Fast Algorithms for Loop-Free Network Updates using Linear Programming and Local Search

Harald Räcke
TU Munich

Stefan Schmid
TU Berlin

Radu Vintan
EPFL

***Abstract*—To meet stringent performance requirements, communication networks are becoming increasingly programmable and flexible, supporting fast and frequent adjustments. However, reconfiguring networks in a dependable and transiently consistent manner is known to be algorithmically challenging. This paper revisits the fundamental problem of how to update the routes in a network in a (transiently) loop-free manner, considering both the Strong Loop-Freedom (SLF) and the Relaxed Loop-Freedom (RLF) property.**

We present two fast algorithms to solve the SLF and RLF problem variants *exactly*, to optimality. Our algorithms are based on a parameterized integer linear program which would be intractable to solve directly by a classic solver. Our main technical contribution is a lazy cycle breaking strategy which, by adding constraints lazily, improves performance dramatically, and outperforms the state-of-the-art exact algorithms by an order of magnitude on realistic medium-sized networks. We further explore *approximate* algorithms and show that while a relaxation approach is relatively slow, with a local search approach short update schedules can be found, outperforming the state-of-the-art heuristics.

On the theoretical front, we also provide an approximation lower bound for the update time of the state-of-the-art algorithm in the literature. As a contribution to the research community, we made all our code and implementations publicly available.

I. Introduction

With the popularity of distributed data-centric applications and AI, communication traffic is growing explosively. As the performance of these applications critically depends on the underlying network, the efficient use of communication resources and network operation is important.

In order to meet these increasingly stringent performance requirements, over the last years, great efforts have been made to render communication networks more flexible and programmable, especially in the context of software-defined networks (SDNs). These technologies allow updating the routes taken by network traffic programmatically, and hence in a fine-grained and fast manner, taking into account the current traffic demand. For example, in a software-defined network, a centralized controller can instruct a node (i.e., a switch or router) to change its forwarding rules determining the routes taken by packets, from a logically centralized perspective.

However, operating networks in a highly adaptive manner is still challenging as frequent changes may harm reliability. In particular, a dependable network must not only ensure a correct operation before and after an update occurred, but also during the update. Indeed, even an SDN which is controlled centrally, remains an asynchronous and distributed system: updates sent out by a controller simultaneously may take effect at different nodes in an asynchronous order. For example, if a controller needs to update a loop-free route r_1 to a loop-free route r_2, it needs to update the forwarding tables of all involved routers. However, once the controller issues updates to a subset of routers, the order in which these routers apply the update and effectively change their forwarding tables is not defined, i.e. some of the routers might in fact update faster than others. This creates temporary hybrid routes in the network, where there exist both nodes using the old rules and nodes using the new rules, and forwarding may transiently induce a loop.

The problem of how to consistently update a (software-defined) network has received significant attention in the literature, and surveys report on hundreds of approaches [1], [2]. In their seminal work, Reitblatt et al. [3] proposed a solution for this problem involving a *2-Phase Commit Protocol* with *packet tagging*. In this approach, the controller first informs all routers of the new route. Then, the controller adds to all packet headers a version number informing the routers whether the old or the new route is to be used. This approach guarantees that packets are routed either entirely on the old route or entirely on the new route. However, the use of packet tagging comes with overheads and has notable disadvantages [4].

An alternative solution is to partition the updates into $k \geq 1$ batches and schedule them across multiple *rounds*. In each round the controller selects a subset of (not yet updated) routers to update, such that the consistency properties can be guaranteed independently of the order in which the nodes apply the update. The controller waits for the nodes to send an acknowledgement and then continues with the next round.

In this paper we are interested in updating all routers using a minimal number of rounds k, focusing on the most fundamental and widely studied *loop-freedom* property [5]. This property requires that no packet ends up in a loop at any time, where the literature distinguishes between *strong loop-freedom (SLF)* and *relaxed loop-freedom (RLF)* [6]–[8]. For SLF, the problem is provably NP-hard, and also no polynomial-time exact algorithm for RLF is known today [6], [7]. Accordingly, most existing literature revolves around approximate algorithms and heuristics [1], [2].

This paper is motivated by the question whether and to which extent it is possible to solve the loop-free network update problem *exactly*.

Our main ***contributions*** are two fast algorithms to solve the SLF and RLF problem variants *exactly*, that is, optimally: the resulting schedules are minimal. Our approach relies on

an integer linear program which cannot be solved efficiently directly. Our main technical contribution is a lazy cycle breaking technique which, by adding constraints in a lazy fashion, dramatically improves performance. Specifically, our empirical evaluation shows that our approach is an order of magnitude faster than the state-of-the-art exact algorithms on realistic networks of medium sizes.

We also explore to which extent our approach can be used to compute *approximate* solutions using relaxation, i.e., by dropping the integrality constraints and using a suitable rounding. While linear programming can yield good schedules, we find that a novel local search algorithm is significantly faster, and our empirical evaluation shows that the resulting schedules are slightly shorter than the ones computed by the state-of-the-art algorithms.

We report on extensive experiments and also provide new theoretical insights on existing algorithms. In particular, we provide a tight approximation lower bound for the update time of the state-of-the-art algorithm for RLF.

As a contribution to the research community, all our code and experimental artefacts are made publicly available together with this paper [9].

II. Model

We consider the standard model and terminology in the literature [1], [2]. We are given n nodes (i.e., routers or switches) $V \coloneqq \{1, \ldots, n\}$. Packets are initially sent from the *source* $s \coloneqq 1$ to the *destination* $d \coloneqq n$ along the directed path $s = 1, 2, \ldots, n-1, n = d$. This route should be replaced with a new route given by $\sigma(1) = s, \sigma(2), \ldots, \sigma(n-1), \sigma(n) = d$, where σ is a permutation of $\{1, 2, \ldots, n\}$.

The nodes are updated across an asynchronous network. In the *first round*, the controller chooses a subset $V_1 \subseteq V$ of the nodes, and instructs these nodes to update their forwarding tables according to σ. However, the order in which the nodes from V_1 will apply the update is not determined and all orders are considered possible. It can therefore happen that at some point in time some nodes from V_1 use their old forwarding table while others applied the update already and use the new policy. Eventually, all nodes from V_1 have acknowledged the update to the controller, and it can schedule the next round.

In general, when scheduling *round* $t \geq 2$, the controller knows that the nodes from $V_{<t} \coloneqq \cup_{j=1}^{t-1} V_j$ updated, and it chooses a new subset V_t of $V \setminus V_{<t}$ to update in the current round. Again, the nodes from V_t can update in any order. Eventually, the controller updates all nodes, and its *solution/schedule* consists of the subsets $V_1, V_2, \ldots, V_k$ it has chosen. Note that V is partitioned by these subsets and that k is the total number of rounds.

In general, the controller is required to update the routers while at the same time ensuring that certain guarantees are met. We consider both problem variants considered in the literature, Strong and Relaxed Loop Freedom. The following definitions prepare and introduce these notions:

Definition 1 (Full Graph)**.** For an update problem with n nodes and new path/permutation σ, let $G \coloneqq (V, E)$ where[1] $V \coloneqq [1; n]$ be the directed graph given by the edges $E \coloneqq \{(i, i+1) : i \in [1; n-1]\} \cup \{(i, \sigma(\sigma^{-1}(i)+1)) : i \in [1; n-1]\}$. Here, σ^{-1} is the inverse permutation of σ, such that $\sigma^{-1}(i)$ is the index of i in the new path σ, and $\sigma(\sigma^{-1}(i)+1))$ is the next node after i in the route given by σ. Hence, the graph G, which we call the *Full Graph*, contains both the edges of the old and of the new path.

Definition 2 (Update Graph)**.** Assume that the nodes from $V' \subseteq V$ have already updated. This corresponds to the *Update Graph* $G_{V'} \coloneqq (V, E_{V'})$ given by:

$$E_{V'} \coloneqq \{(i, \sigma(\sigma^{-1}(i)+1)) : i \in V'\} \cup \{(i, i+1) : i \in V \setminus V'\}$$

i.e. the already updated nodes can only use their new edge, whereas the other nodes can only use their old edge.

We are now ready to introduce the two relevant consistency properties:

Definition 3 (Strong and Relaxed Loop Freedom (SLF and RLF))**.** For an update problem with n nodes and new path/permutation σ, let $V_1, \ldots, V_k$ be a partitioning of V. In each round t the nodes V_t will be scheduled for update. Recall the notation $V_{<t} \coloneqq \cup_{j=1}^{t-1} V_j$.

We say that the schedule fulfills the *Strong Loop Freedom (SLF)* property iff, for all $t \geq 1$ and subsets $V' \subseteq V_t$, the updated subgraph $G_{V_{<t} \cup V'}$ is cycle-free. This means that, independently of the precise order in which the nodes from V_t update (recall the assumption that all orders are possible), there will never be any loop in the current update graph.

We say that the schedule fulfills the *Relaxed Loop Freedom (RLF)* property iff, for all $t \geq 1$ and subsets $V' \subseteq V_t$, the updated subgraph $G_{V_{<t} \cup V'}$ does not have any cycles reachable from the source node $s = 1$. This means that, independently of the precise order in which the nodes from V_t update (recall the assumption that all orders are possible), there will never be any loop reachable from s in the current update graph. Relaxed Loop Freedom is usually an acceptable choice in practice, because cycles which are not reachable by any packets do not create routing issues.

For both variants one can formulate the following optimization problem:

Definition 4 (SLF/RLF Optimization Problem)**.** Find a schedule $V_1, \ldots, V_k$ with a minimum number of rounds k and which fulfills the SLF/RLF consistency guarantee.

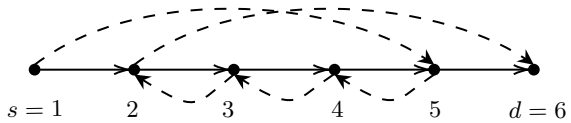

Fig. 1: Example SLF vs. RLF.

[1]For the rest of the paper, we denote by $[1; n]$ the set $\{1, 2, \ldots, n\}$.

Example We consider the instance described by Figure 1. Per convention, the solid edges describe the old route, whereas the dashed edges describe the new route. In this case, the permutation corresponding to the new route is given by $(1, 5, 4, 3, 2, 6)$. If a valid SLF solution updates node $i \in \{3, 4, 5\}$ in round t, then node $i - 1$ is required to have been updated in round $t' \leq t - 1$, for otherwise there is a potential cycle $i - 1 \to i \to i - 1$. An optimal schedule is: $V_1 = \{1, 2\}, V_2 \coloneqq \{3\}, V_3 \coloneqq \{4\}, V_4 \coloneqq \{5\}$. However, if we consider the same instance for the RLF variant of the problem, there exists the shorter schedule which updates $\{1, 2\}$ in the first round, $\{3, 4\}$ in the second round $\{5\}$ in the last round. Indeed, any cycle created during the second round is unreachable from the source, because all packets travel along the route $1, 5, 6$ independently of these subsequent updates. Then, in the third round, 5 can be safely updated.

An easy extension of the above example shows that SLF can require $\Omega(n)$ rounds. However, for the RLF variant, one can show that any instance can be solved using $O(\log n)$ rounds. This bound is achieved by the Peacock algorithm [7], [8], which is presented in Section IV-C.

III. Exact Algorithms

While there already exist Integer Linear Programs (ILP) which model the optimization problem from Definition 4, such as the one used in [8], our modeling approach will be slightly different. Instead of modeling the optimization problem directly, we model the decision variant of the problem: *For a given instance of the loop-free update problem, using either SLF or RLF, and an integer $T \geq 1$, is there a valid SLF/RLF schedule using $\leq T$ rounds?*

A. Strong Loop Freedom

Let $G = ([1; n], E)$ be the *full graph* induced by an instance (n, σ) of the SLF problem, i.e. $E \coloneqq \{(i, i+1) : i \in [1; n-1]\} \cup \{(i, \sigma(\sigma^{-1}(i)+1)) : i \in [1; n-1]\}$. Call these edges *old* and *new* edges, respectively. W.l.o.g. we assume that the sets E_{old} and E_{new} of old and new edges are disjoint (and partition E).

Our LP relaxation will impose the SLF property. To write the corresponding conditions compactly some notation is required. Let $\mathcal{C}$ be the set of the cycles of G and consider some fixed cycle $C \in \mathcal{C}$. Identify a cycle with the vertices it contains, i.e. $C \subseteq [1; n]$. We define the sets C_{old} and C_{new} depending on whether the outgoing edge of a vertex $i \in C$ is old or not. Note that C_{old} and C_{new} thereby partition C.

Lemma 1. *Let $T \geq 1$ be a natural number. Define the variables x_{ti}, y_{ti} for all $t \in \{0, \dots, T\}$ and $i \in [1; n]$. Consider the polytope $LP(T)$ given by the following set of constraints:*

$$y_{0i} = 1, \quad \forall i \in [1; n-1] \tag{1}$$

$$y_{Ti} = 0, \quad \forall i \in [1; n-1] \tag{2}$$

$$x_{ti} + y_{t-1,i} = 1, \quad \forall t \in [1; T], i \in [1; n-1] \tag{3}$$

$$y_{t-1,i} - y_{ti} \geq 0, \quad \forall t \in [1; T], i \in [1; n-1] \tag{4}$$

$$\sum_{i \in C_{old}} x_{ti} + \sum_{y \in C_{new}} y_{ti} \geq 1, \quad \forall t \in [1; T], \forall C \in \mathcal{C} \tag{5}$$

Then, the constraints can be satisfied by integral variables iff there is a schedule with at most T rounds for the corresponding instance (n, σ) of the SLF problem.

Proof. Intuitively, $x_{ti} = 1$ signifies that a new edge leaving from i is in the update graph at time t, and $y_{ti} = 1$ signifies that an old edge leaving from i is in the update graph at time t. In the following we formalize this intuition.

First, note that $0 \leq y_{ti}, x_{ti} \leq 1$ for any t, i, and hence any integral solution is binary. Assume an integral feasible point (x, y) is given. For any $i \in [1; n-1]$, conditions (1), (2) and (4) imply there is a unique $t_i \in [1; T]$ for which $y_{t_i,i} = 0$ and $y_{t_i-1,i} = 1$. Schedule each i in round t_i and define V_t to be the set of nodes which are updated in round t. We claim that conditions (3) and (1) ensure that this schedule does not violate the SLF consistency guarantee. Assume on the contrary that for some t and $V' \subseteq V_t$ the *update graph* $G_{V_{<t} \cup V'}$ contains a cycle C. Fix C and consider a vertex $i \in C$ on this cycle. As C_{old} and C_{new} partition C, we make a case distinction. Clearly, if $i \in C_{\text{old}}$, then $t_i \geq t$ by definition of t_i and thus $y_{t-1,i} = 1$. Condition (3) implies $x_{ti} = 0$. If $i \in C_{\text{new}}$, then $t_i \leq t$ and thus $y_{t,i} = 0$. We have shown that condition (5) is violated for the cycle C, which is a contradiction.

Now assume a valid solution for the given SLF problem instance is given, which schedules the nodes from V_t in round $t \in [1; T]$. Let $t_i \in [1; T]$ be the round in which node i is updated; define $y_{ti} = 1$ for $t < t_i$ and $y_{ti} = 0$ for $t \geq t_i$. Define x according to condition (3). Conditions (1) to (4) are clearly satisfied. Assume condition (5) is violated for some cycle C. Let $V' \coloneqq \{i \in V_t : i \in C_{\text{new}}\}$ and consider the update graph $G' \coloneqq G_{V_{<t} \cup V'}$. If $i \in C_{\text{old}}$, we have $x_{ti} = 0$, hence $y_{t-1,i} = 1$, $t_i \geq t$ and the old edge $(i, i+1)$ exists in G'. If $i \in C_{\text{new}} \cap V_t$, by definition of V' the new edge $(i, \sigma(\sigma^{-1}(i) + 1))$ going out from i is in G'. Else, if $i \in C_{\text{new}} \setminus V_t$: because $y_{ti} = 0$, it follows that $t_i < t$. Hence, $i \in V_{<t}$ and the new edge $(i, \sigma(\sigma^{-1}(i)+1))$ going out from i is in G'. We have shown that the cycle C exists in G', which is a contradiction. □

To solve the optimization SLF problem, we can apply Lemma 1 for all $T \in [1; n]$ and return the smallest T for which the polytope $LP(T)$ is integrally feasible. Obviously, the search procedure for finding such a point cannot be done in polynomial-time. However, a particular algorithmic technique helps the runtime be very good in practical terms. Classic ILP solvers like Gurobi [10] require writing the constraints down explicitly, and this is not tractable for larger n because the number of cycles $C \in \mathcal{C}$ increases dramatically. Hence, we

initially add no cycle constraints at all, and then search for a feasible integral point inside this relaxed version of $LP(T)$. If no such point exists, then adding further cycle constraints would obviously keep the polytope empty, and therefore T needs to be increased. If instead a feasible solution x, y is found, we use the Floyd-Warshall algorithm to find the shortest cycle (w.r.t. weights x_{ti} and y_{ti}). If this cycle has length ≥ 1, the algorithm can stop, because then all cycles in the *full graph* G fulfill the cycle constraint. If instead the cycle is too short we proceed by adding the corresponding constraint in Gurobi and then re-solve the integer linear program with the new constraint. This technique we call *lazy cycle-breaking*.

Hence, while solving our ILP directly would be slower than the state-of-the-art algorithms, we will see that our lazy cycle-breaking technique will dramatically improve performance, outperforming the state-of-the-art by an order of magnitude.

B. Relaxed Loop Freedom

Our modeling for the RLF problem closely resembles the one from the previous section for the SLF problem. In fact, we only need to adapt the cycle constraints of the polytope $LP(T)$. Recall that for the relaxed problem cycles are allowed in the *update graph* as long as they are not reachable from the source node s. To write these constraints properly, we use the following:

Definition 5. Let (n, σ) be an instance of the RLF problem and $G \coloneqq (V, E)$ the *full graph* associated to this instance. Let $\mathcal{S}$ be a set such that:

$$S \coloneqq (P, v) \in \mathcal{S}$$

iff $P \coloneqq (v_1 \coloneqq s, v_2, \ldots, v_k)$ is a tuple of $k \geq 1$ distinct nodes, such that the path P exists in G, $(v_k, v) \in E$ and $v = v_i$ for some $1 \leq i \leq k$. This means that the *S-tuple* $(v_1 = s, \ldots, v_k, v)$ forms a path from s to some node v_k, together with an edge from v_k to an already visited v node closing a cycle (reachable from s). We call such a sequence of edges a *path-cycle structure*.

For $S \coloneqq (P, v) \in \mathcal{S}$ and corresponding *S-tuple* $(v_1 \coloneqq s, \ldots, v_k, v)$, let S_{old} and S_{new} be the vertices from $\{v_1, \ldots, v_k\}$ which use their old and, respectively, their new outgoing edge to form the corresponding *path-cycle structure*. Similarly to how the old and new edges partition E (i.e. an edge is either old or new), S_{old} and S_{new} partition the vertices $\{v_1, \ldots, v_k\}$ appearing in the *path-cycle structure*.

Lemma 2. *Let $T \geq 1$ be a natural number. We define the variables x_{ti}, y_{ti} for all $t \in \{0, \ldots, T\}$ and $i \in [1; n]$. Consider the polytope $LP(T)$ given by the following set of constraints:*

$$y_{0i} = 1, \quad \forall i \in [1; n] \tag{6}$$

$$y_{Ti} = 0, \quad \forall i \in [1; n] \tag{7}$$

$$x_{ti} + y_{t-1,i} = 1, \quad \forall t \in [1; T], i \in [1; n] \tag{8}$$

$$y_{t-1,i} - y_{ti} \geq 0, \quad \forall t \in [1; T], i \in [1; n] \tag{9}$$

$$\sum_{i \in S_{old}} x_{ti} + \sum_{y \in S_{new}} y_{ti} \geq 1, \quad \forall t \in [1; T], \forall S \in \mathcal{S} \tag{10}$$

Then, the constraints can be satisfied by integral variables iff there is a schedule with at most T rounds for the corresponding instance (n, σ) of the RLF problem.

Proof. The proof is analogue to the proof of Lemma 1. It is easy to check that the new cycle breaking constraints (10) model the RLF property. □

To find a feasible integral point (or show that no such point exists) in polynomial-time, we can again use the Floyd-Warshall algorithm. Unlike in the SLF case, where Floyd-Warshall is used to find the shortest cycle, here it is used to find the shortest *path-cycle structure*. To do that, it suffices to compute for all nodes i the length l_i of the shortest cycle C_i containing i, and the length l'_i of the shortest path from the source s to the node i (with regards to the distances given by x_{ti} for old edges and y_{ti} for new edges). If, for all nodes i, it holds that $l_i + l'_i \geq 1$, then the cycle breaking constraints are fulfilled. Otherwise, if for some i it holds that $l_i + l'_i < 1$, then we can easily extract the set $S \in \mathcal{S}$ which violates the corresponding constraint.

For the implementation, we proceed as in the SLF case by using *lazy cycle-breaking*. Initially, no *path-cycle* constraints are added. If there exists no feasible integral point the number of rounds T is increased. Else, Floyd-Warshall is used to find the shortest *path cycle structure* S. If the cost of S is smaller than 1 we add the corresponding constraint and retry searching for a feasible point. Otherwise the algorithm terminates.

IV. Polynomial-Time Heuristics

While the algorithms from the previous section solve the SLF/RLF update problems optimally, they require solving integer linear programs and their theoretical worst-case complexity is not in polynomial-time. In this section we explore even faster, polynomial-time algorithms that solve the update problems close to optimally in practice.

A. LP-based Heuristics

This section applies to both the SLF and RLF variants of the loop-free update problem. We consider, for any integer $T \in [1; n]$, the polytope $LP(T)$ from either Lemma 1 or 2. Instead of solving these problems integrally like in the previous section, we instead instruct Gurobi to provide an optimal *fractional* solution, which is significantly faster.

Assuming we obtained an optimal feasible (fractional) solution y^* of $LP(T)$ for a minimal T, the next question is how to round this solution to make it integral in polynomial-time. Let OPT be the optimal/minimal number of rounds obtained by an integral solution and note that the (minimal) T fulfills the inequality $T \leq OPT$. Assume y^* listed as a $(T + 1) \times n$ matrix Y^*, where the rows match to time/rounds, and the columns match to vertices. We denote with y^*_t the vectors $(y^*_{ti})_{i \in [1;n]} \in \mathbb{R}^n$. i.e. the rows of Y^*. Let $2 \leq k \leq T$; we keep as invariant the property that the rows $1, \ldots, k-1$ are integral. Observe that, if k reaches T, then the whole solution is integral, because the last row of Y^* is the zero vector. We round y^*_k by adding the constraints $y^*_{ki} = 1$ if $y^*_{ki} > 0$

and $y_{ki} = 0$ if $y^*_{ki} = 0$. We then search for another feasible solution (fulfilling the newly added integrality conditions) in the restricted $LP(T)$. If such a solution does not exist, T is increased (without removing any of the previously added integrality constraints) and we retry finding a feasible solution.

B. Other Heuristics for RLF

In this section we concentrate on the RLF variant of the loop-free update problem. A general framework allows to explore additional heuristics for this particularly interesting problem. It partially uses notions already introduced in [8], such as node merging, but we also introduce new concepts to ensure a unified approach.

Definition 6 (Node merging)**.** Let G be the *full graph* of some instance (n, σ) of the RLF problem. For any node $i \in [1; n-1]$, let $out(i) := \sigma(\sigma^{-1}(i) + 1)$ be the successor of i w.r.t. the new path. Updating i, i.e. *merging* i with $out(i)$ in the full graph, amounts to replacing i and $out(i)$ by a single node, with incoming rules from both i and $out(i)$ and outgoing rules just from $out(i)$. Intuitively, merging i with $out(i)$ models the fact that, once updated, i only delegates incoming packets to its new successor $out(i)$.

Definition 7 (Update Tree)**.** Let $S \subseteq [1; n-1]$ be a subset of already updated nodes for an instance (n, σ) of the RLF problem with *full graph* G. We represent this in the full graph by merging all nodes $i \in S$ with their corresponding successors $out(i)$. Note that if both i and $out(i)$ are merged, then they will both be merged into $out(out(i))$. It is easy to observe that, independently of the order in which the nodes from S are merged, the same *update tree* $G(S)$ is obtained. $G(S)$ is a tree with regards to the remaining old edges (hence the name), while the remaining new edges form a permutation over its nodes.

We can visualize the concepts introduced by the previous two definitions in Figure 2. In the first round, nodes 2 and d are merged. In the second round, node 3 is merged to node 5, which is in turn merged with node 4, which is in turn merged with node $(2, d)$. This leads to the update tree on the bottom, which is in fact the full graph corresponding to the only instance of the RLF problem with $n = 2$ nodes. Other examples of node merging are available in [8]. The following lemma contains the unifying idea behind Peacock and also our new algorithm. We include the (straightforward) proof in a future technical report.

Lemma 3 (Main Lemma)**.** *Let G be the full graph of an instance (n, σ) of the RLF problem. Let P be a path in G from the source 1 to the destination n. Let P_{old} and P_{new} be the nodes in P which use their old and, respectively, their new outgoing edge to form the path P. Let $S := [1; n-1] \setminus P_{old}$ and consider the update tree $G(S)$. Then $G(S)$ is the full graph of an instance (k, τ) of the RLF problem, where $k := |P_{old}| + 1$ and τ is a permutation over the nodes in $P_{old} \cup \{n\}$*

According to Lemma 3, constructing the update tree $G(S)$ for the particular choice of $S := [1; n-1] \setminus P_{\text{old}}$ gives another instance of the RLF problem, to which the original instance is reduced. In the following section we make implicit use of this fact to successively reduce the original RLF instance.

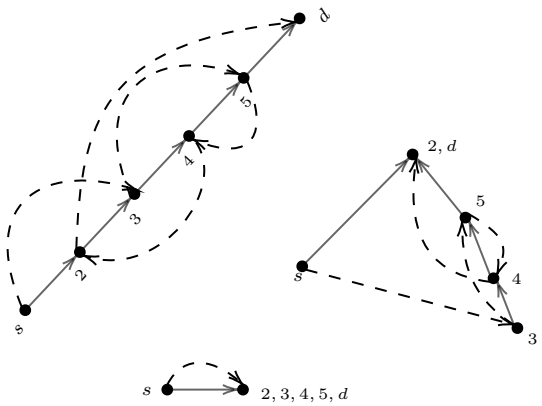

Fig. 2: Running Peacock on an instance with 6 nodes. From top to bottom, we see the update trees $G(\emptyset), G(\{2\})$ and $G(\{2, 3, 4, 5\})$. Next to each node we also write out the indexes of the nodes which have been merged into it.

C. Local Search

Our local search algorithm proposed in the following builds upon concepts from the Peacock algorithm. We hence first revisit Peacock and then present our approach in detail.

In the corresponding *full graph* G of an instance with n nodes and permutation σ of the RLF problem, a *new* edge $e := (i, j) \in E_{\text{new}}$ is called *forward* iff $j > i$, i.e. if it jumps ahead w.r.t. the old path. Otherwise e is called *backward*. The node i is called *forward* or *backward node*, respectively. We provide some further definitions:

Definition 8 (Forward Path)**.** A path P from 1 to n in the *full graph* G of an instance of the RLF problem is called *forward path* if it does not contain any backward edges. Then, P is made up exclusively of old edges and of (new) forward edges.

Definition 9 (Valid Forward Edge Choice)**.** Identify each forward edge $e := (i, j)$ (with $i < j$) of a *full graph* G with the interval $[i, j]$. Let $Forw$ be the set of forward edges of G. Consider the *set of valid forward edge choices* given by:

$$\mathcal{F} := \{F \subseteq Forw : \text{ intervals corresponding to forward edges in } F \text{ share no common nodes except endpoints}\}$$

We call $F \in \mathcal{F}$ a *valid choice (of forward edges)*. If there exists no $F' \in \mathcal{F}$ such that $F \subset F'$ we call F a *maximal valid choice (of forward edges)*

Notice that there exists a *forward path* P_F containing exactly the forward edges of some subset of forward edges $F \in Forw$ if and only if we have $F \in \mathcal{F}$.

Both for the SLF and the RLF problems, it is easy to observe that one can update any forward nodes in the first round. This cannot create any cycles. However, for the RLF problem in particular, a second observation can be made: If a forward

node i with corresponding new edge $(i, j),\ j > i$ is updated in the first round, then all nodes $i < k < j$ can be safely updated in the next/second round, since they are unreachable from the source (and hence any cycles are non-problematic). Taking this thought further leads us to the following:

Lemma 4. *Let P be a forward path in the full graph G of an RLF problem instance. Then one can reduce this instance to the smaller instance $G(S)$, where $S \coloneqq [1; n-1] \setminus P_{old}$, in two rounds. Update the new forward edges in P in the first round and all other nodes from S in the second round.*

In [8], a forward path P_F is constructed starting from a *maximal valid choice of forward edges* F in the following way: first, one sorts all forward edges in descending order of their length; then, one greedily adds forward edges to the current subset F of forward edges, starting from the longest one, while keeping the invariant that F is *valid* (and hence P_F is well defined per Definition 9). It is then shown that:

Lemma 5 (see [8])**.** *The above strategy for constructing the forward path $P \coloneqq P_F$ ensures that the reduced instance $G(S)$, where $S \coloneqq [1; n-1] \setminus P_{old}$ has at most $2n/3$ nodes, where n is the size of the initial instance given by G.*

Combining Lemmas 4 and 5 then allows to conclude that Peacock requires only $O(\log n)$ rounds for any RLF problem instance. An example of Peacock running is shown in Figure 2, which was also used to understand how node merging works. The forward path P is given by $(1, 2, 6)$ and thus $S \coloneqq \{2, 3, 4, 5\}$. We update 2 in the first round and $3, 4, 5$ in the second round, as argued by Lemma 4. This leads to the new RLF problem instance given by $G(S)$, which can be solved in one round, leading to a total of 3 rounds.

Let us now elaborate on our proposed Local Search approach. Our idea for an improved algorithm is to choose the set F more carefully. We will also use a *maximal valid* such subset F, as per Definition 9, and then use the induced *forward path* P_F to reach smaller subinstances. We determine the subset F using Local Search as shown in Algorithm 1.

Note that, at Line 3, the initial choice of a set F can be made in multiple ways. Our implementation goes over the forward edges from left to right and adds them to F as long as this does not break validity (as defined in Definition 9). At Line 9, the forward edges that are to be added to ensure maximality of F' can again be chosen in any arbitrary way. For our implementation, we go left to right and add a forward edge to F' iff this does not break validity.

D. Peacock is not an $o(\log n)$ approximation

Lemma 5 implies that Peacock solves any RLF instance in $O(\log n)$ rounds. In fact, it can be shown that this bound is tight:

Theorem 1 ([8])**.** *There exists an infinite family of graphs G_j, such that G_j has $|G_j| = 2^j$ nodes for any $j \geq 1$, and such that running the first two rounds of Peacock on G_j produces G_{j-1}. In particular, this implies that Pecock requires $O(\log |G_j|)$ rounds to solve the instance given by G_j.*

Algorithm 1 Local Search with 1-Neighborhood

```
if n = 1 then
  return 1
F ∈ 𝓕 some initial maximal valid choice of forward edges
P ≔ P_F and corresponding S ≔ [1; n − 1] \ P_old
r(S) ← LocalSearch(G(S)) (recursive call)
while true do
  for f ∉ F do
    F' ← {f}
    Add all edges from F to F' that can be added without
    making F' invalid
    Add other arbitrary forward edges to F' if it is not yet
    maximally valid.
    P' ≔ P_F' with corresponding S'
    if LocalSearch(G(S')) < r(S) then
      Improve: change F to F', P to P_F', S to S'
  Break loop if no improvement was found.
return 2 + LocalSearch(G(S))
```

Note on proof. The original proof given in [8] contains some mistakes. We fix these mistakes and give a correct proof in a future technical report. □

The fact that Peacock schedules any RLF instance in $O(\log n)$ rounds also implies that Peacock is an $O(\log n)$ approximation. An open question left in [8] was whether this bound is also tight. We answer this question in the negative:

Theorem 2. *For infinitely many n, there are instances of the RLF problem for which the Peacock algorithm requires $\Omega(\log n)$ rounds, but which can be solved optimally using only constantly many rounds. Hence, Peacock is not an $o(\log n)$ approximation.*

Proof sketch (also see Figures 3, 4). Consider the graphs G_j from the previous theorem. These graphs correspond to instances of the RLF problem, and have $n = 2^j$ nodes each. The Peacock algorithm requires $\Omega(\log n)$ rounds to schedule these instances. Also, running the first two rounds of Peacock on G_j produces G_{j-1}. Now fix some j (and thus n) and construct the instance given by the graph G as shown in the upper part of Figure 3 and described in the following: In the initial path, the source node s is followed by the nodes of the graph G_j, then by a node v, then by the nodes of another graph D_j (corresponding to another/any instance of the RLF problem), and then by a node w and the destination node d. The graphs D_j and G_j both have n nodes. Let (s_1, d_1) and (s_2, d_2) be the source-destination pairs of the graphs G_j and D_j, respectively. The new path starts at s, then goes to v, then to s_1 and through the new edges in G_j eventually to d_1, then to w, to s_2 and through the new edges in D_j eventually to d_2, and then finally to d. It is easy to see that Peacock requires $\log n$ rounds to solve this instance, whereas an optimal solution updates s in the first round and all nodes inside G_j in the second round. The resulting instance can be solved in constantly many rounds. □

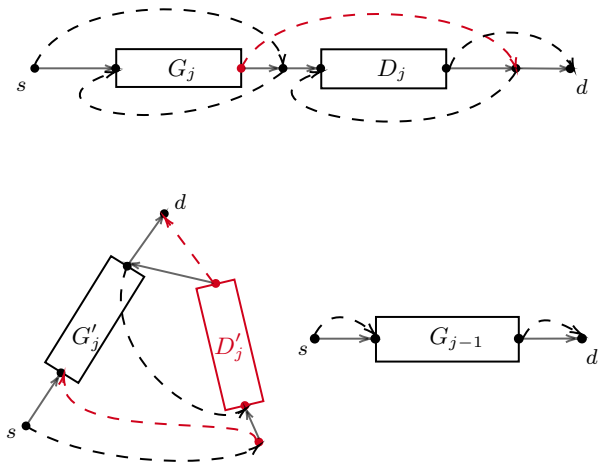

Fig. 3: Running first two rounds of Peacock on our instance leads to a situation where $j-1 = \log n$ rounds will be required. Grey full edges correspond to the old path; black or red dashed edges correspond to the new path. Red elements are updated in the following shown round.

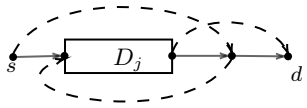

Fig. 4: Configuration reached by a RLF solution which updated s in the first round and then all nodes inside G_j in the second round.

V. Empirical Evaluation

In this section, we report on the results from our empirical evaluation, studying the running times and approximation quality. We implemented all presented algorithms and ran simulations on an Intel i5-7200U Processor (4 cores) with 8GB RAM running Debian Stable 11 (Bullseye). Our code is publicly available at [9]. We include instructions for reproducing the results presented in this paper. The main logic is implemented in C++. To solve linear programs, either fractionally or integrally, we use the library Gurobi [10] in Java.

A. Running Times

We first report on the running times of our algorithms. In [8] a repository is provided which includes an ILP solver for different variants of the SDN Network update problem, including (but not limited to) the SLF and RLF problems. The ILP is adapted from [11]. It can be used to determine the (integral) optimum OPT for any instance of these problems. To the best of our knowledge, this is the only publicly available algorithm based on linear programming which precedes our techniques. It is also the only tool we know besides ours that can exactly solve large instances of SLF/RLF in a reasonable amount of time.

TABLE I: Comparison with state-of-the-art exact algorithm for SLF and RLF.

Size of instance	Number of samples	[8] SLF	New Exact SLF
60	100	942s	42s
70	80	1145s	73s
80	60	1616s	78s
90	40	1671s	122s
100	30	1135s	188s
110	30	1683s	405s

Size of instance	Number of samples	[8] RLF	New Exact RLF
60	50	1610s	110s
70	40	1948s	248s
80	30	2124s	527s
90	20	2242s	1118s
100	20	2816s	4154s
110	10	4014s	6185s

From the results in Table I we can see that the speed of this state-of-the-art ILP solver is slower compared to our algorithms. To analyze the running times we used randomly generated instances/permutations. The first two columns indicate the size of the randomly generated instances/permutations and the number of samples generated for this size, respectively. The following columns contain the number of seconds required by each of the algorithms to solve all of these instances. For the SLF problem our implementation is usually faster by a factor of 10 to 20 for instances with less than 90 nodes, which are relevant network sizes in practice. The benefit however lowers for larger networks. For the RLF problem again the speed difference is significant in the favor of our algorithms for reasonably sized instances with less than 90 nodes. We credit the good performance of our solver to our lazy cycle-breaking technique. Indeed, without the use of lazy cycle-breaking, i.e. by writing down all cycle constraints in Gurobi, our algorithms usually ran into memory limitations for instances with more than 40 nodes.

Regarding the RLF heuristics, Table II provides a summary of their performance. It is apparent that using the Linear Programming approach to solve the RLF instances exactly is much slower than using the corresponding rounding approximation algorithm. In particular, exactly solving large RLF instances, e.g. with more than 100 nodes, requires several hours and our experiments hence timed out. Regarding the other RLF heuristics, Peacock runs in under 1 second for all experiments (not shown in the table) but Local Search is also significantly faster than any of the LP-based approaches. Later we will see that Local Search offers the best approximation quality, improving on Peacock and showing that it provides an attractive trade-off between accuracy and speed.

TABLE II: Running times of different RLF algorithms. A hyphen indicates a running time of over one hour.

Size of instance	Number of samples	LP Exact RLF	LP Round RLF	Local Search RLF
60	200	442s	136s	5s
70	100	756s	134s	4s
80	75	2192s	154s	3s
90	50	2786s	111s	4s
100	50	-	292s	6s
110	50	-	303s	7s

TABLE III: RLF: Empirical approximation factors.

	Maximum difference	Maximum factor	Mean difference	Mean factor
Rounding ($n = 70$)	7	2.75	1.29	1.35
Peacock ($n = 70$)	4	2.33	1.26	1.37
Local Search ($n = 70$)	2	1.66	0.74	1.2
Rounding ($n = 85$)	5	2.25	0.93	1.24
Peacock ($n = 85$)	4	2.33	1.36	1.41
Local Search ($n = 85$)	2	1.66	0.66	1.18

B. Approximation Quality

We next report on our results on the approximation quality. In Figure 5 we consider the SLF algorithms. On the top of Figure 5 we compare the three quantities LP_{OPT}, i.e. the minimal T for which the polytope $LP(T)$ is (fractionally) feasible, OPT, i.e. the optimal/minimal number of rounds for an integral solution, and the solution ALG provided by the SLF rounding algorithm, i.e. the number of rounds the rounded integral solution y^* uses. This boxplot has been produced by running the algorithms on randomly generated permutations/instances of the SLF problem. Notice that LP_{OPT} and OPT are usually very close to each other. Also, for most instances OPT and LP_{OPT} stabilize at around 4 rounds. The rounded solution ALG is usually close to OPT, but outliers are present too, requiring 8 and 10 rounds to solve, respectively. Similar experiments for other instance sizes show almost identical results.

At the bottom of Figure 5 we show how the mode and the mean of the empirical approximation factor behave for different values of n. The data has been produced by running 50 simulations on randomly generated permutations/instances of the SLF problem for each $n = 60, 65, \ldots, 110$. We kept the number of simulations relatively small for performance reasons. This leads to quite high variance. In particular, there is an outlying maximal approximation factor 6.5 obtained for $n = 85$. Such spikes are relatively common for the SLF problem: most randomly generated instances are well approximated, and instances for which the approximation factor is high are quite rare. The worst-case empirical approximation factor ALG/OPT seems to get progressively worse, both in mode and in mean, as we increase the size n of the instances. While it is in principle possible that for higher n the approximation factor stabilizes/converges to a constant, we suspect that our rounding procedure can behave poorly in pathological cases, even though it is usually competitive.

In Figure 6 we next consider the RLF algorithms. As for the

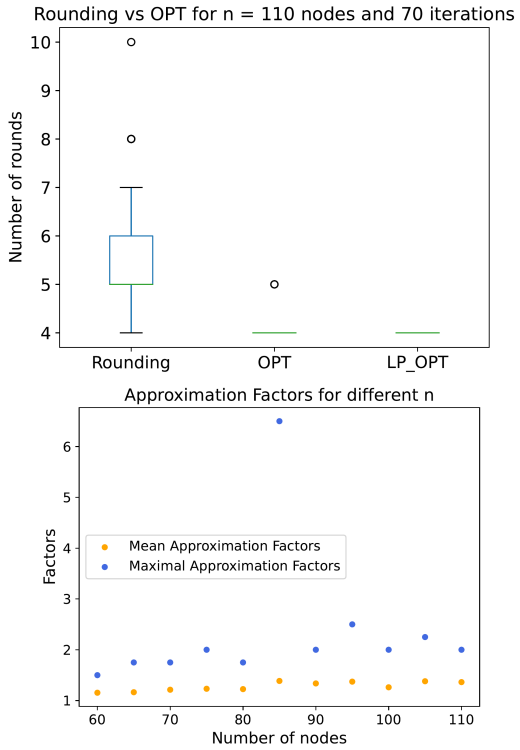

Fig. 5: SLF: Comparison of Rounding with OPT

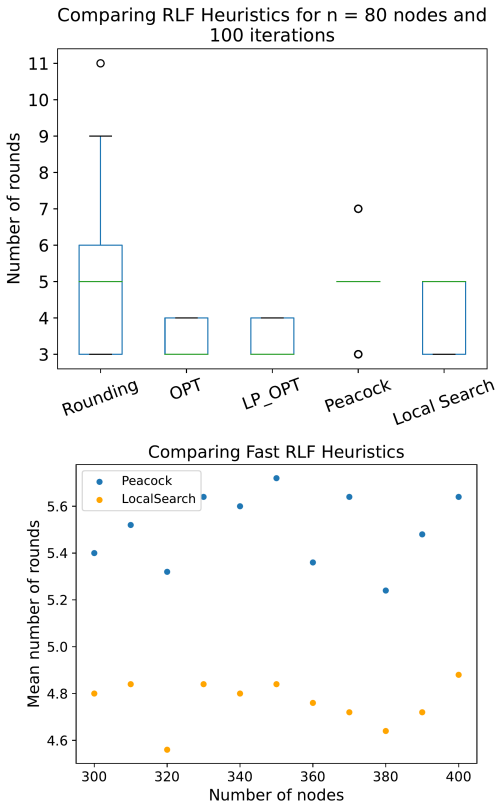

Fig. 6: RLF: Comparison of Heuristics with OPT

SLF problem, on the top of Figure 6, the quantities LP_{OPT} and OPT have a narrow distribution and are almost always equal to each other; for most instances only 3 or 4 rounds are required. The LP Rounding Algorithm has a broad distribution with long upper whiskers and outliers. Peacock solves almost all instances in 5 rounds, but outliers exist at 3 and 7 rounds, respectively. In all cases, the Local Search algorithm seems to perform the best: all its results fit compactly in a small box which is almost as low as the boxes of LP_{OPT} and OPT. There are no whiskers or outliers. In Table III we gather some information regarding the empirical approximation factors. For space reasons we only show the results for $n = 70$ and $n = 85$. The results for other values of n are very similar and confirm that Local Search is the best performing approximation.

At the bottom of Figure 6 we compare the non-LP based heuristics for larger values of n. (The LP cannot be used for such big instances.) For each value of $n = 300, 310, \ldots, 400$ we sample 50 random permutations and compute the mean number of rounds required to solve these instances for all algorithms. The results confirm the impressions from above: Local Search finds the shortest schedules.

VI. Related Work

There already exists a large body of literature surrounding the broad question of how to update policies and routes in (software-defined) networks [3], [5], [12]–[20]. For comprehensive surveys of recent literature in the area of consistent network updates we refer the reader to [1], [2].

Reitblatt et al. [3] initiated the study of network updates using a 2-Phase Commit Protocol (2PC) with packet tagging. The protocol also served as a foundation for distributed control plane implementations [13]. However, packet tagging is often undesirable and can lead to unexpected issues as well [4].

In [5], [21], Wattenhofer et al. studied transient consistency properties, including strong loop-freedom (SLF), for destination-based routing policies. However, their model uses a different objective function, where the number of nodes which can be updated in a single round needs to be *maximized*, instead of *minimizing* the number of rounds. The paper [7] also analyzes this alternative optimization problem and provides a proof of NP-hardness in the case of SLF. The problem we are interested in, i.e. of *minimizing* the number of rounds, is also proven to be NP-hard in the SLF case [6], while for RLF this remains an open question. The objective of *minimizing* the number of rounds was first studied by Schmid et al. in [7], [8]. They show that this problem is NP-hard for SLF but it remains open if this also holds for the RLF case. In [8] the Peacock algorithm is introduced, which can solve any RLF instance using $O(\log n)$ rounds where this is a tight bound [8]. In the more recent work [22] the stronger result is proven that there exist RLF instances which cannot be scheduled in less than $\Omega(\log n)$ many rounds (with *any* algorithm). Researchers have also investigated how to efficiently update multiple policies simultaneously [23].

Several previous exact algorithms for the SLF and/or RLF problems are based on Petri Networks [24], [25], Linear Temporal Logic [26], [27], Stackelberg Games [28], BDDs [29], [30] or other techniques [31]. These tools can be used to enforce other consistency guarantees than loop-freedom and are thus more general yet slower than more specialized techniques. Examples of such tools include the recent Kaki [25], Netstack [28], Latte [32] and Netsynth [26].

In [8] an exact algorithm, based on an integer linear programming formulation from [11], is used specifically for the SLF/RLF problems. However, even the aforementioned integer linear program [11] is significantly more complex than our formulation, and leads to slower performance than our (more specialized) exact algorithms. Our paper provides, as far as we know, the first exact algorithms designed specifically for the SLF and RLF problems. In the case of RLF there has also been a line of research to design heuristics, like Peacock [8] or the more recent Savitar [22].

VII. Conclusion

We presented both exact and approximate algorithms for the strong and relaxed loop-free network update problems. In particular, we presented a parameterized LP which allows (after our optimizations) to solve the problems both exactly (using an ILP formulation) and approximately (using an LP formulation and a suitable rounding algorithm) for a significant number of nodes. Our algorithms provide a way to compute the optimum much faster than with any other method known to us for instances of reasonable size. We further find that for the RLF problem, which relaxes the SLF problem, if only an approximation is required, the best-performing heuristic is the Local Search algorithm. This heuristic improves on the state-of-the-art Peacock algorithm in that it explicitly searches for the best set of forward edges to update. We implemented all algorithms and published our code to ensure that our results are reproducible and falsifiable.

Our work opens several interesting avenues for future research. On the theory front, we showed that Peacock is not an $o(\log n)$-approximation: we gave an example where Peacock requires $\Theta(\log n)$ rounds though it could be solved in $O(1)$ rounds. It is easy to see that our Local Search algorithm actually finds a schedule with $O(1)$ rounds for these instances. Hence, it would be interesting to study whether Local Search provides a constant approximation. On the practical front, it would be interesting to explore algorithm engineering approaches to further improve the performance of our algorithms, and generalize them for additional consistency properties. The authors have provided public access to their code and/or data at [9].

Acknowledgments

Research supported by German Research Foundation (DFG), grant 470029389 (FlexNets), 2021-2024, by the Vienna Science and Technology Fund (WWTF) project 10.47379/ICT19045 (WHATIF), 2020-2024. and by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number MB22.00054.

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of Consistent Software-Defined Network Updates," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1435–1461, 2019.

[3] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," ser. SIGCOMM '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 323–334.

[4] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul, "FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions." Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14), 08 2013, pp. 19–24.

[5] R. Mahajan and R. Wattenhofer, "On Consistent Updates in Software Defined Networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII. New York, NY, USA: Association for Computing Machinery, 2013.

[6] S. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid, "Transiently Consistent SDN Updates: Being Greedy is Hard," in *International Colloquium on Structural Information and Communication Complexity*, 07 2016, pp. 391–406.

[7] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling Loop-Free Network Updates: It's Good to Relax!" in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, 2015, p. 13–22.

[8] K.-T. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid, "Loop-Free Route Updates for Software-Defined Networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, p. 328–341, 2018.

[9] RaduVintan, "Raduvintan/slf_rlf_solver: slf_rlf_solver," Jan. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10458478

[10] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[11] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently Secure Network Updates," *SIGMETRICS Perform. Eval. Rev.*, 2016.

[12] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," in *Proceedings of the 2014 ACM Conference on SIGCOMM*. Association for Computing Machinery, 2014.

[13] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A Distributed and Robust SDN Control Plane for Transactional Network Updates," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[14] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "ZUpdate: Updating Data Center Networks with Zero Loss," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. Association for Computing Machinery, 2013.

[15] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good Network Updates for Bad Packets Waypoint Enforcement Beyond Destination-Based Routing Policies," in *13th ACM Workshop on Hot Topics in Networks (HotNets)*, 2014.

[16] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis." USENIX Association, 2013.

[17] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2013.

[18] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Association for Computing Machinery, 2007.

[19] K.-T. Foerster and S. Schmid, "Distributed consistent network updates in sdns: local verification for global guarantees," in *2019 IEEE 18th international symposium on network computing and applications (NCA)*. IEEE, 2019, pp. 1–4.

[20] J. Zheng, B. Li, C. Tian, K.-T. Foerster, S. Schmid, G. Chen, and J. Wux, "Scheduling congestion-free updates of multiple flows with chronicle in timed sdns," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 12–21.

[21] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes," in *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, 2016.

[22] H. Zhou, X. Gao, J. Zheng, and G. Chen, "A Tight Lower Bound for Relaxed Loop-Free Updates in SDNs," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[23] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 133–143.

[24] M. Didriksen, P. G. Jensen, J. F. Jønler, A.-I. Katona, S. D. L. Lama, F. B. Lottrup, S. Shajarat, and J. Srba, "Automatic Synthesis of Transiently Correct Network Updates via Petri Games," in *Application and Theory of Petri Nets and Concurrency*, 2021.

[25] Johansen, Nicklas S. and Kær, Lasse B. and Madsen, Andreas L. and Nielsen, Kristian Ø. and Srba, Jiří and Tollund, Rasmus G., "Kaki: Concurrent Update Synthesis for Regular Policies via Petri Games," in *Integrated Formal Methods*. Springer International Publishing, 2022, pp. 249–267.

[26] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient Synthesis of Network Updates," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[27] T. Schneider, R. Birkner, and L. Vanbever, "Snowcap: Synthesizing Network-Wide Configuration Updates," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 33–49.

[28] S. Schmid, B. C. Schrenk, and A. Torraiba, "NetStack: A Game Approach to Synthesizing Consistent Network Updates." IFIP Networking Conference, 2022.

[29] K. G. Larsen, A. Mariegaard, S. Schmid, and J. Srba, "Allsynth: A bdd-based approach for network update synthesis," *Science of Computer Programming*, vol. 230, p. 102992, 2023.

[30] ——, "Allsynth: Transiently correct network update synthesis accounting for operator preferences," in *International Symposium on Theoretical Aspects of Software Engineering*. Springer, 2022, pp. 344–362.

[31] S. Vissicchio and L. Cittadini, "FLIP the (Flow) Table: Fast Lightweight Policy-preserving SDN Updates," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016.

[32] N. Christensen, M. Glavind, S. Schmid, and J. Srba, "Latte: Improving the Latency of Transiently Consistent Network Update Schedules," *SIGMETRICS Perform. Eval. Rev.*, vol. 48, no. 3, p. 14–26, mar 2021.