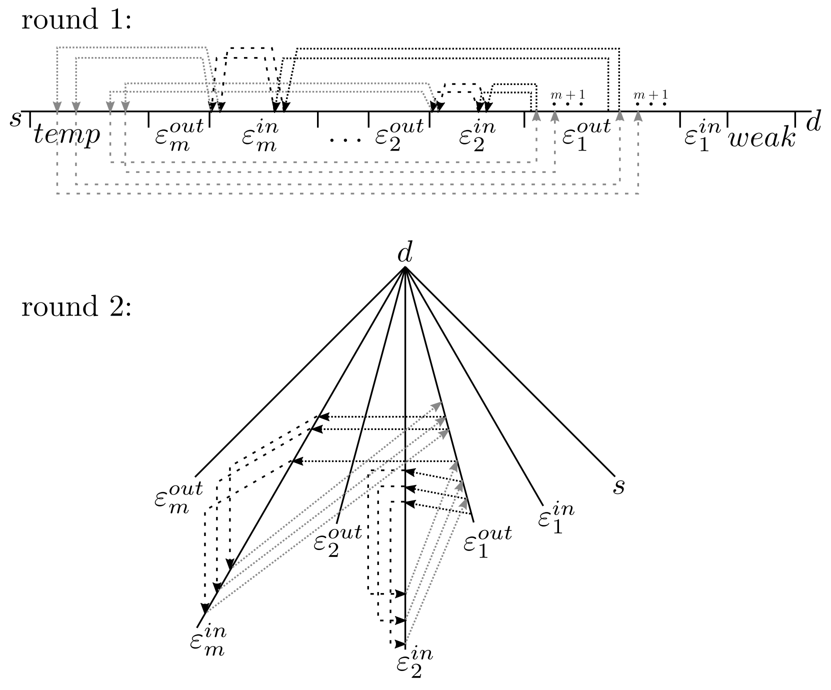

Figure 6: Connecting the ε_1^{out} branch with the branches ε_2^{in}, ε_3^{in}, ε_m^{in}. This scenario would be created for the sets: $\{1,2,\dots\}$,$\{1,3,\dots\}$, $\{1,m\}$. The densely dashed black edges show the outgoing edges from ε_1^{out}. The loosely dashed black edges are the backward edges from the top part of a branch ε_i^{in} to its bottom part (ε_{i-t}^{in} to ε_{i-b}^{in}). The densely dashed grey edges are the way back from ε_i^{in} to ε_1^{out} and are needed to complete the path.

point to the rightmost part of ε_{i+1}^{out}. Since this edge is always a backward edge in the first round (we start closer to the destination and move backward towards the source), it will turn out to be an edge which points to the very top of ε_{i+1}^{out} at the beginning of the second round. This assures that there are no loops created, since the only way is going directly towards the destination. From here we create an edge pointing to the very left side of ε_{i+1}^{out} (evolving to a backward rule from top to bottom of the branch in the second round, hence not being part of the update set in the first nor the second round).

To finish the construction, we need to add the anti-selector edges (AEs), and connect the in and out branches of every single ε_i with each other. The goal is to create, for each given i, an edge from the top of each ε_i^{in} to the bottom of each ε_i^{out}. This way, if this edge is included in the update, a loop may be formed: as every incoming edge to ε_i^{in} arrives below the AEs start point and every outgoing edge on ε_i^{out} is above AE's destination. The decision to not include one of these edges is equivalent to $\varepsilon_i \in \mathcal{E}'$ in the minimum hitting set problem. In order to keep the path connected we will also need to include edges from ε_i^{out} to ε_{i+1}^{in}, compare Figure 7. These edges will point to the top of ε_{i+1}^{in} and therefore do not create loops, since the only way is going directly to the destination. From here we create another backward edge to its left neighbor such that there is no possible other way than traversing towards d from this point. Without this backward edge loops may be created, since it introduces connections between branches which are not both in a set S_i of the hitting set problem. Therefore, an update of one of the additional connector edges will never lead to a loop, and the edges can all be included in the update set of the round 2.

The construction of these edges is straightforward. From the end of the current path which is located on the ε_m^{out} segment, we create a delayed edge (over *temp*) to the very right part of the ε_1^{in} segment. From here we construct the path as described with a short backward edge to its left neighbor and then to the very left part of the ε_i^{out} segment and again to the very right part of the ε_{i+1}^{in} segment afterwards, until we arrive at

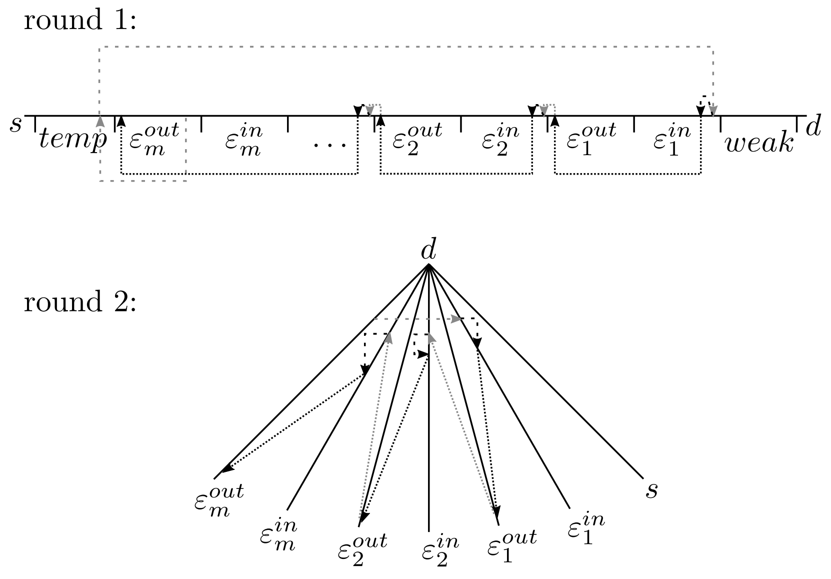

Figure 7: Connecting the in and out branches of every ε_i, shown in densely dashed black. The edges shown in densely dashed grey are needed to keep the path complete and the backward edges in loosely dashed black are needed to ensure that only the destination can be reached from that point in the second round.

the very left part of the ε_m^{out} segment.

It remains to create the segments and branches for the second round. From ε_m^{out}, we create a backward edge to the *temp* part. From here we use the branching concept and connect all horizontal nodes in-between the single parts that we created on the line (see Figure 8).

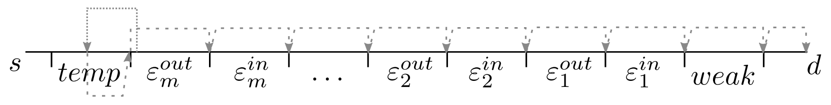

Figure 8: Connecting the segments with forward edges. This creates a single branch from the destination for every segment due to the merging. The edge shown in loosely dashed grey is connecting this step with the step before.

In summary, we ensured that already after a single greedy first update round, we end up in a situation where choosing the maximum set of updateable nodes is equivalent to choosing the minimum hitting set.

4 Polynomial-Time Algorithms

While the computational hardness is disappointing, we can show that there exist several interesting specialized and approximate algorithms.

Optimal Algorithms. There are settings where an optimal solution can be computed quickly. For instance, it is easy to see that in the first round, in a configuration with two paths, updating all forward edges is optimal: Forward edges never introduce any loop, and at the same time we know that backward edges can never be updated in the first round, as any backward edge alone (i.e., taking effect in the first round), will immediately introduce a loop. In the following, we first present an optimal algorithm for SLF, for trees with only two leaves. We will then extend this algorithm to RLF. In the our arXiv report we will also show that optimal solutions can be computed efficiently if the underlying undirected graph is of bounded tree-width.

Lemma 1. *A maximum SLF update set can be computed in polynomial-time in trees with two leaves.*

Proof. Recall that there are three types of new edges in the graph (see also Figure 9): forward edges (F), backward edges (B) and horizontal edges (H), hence $E = H \cup B \cup F$. Moreover, recall that forward edges can always be updated while backward edges can never be updated in SLF. Thus, the problem boils down to selecting a maximum subset of H, pointing from one branch to the other. If there is a simple loop $C \in G$ such that $H^C = E(C) \cap H \neq \varnothing$, then $|H^C| = 2$ and we say that the two edges $e_1, e_2 \in H^C$ cross each other, written $e_1 \times e_2$.

We observe that the different edge types can be computed efficiently. For illustration, suppose the policy graph $G = (V, E)$ (the union of old and new policy edges) is given as a straight line drawing Π in the 2-dimensional Euclidean plane, such that the old edges of the 2-branch tree form two disjoint segments which meet at the root of the tree (the destination), and such that each node is mapped to a unique location. Given the graph, such a drawing (including crossings) in the plane can be computed efficiently. Also note that there could be other edges which intersect w.r.t. the drawing Π, but those are not important for us.

Now create an auxiliary graph $G' = (V', E')$ where $V' = \{v_e \mid e \in H\}$, $E' = \{(v_{e_1}, v_{e_2}) \mid e_1, e_2 \in H : e_1 \times e_2\}$. The graph G' is bipartite, and therefore finding a minimum vertex cover $VC \in V(G)$ is equivalent to finding maximum matching, which can be done in polynomial time. Let $H' = \{e \mid e \in H : v_e \in VC\}$, then the set H' is a minimum size subset of H which is not updateable. Therefore the set $H \setminus H'$ is the maximum size subset of H which we can update in a SLF manner (the complement of H' is a maximum independent set in G' and therefore, by the definition of the collision graph G', a maximum updateable set).

We conclude the proof by observing that all these algorithmic steps can be computed in polynomial time. □

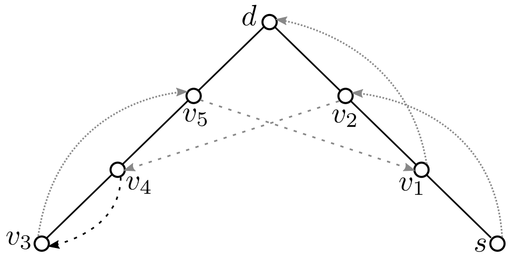

Figure 9: Concept of horizontal edges shown in loosely dashed grey. Both horizontal edges (v_2, v_4) and (v_5, v_1) are crossing each other. The backward edge (v_4, v_3) is shown in loosely dashed black and the forward edges in densely dashed grey. Note that s does not necessarily have to be a leaf.

Lemma 2. *A maximum RLF update set can be computed in polynomial-time in trees with two leaves.*

Approximation Algorithms. Even in scenarios for which there is no optimal polynomial time scheduling algorithm, there can exist good approximations. It is easy to observe that there is a reduction to the Maximum Acyclic Subgraph Problem (*MASP*) which ensures that both RLF and SLF can be approximated at least as well as MASP. It is also easy to see that the problem for strong loop-freedom (for SLF) is 1/2-approximable in general, as the problem boils down to finding a maximum subset of H edges which are safe to update, and at least half of the H edges are pointing out to the left resp. right, and we can take the majority. Similarly for RLF: let F be the set of vertices where every $v \in F$ appears along a walk between source and destination. Similar to SLF, at least half of the edges of F are safe to update, and we can find these edges quickly. Also every $e = (u, v) \in E(G)$, where $u \notin F$ or $v \notin F$, is safe to update. So we have at least a 1/2-approximation.

However, for a small number of leaves, even better approximations are possible. The following lemma can be proven by an approximation preserving reduction to the hitting set problem.

Lemma 3. *The optimal SLF schedule is* $2/3$*-approximable in polynomial time in scenarios with exactly three leaves. For scenarios with four leaves, there exists a polynomial-time* $7/12$*-approximation algorithm.*

5 Related Work

In their seminal work, Reitblatt et al. [21] 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 [3]. Mahajan and Wattenhofer [19] started investigating a hierarchy of transient consistency properties—in particular also (strong) loop-freedom but for example also bandwidth-aware updates [1]—for destination-based routing policies. The measurement studies in [11] and [15] provide empirical evidence for the non-negligible time and high variance of switch updates, further motivating their and our work. In their paper, Mahajan and Wattenhofer proposed an algorithm to "greedily" select a maximum number of edges which can be used early during the policy installation process. This study was recently refined in [7, 8], a parallel work to ours, where the authors also establish a hardness result for destination based routing (single- and multi-destination). Our work builds upon [19] and complements the results in [7, 8]: We consider the scheduling complexity of updating *arbitrary routes* which are not necessarily destination-based. Interestingly, our results (using a different reduction) show that even with the requirement that the initial and the final routes are simple paths, the problem is NP-hard. Moreover, our results hold for both the strong SLF and the relaxed RLF loop-free problem variants introduced in [17]. The SLF can be seen as a special variant of the Dual Feedback Arc Set Problem (FASP) resp. Maximum Acyclic Subgraph Problem (MASP): important classic problems in approximation theory [12]. In particular, it is known that dual-FASP/MASP can be $1/2+\varepsilon$ approximated on general graphs (for arbitrary small ε). The results presented in this paper also imply that better approximation algorithms and even optimal polynomial-time algorithms exist for special graph families, namely graph families describing network update problems; this may be of independent interest. The RLF variant is a new optimization problem, and to the best of our knowledge, existing bounds are not applicable to this problem. We should note that FASP is in FPT [4], and the hitting set problem is W[2]-hard [6]. In our hardness construction we actually find a reduction from hitting set to FASP for particular graph classes. But the reduction is not parameter preserving, so the W-hierarchy does not collapse. Finally, our model is orthogonal to the network update problems aiming at minimizing the number of interactions with the controller (the so-called *rounds*), which we have recently studied for single [17] and multiple [5] policies, also including additional properties, beyond loop-freedom, such as waypointing [16]. The two objectives conflict [17], a good approximation for the number of update edges yields a bad approximation for the number of rounds, and vice versa.

6 Open Problems

An interesting open question regards whether SLF and RLF can be approximated well or even solved optimally in polynomial time, in graphs of bounded tree width (see our accompanying arXiv report for a longer discussion). For graphs of bounded tree-width and degree, this is unlikely, due to the related negative results for the Feedback Arc Set Problem (FASP) [13]. Also, in bounded degree graphs, vertex covering problems [9] are NP-complete, and these results extend to FASP. However, problems on bounded *directed path-width* graphs may be solvable efficiently: none of the negative results for bounded degree graphs on graphs of bounded directed treewidth seem to be extendable to digraphs of bounded directed pathwidth with bounded degree. Indeed, we claim that there is a function $f: \mathbb{N} \to \mathbb{N}$ such that for a digraph G of directed path-width k and maximum degree d, there is an algorithm which runs in time and space $n^{f(k+d)}$ and finds an optimal solution to the FASP problem.

References

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

[2] Tian Bu, Lixin Gao, and Don Towsley. On characterizing bgp routing table growth. *Comput. Netw.*, 2004.

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

[4] Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *J. ACM*, 55(5):21:1–21:19, November 2008.

[5] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. In *Proc. 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[6] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

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

[8] Klaus-Tycho Förster and Roger Wattenhofer. The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration. In *Proc. 25th International Conference on Computer Communication and Networks (ICCCN)*, 2016.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[10] Soudeh Ghorbani and Brighten Godfrey. Towards correct network virtualization. In *Proc. ACM HotSDN*, pages 109–114, 2014.

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

[12] Viggo Kann. On the approximability of np-complete optimization problems. *PhD Thesis*, 1992.

[13] Stephan Kreutzer and Sebastian Ordyniak. Digraph decompositions and monotonicity in digraph searching. *Theoretical Computer Science*, 412(35):4688 – 4703, 2011.

[14] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. Providing reliable FIB update acknowledgments in SDN. In *Proc. 10th ACM CoNEXT*, pages 415–422, 2014.

[15] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. *Proc. Conference on Passive and Active Measurement (PAM)*, chapter What You Need to Know About SDN Flow Tables. 2015.

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

[17] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.

[18] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan 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.

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

[20] Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. Decentralizing sdn policies. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 663–676, 2015.

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

[22] Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. In *Proc. ACM HotSDN*, August 2013.

[23] Z. Qazi et al. Simple-fying middlebox policy enforcement using sdn. In *Proc. ACM SIGCOMM*, 2013.

[24] Wenxuan Zhou, Dong (Kevin) Jin, Jason Croft, Matthew Caesar, and Philip Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 73–85, 2015.