Scheduling Relaxed Loop-Free Updates Within Tight Lower Bounds in SDNs

Hao Zhou, Xiaofeng Gao, *Member, IEEE*, Jiaqi Zheng, *Member, IEEE, ACM*, and Guihai Chen, *Member, IEEE*

***Abstract*— We consider a fundamental update problem of avoiding forwarding loops based on the node-ordering protocol in Software Defined Networks (SDNs). Due to the distributed data plane, forwarding loops may occur during the updates and influence the network performance. The node-ordering protocol can avoid such forwarding loops by controlling the update orders of the switches and does not consume extra flow table space overhead. However, an $\Omega(n)$ lower bound on the number of rounds required by any algorithm using this protocol with loop-free constraint has been proved, where n is the number of switches in the network. To accelerate the updates, a weaker notion of loop-freedom — relaxed loop-freedom — has been introduced. Despite that, the theoretical bound of the node-ordering protocol with relaxed loop-free constraint remains unknown yet. In this article, we solve a long-standing open problem: how to derive $\omega(1)$-round lower bound or to show that $O(1)$-round schedules always exist for the relaxed loop-free update problem. Specifically, we prove that any algorithm needs $\Omega(\log n)$ rounds to guarantee relaxed loop freedom in the worst case. In addition, we develop a fast relaxed loop-free update algorithm named Savitar that touches the tight lower bound. For any update instance, Savitar can use at most $2\lfloor\log_2 n\rfloor - 1$ rounds to schedule relaxed loop-free updates. Extensive experiments on Mininet using a Floodlight controller show that Savitar can significantly decrease the update time, achieve near optimal performance and save over 30% of the rounds compared with the state of the art.**

***Index Terms*— Software defined networks, relaxed loop freedom, network updates.**

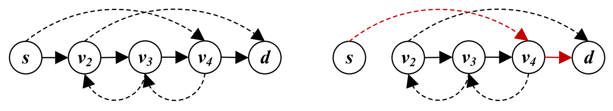
I. Introduction

SOFTWARE defined networking (SDN) [12] enables flexible and global network management by decoupling the control plane from the data plane and integrating the control plane into logically centralized controllers. Compared to the static architecture of traditional networks, the centralized intelligence of the controllers in SDNs can contribute to dynamic and more efficient network configurations to improve the network performance. Consequently, more enterprises such as Microsoft SWAN [14] and Google B4 [15] have started trying to use SDN to manage their data center networks.

Despite the centralized control plane, the data plane remains distributed. Network updates are associated with network availability and performance in traffic engineering and are frequently scheduled when network conditions change, e.g., switch upgrade, link failures or traffic variations. Due to the unpredictable update orders of the switches in the data plane, network updates may cause inconsistency and introduce forwarding loops that consist of a mixture of old rules and new rules into networks. Packets entering loops cannot be forwarded out until the loops are broken, which may result in packet drops and further influence the performance of delay-sensitive applications.

Avoiding such forwarding loops can guarantee the network correctness and availability during the updates. However, given the initial route and the final one, scheduling loop-free updates remains algorithmically challenging. The node-ordering protocol is a major mechanism to guarantee loop-freedom without incurring extra flow table space overhead. This protocol partitions the switches that need to be updated into multiple disjoint subsets, and requires that the updates of each subset of the switches will not introduce any loops. Therefore, the whole network updates can be completed by scheduling the updates of each subset one by one. Actually, such an update of one subset is called one *round* and hence the number of subsets is called the number of rounds, which can reflect the update time to some degree. On the one hand, a loop-free update algorithm should guarantee the correctness, i.e., compute how to partition the switches to avoid forwarding loops. On the other hand, this algorithm should minimize the number of rounds in order to schedule loop-free updates quickly.

Strong and relaxed loop-freedom are two different definitions for loop-freedom. Strong loop-freedom requires that no loop should occur at any time during updates. In Fig. 1(a),



(a) Example for strong loop-freedom (b) Example for relaxed loop-freedom

Fig. 1. Two notions of loop-freedom. The solid lines represent the old routing path op and the dashed lines represent the new routing path np.

Manuscript received November 10, 2019; revised June 4, 2020; accepted July 17, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. Giaccone. Date of publication August 26, 2020; date of current version December 16, 2020. This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB2102200; in part by the National Natural Science Foundation of China under Grant 61872238, Grant 61802172, and Grant 61972254; in part by the CCF-Huawei Database System Innovation Research Plan under Grant CCF-Huawei DBIR2019002A; and in part by the CCF-Tencent Open Research Fund. *(Corresponding author: Xiaofeng Gao.)*

Hao Zhou, Xiaofeng Gao, and Guihai Chen are with the Shanghai Key Laboratory of Scalable Computing and Systems, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: h-zhou@sjtu.edu.cn; gao-xf@cs.sjtu.edu.cn; gchen@cs.sjtu.edu.cn).

Jiaqi Zheng is with the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China (e-mail: jzheng@nju.edu.cn).

Digital Object Identifier 10.1109/TNET.2020.3017771

we notice that v_2 should be updated before v_3 and v_3 should be updated before v_4 in order to preserve strong loop-freedom. The work [20] first proved that a strong loop-free update sequence can be generated by updating one switch in each round along the reverse of the new routing path. However, the number of rounds depends on the number of switches in this path. An improved solution was introduced in [10] which greedily updated a maximal subset of switches that will not introduce any loops in each round. Unfortunately, it has been proved that this greedy algorithm may produce $\Omega(n)$-round schedules for instances that can actually be solved in $O(1)$ rounds, where n is the number of switches in the network [8], [19]. In addition, the inherent hardness for strong loop-freedom is that any algorithm requires $\Omega(n)$ rounds in the worst case [8], [19]. In contrast, relaxed loop-freedom allows the existence of certain types of loops. Specifically, if the loops do not occur on the path from source to destination during updates, these cases can be accepted since they cannot be reached by the source and are not very harmful in practice [8], [19]. Taking Fig. 1(b) as an example, if s has been updated, v_2 and v_3 can be updated in the same round in order to preserve relaxed loop-freedom because the source s cannot reach the loop $v_2 \rightarrow v_3 \rightarrow v_2$ at this time. Therefore, fewer rounds are required in order to schedule relaxed loop-free updates compared with strong loop-freedom. Indeed, the works [8], [19] proposed an algorithm named Peacock and claimed that it can schedule relaxed loop-free updates in $O(\log n)$ (more precisely at most $\lceil 6 \log_2 \ n \rceil$) rounds.

Due to the great advantages compared with strong loop-freedom, relaxed loop-freedom has drawn much attention in recent years. However, a crucial problem in relaxed loop-freedom is whether $O(1)$-round schedules always exist or how to derive $\omega(1)$-round lower bounds, but it remains unsolved yet. In addition, besides the theoretical result, Peacock [8], [19] does not perform well in practice and the experiments show that it is far from the optimal solution and only achieves a slight improvement compared with the greedy algorithm, while the latter provides a stronger guarantee of loop-freedom (strong loop-freedom). Therefore, how to design a faster relaxed loop-free update algorithm in theory or practice is especially important and remains technically challenging. In this article, we attempt to answer these problems.

Our contributions: Our first contribution is that we provide an $\Omega(\log n)$-round lower bound for the relaxed loop-free update problem. We construct a special update instance to prove that any algorithm requires $\Omega(\log n)$ rounds to guarantee relaxed loop-freedom in the worst case, which closes the long-standing open problem whether $O(1)$-round schedules always exist or not. In order to derive a rigorous proof, we have to analyse all the possible intermediate states during updates for this special update instance. We claim that an optimal intermediate state can be obtained by making some specific decisions in each step and the optimal solution also requires $\Omega(\log n)$ rounds to guarantee relaxed loop-freedom for this update instance.

Our second contribution is that we develop a fast relaxed loop-free update algorithm — Savitar — that can touch the tight lower bound and is faster than prior works both in theory and practice. In particular, we prove that Savitar uses at most $2\lfloor \log_2 \ n \rfloor - 1$ rounds to guarantee relaxed loop-freedom for any update instance, which achieves a constant-factor improvement over Peacock ($6\lceil \log_2 \ n \rceil$ rounds) and touches the tight lower bound in theory.

Our third contribution is a concrete implementation and extensive simulations for our algorithm. We conduct a prototype implementation on Mininet by using a Floodlight controller. Extensive experiments show that Savitar can significantly decrease the update time, achieve near optimal performance and save over 30% of the rounds compared to the state of the art.

This article extends the initial study [32], via (i) presenting a complete proof for Theorem 1 in Sec. II; (ii) adding more description and comparison for the hard update instance and adjusting the structure of the proof of Theorem 1 to help understand the proof process in Sec. III; (iii) analysing the theoretical performance of Algorithm 1 in Sec. IV; (iv) performing more comparison experiments with the state-of-the-art in Sec. V; (v) surveying up-to-date literature and summarizing more related works about consistent route updates in Sec. VI; and (vi) improving the organization and presentation of the paper by a major revision and careful proofreading.

II. Network Model

A. Basic Definitions

Since our paper is an extension of the works [8], [19], we overview some essential definitions in [8], [19] for a better understanding of our paper in this subsection.

We are given two simple directed paths, the old path op and the new path np, which represent the old route and the new route, respectively. All the route rules (the old edges) in op should be updated to the new rules (the new edges) in np. We call the pair (op, np) an *update instance*. However, unreasonable update orders of these nodes may cause a transient loop. For example, if v_3 is updated before v_2, there will be a transient loop $v_2 \rightarrow v_3 \rightarrow v_2$ in Fig. 1(a). Without loss of generality, we assume that op and np have the same node set in this article, denoted by U. Denote by s the source node and d the destination node. Let $U_1, U_2, \ldots, U_m$ be a partition of $U \backslash d$, where U_i denotes the node set that is updated in the i-th round. Such a partition $U_1, U_2, \ldots, U_m$ is called an *update sequence* for the instance (op, np). Let $\mathcal{G}_i = (U, E_i)$ be the graph obtained from op after all nodes in $U_1, \ldots, U_i$ have been updated. Based on the above, we present the formal definitions of strong loop-freedom and relaxed loop-freedom as follows. More notations can be found in Table I.

Definition 1 (Strong Loop-Freedom): An update sequence $U_1, U_2, \ldots, U_m$ satisfies strong loop-freedom if $\mathcal{G}_i \cup \mathcal{G}_{i+1}$ does not contain a cycle for any $0 \leq i < m$, where $\mathcal{G}_0 = op$ and $\mathcal{G}_m = np$.

Definition 2 (Relaxed Loop-Freedom): Let $\mathcal{G}^i_{i+1}$ be a subgraph of $\mathcal{G}_i \cup \mathcal{G}_{i+1}$, which only contains the nodes and the edges that the source node s can reach. An update sequence $U_1, U_2, \ldots, U_m$ satisfies relaxed loop-freedom if $\mathcal{G}^i_{i+1}$ does not

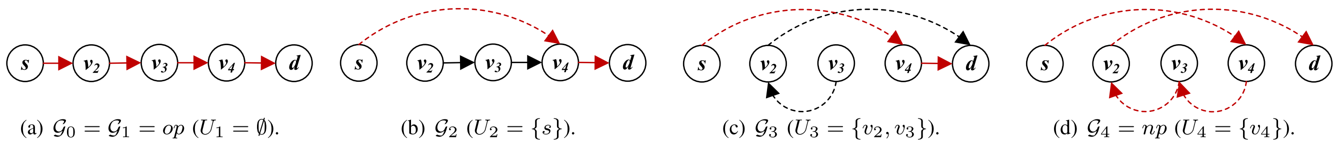

s
v2
v3
v4
d
(a) $\mathcal{G}_0 = \mathcal{G}_1 = op$ ($U_1 = \emptyset$).
(b) $\mathcal{G}_2$ ($U_2 = \{s\}$).
(c) $\mathcal{G}_3$ ($U_3 = \{v_2, v_3\}$).
(d) $\mathcal{G}_4 = np$ ($U_4 = \{v_4\}$).

Fig. 2. An update example. OPT-HP $= \{hp_1\}$ where $hp_0 = op, hp_1 = s \rightarrow v_4 \rightarrow d$ and $hp_2 = np$. Let us follow the process in the proof of Lemma 1 to construct a relaxed loop-free update sequence $\mathrm{R} = \{U_1, U_2, U_3, U_4\}$. According to Equation (1), when $i = 0$, we have $U_1 = \emptyset$ because of $hp_1 \backslash hp_0 = \emptyset$, and $U_2 = \{s\}$ because of $s \in hp_1 \cap hp_0$ and $(s, v_4) \in np \cap hp_1$; When $i = 1$, we have $U_3 = \{v_2, v_3\}$ because of $v_2, v_3 \in hp_2 \backslash hp_1$ and $(v_2, d), (v_3, v_2) \in np \cap hp_2$, and $U_4 = \{v_4\}$ because of $v_4 \in hp_2 \cap hp_1$ and $(v_4, v_3) \in np \cap hp_2$. Notice that this update sequence $\mathrm{R} = \{U_1, U_2, U_3, U_4\}$ satisfies relaxed loop-freedom since we have $\mathcal{G}_1^0 = hp_0, \mathcal{G}_2^1 = hp_0 \cup hp_1, \mathcal{G}_3^2 = hp_1$ and $\mathcal{G}_4^3 = hp_1 \cup hp_2$.

TABLE I
KEY NOTATIONS

Notation	Description
s	The source node
d	The destination node
n	Number of nodes
op	The old routing path
np	The new routing path
hp_i	The i-th helper path
U	The node set, $n = \|U\|$
U_i	The node set updated in Round i
$\mathcal{G}_i$	The graph obtained from op after $U_1, \ldots, U_i$ have been updated
E_i	The edges in $\mathcal{G}_i$
$\mathcal{G}_{i+1}^i$	The subgraph of $\mathcal{G}_i \cup \mathcal{G}_{i+1}$ containing the nodes and edges that s can reach
$\mathcal{I}_k$	The hard update instance constructed in Sec. III
$\mathcal{I}_k^i$	The update instance constructed based on $\mathcal{I}_k$

contain a cycle for any $0 \leq i < m$, where $\mathcal{G}_0 = op$ and $\mathcal{G}_m = np$.

In Fig. 1(a), suppose that v_3 is updated in the t-th round U_t. If v_2 is not updated before v_3, we have $(v_2, v_3) \in \mathcal{G}_{t-1}$ and $(v_3, v_2) \in \mathcal{G}_t$. Therefore, a cycle $v_2 \rightarrow v_3 \rightarrow v_2$ is included in $\mathcal{G}_{t-1} \cup \mathcal{G}_t$. As is shown by Fig. 1(b), if s has been updated in U_1, then v_2 and v_3 can be updated in the same round U_2 for relaxed loop-freedom because $\mathcal{G}_2^1 = s \rightarrow v_4 \rightarrow v_5$ at this time.

Compared to strong loop-freedom, it requires fewer rounds to preserve relaxed loop-freedom for an update instance. In addition, loops are not very harmful if they cannot be reached by the source node as guaranteed by relaxed loop-freedom [19]. Therefore, all of [8], [19] and our paper focus on the fast relaxed loop-free update problem.

B. Further Extensions

In this subsection, we introduce a notion of helper paths that will help us analyse the updates.

Definition 3 (Helper Path): For an update instance (op, np), a path hp is called a helper path if hp is a path from s to d and $hp \subset op \cup np$.

Definition 4 (Helper Path Sequence): For an update instance (op, np), a path sequence $hp_1, hp_2, \ldots, hp_r$ is called a helper path sequence if

1) for all $1 \leq i \leq r$, hp_i is a helper path;
2) for all $1 \leq i \leq r$, if hp_i contains a new edge (the edge in np) on some node u, then the corresponding old edge (the edge in op) on this node u will not occur in hp_j for $\forall i < j \leq r$.

Actually, Definition 4 captures a fact that the route configuration in a switch remains unchanged once it has been updated, which is also guaranteed by letting $U_1, \ldots, U_m$ be a partition in Definition 1 or Definition 2.

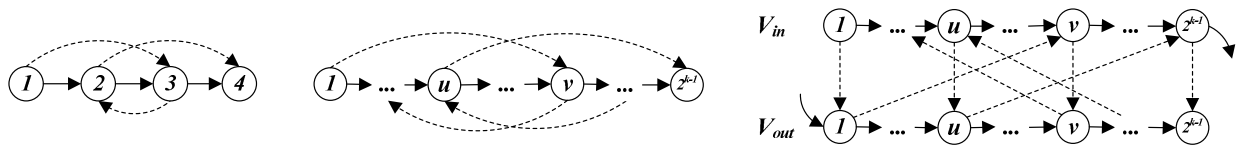
Definition 5 (Relaxed Loop-Freedom of Helper Paths): For an update instance (op, np), a helper path sequence $hp_1, hp_2, \ldots, hp_r$ satisfies relaxed loop-freedom if $hp_i \cup hp_{i+1}$ does not contain any cycles for all $0 \leq i \leq r$, where $hp_0 = op$ and $hp_{r+1} = np$.

Based on a helper path sequence that satisfies relaxed loop-freedom, we can schedule updates from hp_i to hp_{i+1} iteratively so as to produce a relaxed loop-free update sequence from op to np. Notice that here the updates from hp_i to hp_{i+1} means that the path from source to destination in the network is updated from hp_i to hp_{i+1}. Take Fig. 2 as an example. The old path op is shown in Fig. 2(a) and the new path np is shown in Fig. 2(d). Let hp_1 be $s \rightarrow v_4 \rightarrow d$ (red lines in Fig. 2(b) and Fig. 2(c)) and we know that both $op \cup hp_1$ and $hp_1 \cup np$ do not contain any cycles. In order to update op to hp_1, we can directly update s ($U_2 = \{s\}$) and get Fig. 2(b). Next, v_2, v_3, v_4 need to be updated in order to update hp_1 to np. Notice that we cannot do that in one round. For example, if let $U_3 = \{v_2, v_3, v_4\}$, $\mathcal{G}_3$ is shown as Fig. 2(d) instead of Fig. 2(c). Therefore, there will be an cycle $v_3 \rightarrow v_4 \rightarrow v_3$ that can be reached by s in $\mathcal{G}_2 \cup \mathcal{G}_3$, which violates the definition of relaxed loop-freedom. Fortunately, two-round schedules always exist for the relaxed loop-free updates from hp_i to hp_{i+1}, which will be presented in Lemma 1 with its proof. In this example, we can update hp_1 to np by letting $U_3 = \{v_2, v_3\}$ and $U_4 = \{v_4\}$ as shown in Fig. 2(c) and Fig. 2(d).

Lemma 1: For any update instance (op, np), let OPT-R $=$ *$\{U_1, \ldots, U_m\}$ be a relaxed loop-free update sequence with the minimum number of rounds and* OPT-HP $= \{hp_1, \ldots, hp_r\}$ *be a relaxed loop-free helper path sequence with the minimum number of helper paths. We have*

$$|\text{OPT-HP}| + 1 \leq |\text{OPT-R}| \leq 2|\text{OPT-HP}| + 1.$$

Proof: Let $\mathcal{G}_i$ be the graph obtained from op after all the nodes in $U_1, \ldots, U_i$ have been updated and hp'_i be a path from s to d in $\mathcal{G}_i$. From Definition 2, we have that $hp'_i \cup hp'_{i+1}$ does not contain a cycle for any $0 \leq i < m$ where $hp'_0 = op$ and



(a) $n = 4$ (b) $n = 2^{k-1}$ (c) $n = 2^k$

Fig. 3. The construction process of $\mathcal{I}_k$ using mathematical induction. The solid lines represent the old path, the dashed lines represent the new path.

$hp'_m = np$. Therefore, we get a relaxed loop-free helper path sequence $\mathrm{HP} = \{hp'_1, \ldots, hp'_{m-1}\}$. Hence, we have

$$|\text{OPT-HP}| \leq |\text{HP}| = |\text{OPT-R}| - 1.$$

In addition, we can construct a relaxed loop-free update sequence $\mathrm{R} = \{U'_1, \ldots, U'_{2r+2}\}$ based on OPT-HP. Denote by v the next node of u in the new path, i.e., $(u, v) \in np$. For $0 \leq i \leq r$, we construct U'_{2i+1} and U'_{2i+2} as follows:

$$\begin{cases} U'_{2i+1} = \{u \in hp_{i+1} \backslash hp_i : (u, v) \in hp_{i+1}, u \notin \bigcup_{j=1}^{2i} U'_j\} \\ U'_{2i+2} = \{u \in hp_{i+1} \cap hp_i : (u, v) \in hp_{i+1}, u \notin \bigcup_{j=1}^{2i+1} U'_j\} \end{cases} \tag{1}$$

See a construction example in Fig. 2 (we use U_i to represent U'_i in Fig. 2). Next, we show that this update sequence R satisfies relaxed loop-freedom. Let $\mathcal{G}'_i$ be the graph obtained from *op* after the nodes in $U'_1, \ldots, U'_i$ are updated. Notice that

$$U'_{2i+1} \cup U'_{2i+2} = \{u \in hp_{i+1} : (u, v) \in hp_{i+1}, u \notin \bigcup_{j=1}^{2i} U'_j\}.$$

Combining the case where the path from s to d in $\mathcal{G}'_0$ is hp_0, we can obtain that for any $0 \leq i \leq r$, the path from s to d in $\mathcal{G}'_{2i}$ is hp_i based on mathematical induction. According to Equation (1), only the nodes that are not in hp_i are updated in U'_{2i+1}, the path from s to d in $\mathcal{G}'_{2i+1}$ remains hp_i. Therefore, we have $\mathcal{G}^{2i'}_{2i+1} = hp_i$ according to Definition 2 and it does not contain any cycles. Since the path from s to d in $\mathcal{G}'_{2i+2}$ is hp_{i+1}, we have $hp_i \cup hp_{i+1} \subseteq \mathcal{G}^{2i+1'}_{2i+2}$. Next, we will show that besides that in $hp_i \cup hp_{i+1}$, there is no other node and edge that can be reached by s in $\mathcal{G}'_{2i+1} \cup \mathcal{G}'_{2i+2}$ (i.e., $\mathcal{G}^{2i+1'}_{2i+2} = hp_i \cup hp_{i+1}$). Due to $hp_i \cup hp_{i+1} \subseteq \mathcal{G}^{2i+1'}_{2i+2}$, we only need to prove that all the outgoing edges on the nodes $u \in hp_i \cup hp_{i+1}$ in $\mathcal{G}'_{2i+1} \cup \mathcal{G}'_{2i+2}$ have been contained in $hp_i \cup hp_{i+1}$. The nodes in $hp_i \cup hp_{i+1}$ can be divided into three classes, i.e., $hp_i \backslash hp_{i+1}$, $hp_i \cap hp_{i+1}$ and $hp_{i+1} \backslash hp_i$. For node $u \in hp_i \backslash hp_{i+1}$, it is not updated in U'_{2i+2} according to Equation (1). Therefore, the outgoing edge on u in $\mathcal{G}'_{2i+1}$ is the same as that in $\mathcal{G}'_{2i+2}$, and it is contained in hp_i. For node $u \in hp_i \cap hp_{i+1}$, the outgoing edge on u in $\mathcal{G}'_{2i+1}$ is contained in hp_i and the one in $\mathcal{G}'_{2i+2}$ is contained in hp_{i+1}. For node $u \in hp_{i+1} \backslash hp_i$, according to Equation (1), u is not updated in U'_{2i+1} and U'_{2i+2} if the outgoing edge on u in hp_{i+1} is an old edge; Otherwise, u is updated in U'_{2i+1} or earlier rounds. Therefore, we have that the outgoing edge on u in $\mathcal{G}'_{2i+1}$ keeps the same as that in $\mathcal{G}'_{2i+2}$ and it is contained in hp_{i+1}. Based on the above, we have that $\mathcal{G}^{2i+1'}_{2i+2} = hp_i \cup hp_{i+1}$ holds and it is acyclic. Therefore, R is a relaxed loop-free update sequence. Due to $U_1 = \emptyset$, we have

$$|\text{OPT-R}| \leq |\text{R}| - 1 = 2|\text{OPT-HP}| + 1.$$

■

From Lemma 1, we can get the following two points:

(a) A lower bound for the number of helper paths is a lower bound for the number of rounds.

(b) A relaxed loop-free sequence with r helper paths means a feasible update sequence with at most $2r + 1$ rounds.

The first point brings forth Section III, where we present a lower bound for the relaxed loop-free update problem based on the analysis for helper paths. The second point brings forth Section IV, where we design a fast relaxed loop-free update algorithm by decreasing the number of helper paths and accelerating the updates from hp_i to hp_{i+1}.

III. The $\Omega(\log n)$-Round Lower Bound

In this section, we present an $\Omega(\log n)$-round lower bound for relaxed loop-free updates where n is the number of nodes in the network, which closes the long-standing open problem whether $O(1)$-round schedules always exist or not [8], [19].

First of all, we construct an update instance $\mathcal{I}_k$ to support all the proofs in this section, where the number of nodes in $\mathcal{I}_k$ is $n = 2^k$ ($k \in \mathbb{N}^*$). The way to construct $\mathcal{I}_k$ is based on mathematical induction.

Mathematical Induction:

Base case: For $k = 2$, $\mathcal{I}_2$ is shown by Fig. 3(a), where $op = 1 \to 2 \to 3 \to 4$ (solid lines) and $np = 1 \to 3 \to 2 \to 4$ (dashed lines) (Note: The base case does not begin with $k = 1$ since the old path and the new path are the same for $\mathcal{I}_1$).

Inductive step: Assume that we have got the update instance $\mathcal{I}_{k-1}$. Next, we will construct $\mathcal{I}_k$ based on $\mathcal{I}_{k-1}$ as follows. See an example in Fig. 3(b)-3(c).

(a) Divide the 2^k nodes into two parts: $V_{in}(1), V_{in}(2), \ldots, V_{in}(2^{k-1})$ and $V_{out}(1), V_{out}(2), \ldots, V_{out}(2^{k-1})$;

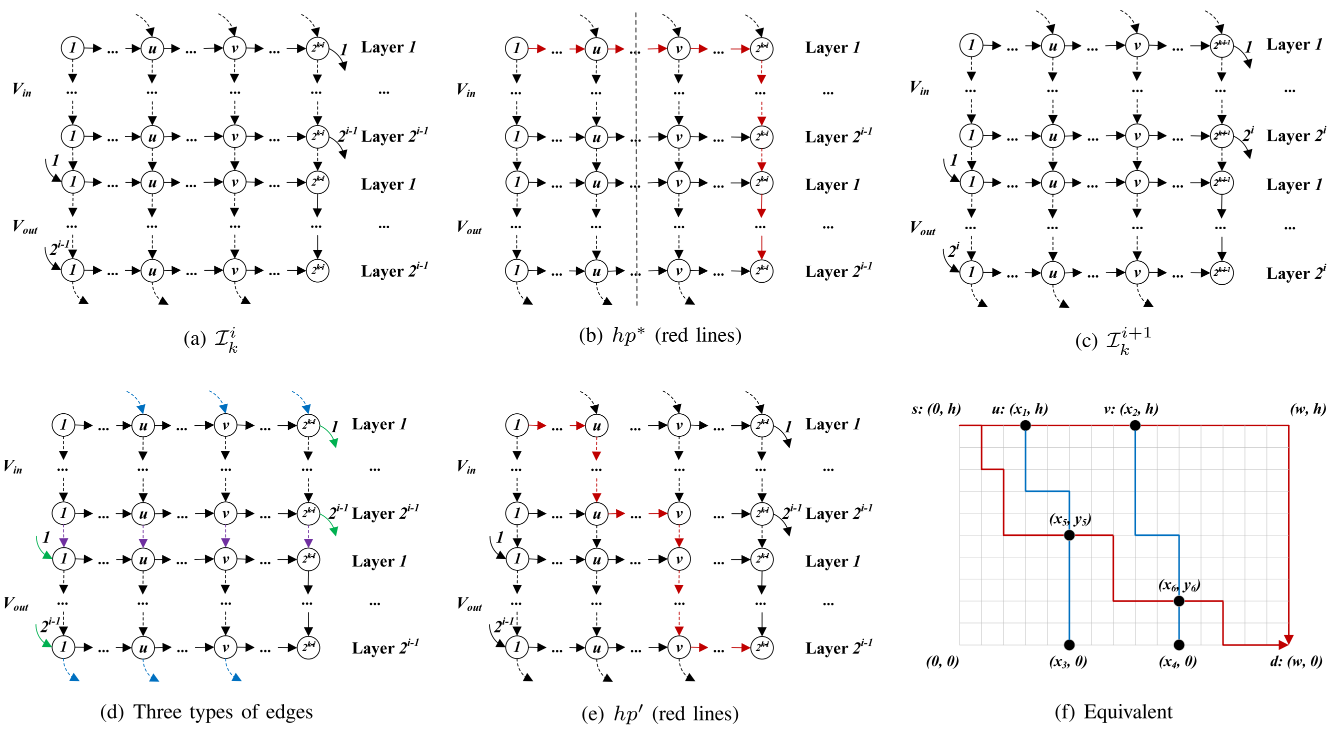
(b) Let the old path be $op = V_{in}(1) \to V_{in}(2) \to \ldots \to V_{in}(2^{k-1}) \to V_{out}(1) \to V_{out}(2) \to \ldots \to V_{out}(2^{k-1})$;

(c) Construct the new path np:

$$\begin{cases} e_f = (V_{in}(i), V_{out}(i)), & 1 \leq i \leq 2^{k-1} \\ e_b = (V_{out}(u), V_{in}(v)), & 1 \leq u,\ v \leq 2^{k-1} \end{cases}$$

where (u, v) is an edge in the new path of $\mathcal{I}_{k-1}$, and e_f (e_b) is actually a forward (backward) edge whose formal definition can be found in Definition 9 (Definition 10).

Through some rigorous mathematical derivation for the above process, the node sequence of op and np in $\mathcal{I}_k$ can



(a) $\mathcal{I}_k^i$ (b) hp^* (red lines) (c) $\mathcal{I}_k^{i+1}$

(d) Three types of edges (e) hp' (red lines) (f) Equivalent

Fig. 4. The $\Omega(\log n)$-round lower bound.

be directly computed by another way based on analytical equations, which is presented as follows. However, it is worth noting that all analysis for $\mathcal{I}_k$ still uses its definition based on mathematical induction throughout this article.

Analytical Equations: The old path in $\mathcal{I}_k$ is numbered as $op = 1 \rightarrow 2 \rightarrow \ldots \rightarrow 2^k$ and the new path is defined by the following equations. Denote by $next_{new}(v)$ the next node of v in the new path for $1 \leq v < 2^k$. For $0 \leq j \leq k-1$, we divide $[1, 2^k - 1]$ into k subintervals, i.e.,

$$I_j = \left[1 + 2^k - 2^{k-j},\ \ 2^k - 2^{k-j-1}\right].$$

If $v \in I_j$, let

$$next_{new}(v) \ = v - 2^k + 3 \times 2^{k-j-1}.$$

Note: The update instance $\mathcal{I}_k$ is motivated by a similar graph constructed in [8], where the latter is used to prove that their algorithm, named Peacock, needs $\Theta(\log n)$ rounds to schedule relaxed loop-free updates for this instance. Their structures are totally different when the number of rounds is more than 32.

In order to produce a lower bound for relaxed loop-freedom, we construct another special update instance $\mathcal{I}_k^i$ based on $\mathcal{I}_k$. As is presented in Fig. 4(a), $\mathcal{I}_k^i$ contains two types of edges, i.e., old edges (solid lines) and new edges (dashed lines). All the nodes in $\mathcal{I}_k^i$ are partitioned into two parts, V_{in} and V_{out}. In each part, there are 2^{i-1} layers and each layer has 2^{k-i} nodes. Denote by $V_{in}^l(u)$ or $V_{out}^l(u)$ the u-th nodes in the l-th layer of V_{in} or V_{out}, respectively. For $1 \leq l \leq 2^{i-1}$, there is one old edge from $V_{in}^l(2^{k-i})$ to $V_{out}^l(1)$, which is marked with the same label in Fig. 4(a). For $1 \leq u, v \leq 2^{k-i}$, there is one new edge from $V_{out}^{2^{i-1}}(u)$ to $V_{in}^1(v)$ if there is one new edge from u to v in $\mathcal{I}_{k-i}$, which is also presented in Fig. 4(a). Let the source node be $V_{in}^1(1)$ and the destination node be $V_{out}^{2^{i-1}}(2^{k-i})$, respectively. Therefore, the old path in $\mathcal{I}_k^i$ is $V_{in}^1(1) \rightarrow V_{in}^1(2) \rightarrow \ldots \rightarrow V_{in}^1(2^{k-i}) \rightarrow V_{out}^1(1) \rightarrow \ldots \rightarrow V_{out}^1(2^{k-i}) \rightarrow \ldots \rightarrow V_{out}^{2^{i-1}}(2^{k-i})$, while the new path is $V_{in}^1(1) \rightarrow \ldots \rightarrow V_{out}^{2^{i-1}}(1) \rightarrow \ldots \rightarrow V_{in}^1(2^{k-i}) \rightarrow \ldots \rightarrow V_{out}^{2^{i-1}}(2^{k-i})$. Notice that the new path contains all nodes in $\mathcal{I}_k^i$ while the old path only uses some of the nodes.

There are three special types of edges in $\mathcal{I}_k^i$ and their definitions are presented as follows. As shown in Fig. 4(d), in-to-out edges are marked with purple, right-to-left edges are marked with green and bottom-to-top edges are marked with blue.

Definition 6 (In-to-Out Edge): For $1 \leq u \leq 2^{k-i}$, the new edge from $V_{in}^{2^{i-1}}(u)$ to $V_{out}^1(u)$ is called an in-to-out edge.

Definition 7 (Right-to-Left Edge): For $1 \leq l \leq 2^{i-1}$, the old edge from $V_{in}^l(2^{k-i})$ to $V_{out}^l(1)$ is called a right-to-left edge.

Definition 8 (Bottom-to-Top Edge): For $1 \leq u, v \leq 2^{k-i}$, the new edge from $V_{out}^{2^{i-1}}(u)$ to $V_{in}^1(v)$ is called a bottom-to-top edge.

Now we present the hardness of this update instance $\mathcal{I}_k^i$.

Theorem 1: For $i, k \in \mathbb{N}^*$ and $i \leq k$, any algorithm needs at least $k-i$ helper paths to schedule relaxed loop-free updates for $\mathcal{I}_k^i$.

Notice that $\mathcal{I}_k^i$ is exactly $\mathcal{I}_k$ when $i = 1$. Therefore, Theorem 1 also indicates that any relaxed loop-free update algorithm needs at least $k-1$ helper paths for $\mathcal{I}_k$. The proof of Theorem 1 is based on one important lemma.

Lemma 2: Let $hp^* = V_{in}^1(1) \rightarrow \ldots \rightarrow V_{in}^1(2^{k-i}) \rightarrow V_{in}^2(2^{k-i}) \rightarrow \ldots \rightarrow V_{out}^{2^{i-1}}(2^{k-i})$ (the red lines in Fig. 4(b)). There is a relaxed loop-free helper path sequence for $\mathcal{I}_k^i$ with

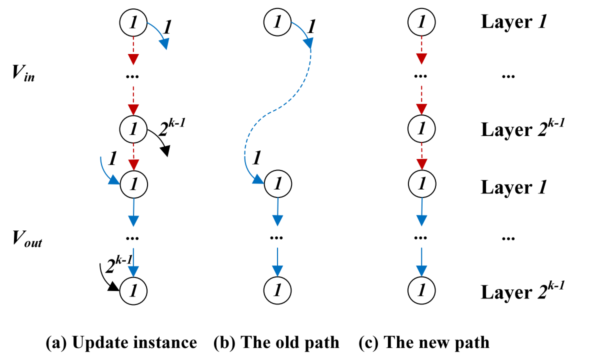


Fig. 5. $\mathcal{I}_k^k$.

the minimum number of helper paths where hp^* is the first helper path.

Proof of Theorem 1: We prove this theorem by using mathematical induction. Fix an arbitrary positive integer k, and let $P(i)$ be the statement that any algorithm needs at least $k-i$ helper paths to schedule relaxed loop-free updates for $\mathcal{I}_k^i$. We induct on i.

Base case: We first show that $P(k)$ holds. For $i = k$, the update instance $\mathcal{I}_k^k$ is shown in Fig. 5(a). The old path op is $V_{in}^1(1) \to V_{out}^1(1) \to \ldots \to V_{out}^{2^{k-1}}(1)$ (Fig. 5(b)) and the new path np is $V_{in}^1(1) \to \ldots \to V_{in}^{2^{k-1}}(1) \to V_{out}^1(1) \to \ldots \to V_{out}^{2^{k-1}}(1)$ (Fig. 5(c)). Since there is no cycle in $op \cup np$, it does not need any helper paths.

Inductive step: We next show that for any $i \in \mathbb{N}^*$ and $i < k$, if $P(i+1)$ holds, then $P(i)$ also holds. Assume that any update algorithm needs at least $k-i-1$ helper paths to schedule relaxed loop-free updates for $\mathcal{I}_k^{i+1}$.

According to Lemma 2, there is a relaxed loop-free helper path sequence for $\mathcal{I}_k^i$ with the minimum number of helper paths where hp^* is the first helper path. Next, we can renumber all the nodes for a better understanding of the structure of the update instance from hp^* to np. See the vertical separation line (gray, dashed) in Fig. 4(b). There are 2^{k-1} nodes at the left of the separation line, which are divided into 2^i layers and there are 2^{k-i-1} nodes in each layer. These 2^{k-1} nodes are renamed as part V_{in}. Similarly, there are also 2^{k-1} nodes at the right of this separation line and they are renamed as part V_{out}. Notice that all the bottom-to-top edges are constructed based on $\mathcal{I}_{k-i}$ and the instance $\mathcal{I}_{k-i}$ is constructed based on $\mathcal{I}_{k-i-1}$. After putting all the nodes in part V_{in} on the top of the nodes in part V_{out}, we get Fig. 4(c). In other words, the update instance from hp^* to np is exactly the instance $\mathcal{I}_k^{i+1}$. Hence, according to the inductive hypothesis, we conclude that any update algorithm needs at least $k-i-1+1 = k-i$ helper paths to schedule relaxed loop-free updates for $\mathcal{I}_k^i$.

Conclusion: For an arbitrary positive integer k, the statement $P(i)$ holds for any $i \in \mathbb{N}^*$ and $i \le k$. ■

For completeness, we will present the detailed proof for Lemma 2. First of all, we begin with some lemmas.

Lemma 3: For paths p_a and p_b in an arbitrary directed graph, if there is a cycle in $p_a \cup p_b$, then $\exists u, v \in p_a \cap p_b$, there is a path from u to v in p_a and there is a path from v to u in p_b.

Proof: We will prove its contrapositive, i.e., if there are two paths in the same direction from u to v (or from v to u) respectively in p_a and p_b for $\forall u, v \in p_a \cap p_b$, then $p_a \cup p_b$ is acyclic. Let $u_1, u_2, \ldots, u_m$ be the shared common nodes between p_a and p_b, and their numbering order be along p_a. Next, we number all the nodes in $p_a \cup p_b$ to construct a topological ordering. Firstly, the first common node is u_1, labeled as x. Let $l_q(u, v)$ be the length of the subpath from u to v in path q (denoted by $p_q(u, v)$). Let l be the sum of $l_a(u_1, u_2)$ and $l_b(u_1, u_2)$. Then u_2 can be numbered as $x + l - 1$. It is similar for $u_3, \ldots, u_m$. For the non-common nodes in subpath $p_a(u_1, u_2)$ ($p_b(u_1, u_2)$), we use $x+1, \ldots, x + l_a(u_1, u_2) - 1$ ($x + l_a(u_1, u_2), \ldots, x + l - 2$) to number them and guarantee the increasing order. It is similar for other non-common nodes. Then the number constitutes a topological ordering, which indicates that $p_a \cup p_b$ is a directed acyclic graph. ■

Lemma 4: There is a relaxed loop-free helper path sequence OPT-HP *for $\mathcal{I}_k^i$ with the minimum number of helper paths, where no helper path contains a right-to-left edge.*

Proof: Let OPT-HP$^* = \{hp_1^*, \ldots, hp_r^*\}$ be a relaxed loop-free helper path sequence with the minimum helper paths and assume that there are some helper paths in OPT-HP* which contain the right-to-left edges. Based on OPT-HP*, we construct another solution HP $= \{hp_1, \ldots, hp_r\}$ where no helper path contains a right-to-left edge. For any $1 \le j \le r$, the way to construct hp_j is as follows:

(a) If hp_j^* does not contain node $V_{in}^l(2^{k-i})$ for any $1 \le l \le 2^{i-1}$, then let $hp_j = hp_j^*$;
(b) Otherwise, let $V_{in}^t(2^{k-i})$ be the first such node that occurs in hp_j^* for all $1 \le t \le 2^{i-1}$. Denote by $hp_j^*(u, v)$ the subpath from u to v in hp_j^*. Then let $hp_j = hp_j^*(s, V_{in}^t(2^{k-i})) \cup (V_{in}^t(2^{k-i}) \to V_{in}^{t+1}(2^{k-i}) \to \ldots \to V_{in}^{2^{i-1}}(2^{k-i}) \to V_{out}^1(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$.

Following the above construction process, if hp_j is obtained from step (a), then hp_j does not contain a right-to-left edge since it does not contain node $V_{in}^l(2^{k-i})$ for any $1 \le l \le 2^{i-1}$; Otherwise, hp_j will contain some nodes $V_{in}^l(2^{k-i})$ where $t \le l \le 2^{i-1}$ but the outgoing edges in hp_j for such nodes are $(V_{in}^l(2^{k-i}), V_{in}^{l+1}(2^{k-i}))$ or $(V_{in}^{2^{i-1}}(2^{k-i}), V_{out}^1(2^{k-i}))$. Therefore, no helper path in HP contains a right-to-left edge. Notice that hp_j may use more new edges on nodes $V_{in}^l(2^{k-i})$ than hp_j^*, but the corresponding old edges on these nodes are right-to-left edges that are never contained in next helper paths in HP. Therefore, HP is a helper path sequence according to Definition 4. Since HP contains the same number of helper paths as OPT-HP*, we only need to prove that HP $= \{hp_1, \ldots, hp_r\}$ satisfies relaxed loop-freedom.

Claim 1: $G_{op} = op \cup hp_1$ does not contain any cycles.

Let $G_{op}^* = op \cup hp_1^*$ and G_{op}^* does not contain any cycles since OPT-HP* is a relaxed loop-free helper path sequence. Therefore, if $hp_1 = hp_1^*$, then G_{op} is also acyclic. Otherwise, $hp_1 = hp_1^*(s, V_{in}^t(2^{k-i})) \cup (V_{in}^t(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$ according to the construction process. Suppose by way of contradiction that there is a cycle in G_{op}. Based on Lemma 3, we have that $\exists u, v \in op \cap hp_1$, there is a path from u to v in op and there is a path from v to u in hp_1. Divide the nodes in $op \cap hp_1$ into two parts, one part in $op \cap hp_1^*(s, V_{in}^t(2^{k-i}))$ and the other part in $op \cap (V_{in}^{t+1}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i})) = (V_{out}^{1}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$. Notice that $u, v \notin (V_{out}^{1}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$ since both op and hp_1 are the subgraphs of $\mathcal{I}_k^i$ but there is no cycle that includes $V_{out}^{l}(2^{k-i})$ for $1 \le l \le 2^{i-1}$ in $\mathcal{I}_k^i$ (see Fig. 4(a)). Therefore, we have $u, v \in op \cap hp_1^*(s, V_{in}^{t}(2^{k-i}))$ but it contradicts that G_{op}^* is acyclic. Hence, the above assumption does not hold.

Claim 2: For $1 \le j < r$, $G_j = hp_j \cup hp_{j+1}$ does not contain any cycles.

If neither hp_j nor hp_{j+1} contains $V_{in}^{l}(2^{k-i})$ for $1 \le l \le 2^{i-1}$, then $hp_j = hp_j^*$ and $hp_{j+1} = hp_{j+1}^*$. Therefore, $G_j = hp_j \cup hp_{j+1}$ is acyclic. If one of hp_j and hp_{j+1} contains $V_{in}^{l}(2^{k-i})$, let it be hp_{j+1} without loss of generality. Therefore, $hp_j = hp_j^*$ and $hp_{j+1} = hp_{j+1}^*(s, V_{in}^{t}(2^{k-i})) \cup (V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$. In this case, the proof is the same as that of Claim 1. If both hp_j and hp_{j+1} contain some nodes $V_{in}^{l}(2^{k-i})$ for $1 \le l \le 2^{i-1}$, let $hp_j = hp_j^*(s, V_{in}^{h}(2^{k-i})) \cup (V_{in}^{h}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$ and $hp_{j+1} = hp_{j+1}^*(s, V_{in}^{t}(2^{k-i})) \cup (V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$. Without loss of generality, let $h \le t$. Suppose by way of contradiction that G_j has a cycle. Therefore, we have that $\exists u, v \in hp_j \cap hp_{j+1}$, there is a path from u to v in hp_j and there is a path from v to u in hp_{j+1}. Similarly, divide the nodes in $hp_j \cap hp_{j+1}$ into two parts, one part in $hp_j^*(s, V_{in}^{h}(2^{k-i})) \cap hp_{j+1}^*(s, V_{in}^{t}(2^{k-i}))$ and the other part in $(V_{in}^{h}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i})) \cap (V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i})) = (V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$. Since neither hp_j nor hp_{j+1} contains right-to-left edges, no cycle in $hp_j \cup hp_{j+1}$ contains the node $V_{in}^{l}(2^{k-i})$ or $V_{out}^{l}(2^{k-i})$. Therefore, we have that $u, v \in hp_j^*(s, V_{in}^{h}(2^{k-i})) \cap hp_{j+1}^*(s, V_{in}^{t}(2^{k-i}))$. However, it contradicts that $G_j^* = hp_j^* \cup hp_{j+1}^*$ is acyclic. Hence, the above assumption does not hold.

Claim 3: $G_{np} = hp_r \cup np$ does not contain any cycles.

Similarly, we only need to consider the case where $hp_r = hp_r^*(s, V_{in}^{t}(2^{k-i})) \cup (V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$. Suppose by way of contradiction that $G_{np} = hp_r \cup np$ has a cycle. Therefore, we have that $\exists u, v \in hp_r \cap np$, there is a path from u to v in hp_r and there is a path from v to u in np. Divide the nodes in $hp_r \cap np$ into two parts, one part in $np \cap hp_r^*(s, V_{in}^{t}(2^{k-i}))$ and the other part in $np \cap (V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$. Notice that $(V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$ is a subpath of np and there is no cycle including $V_{in}^{l}(2^{k-i})$ or $V_{out}^{l}(2^{k-i})$ in G_{np}. Therefore, we conclude that $u, v \notin (V_{in}^{t}(2^{k-i}) \to \ldots \to V_{out}^{2^{i-1}}(2^{k-i}))$. In other words, u, v belongs to $np \cap hp_r^*(s, V_{in}^{t}(2^{k-i}))$, which contradicts that $G_{np}^* = hp_r^* \cup np$ is acyclic. ■

Lemma 5: The first helper path hp_1 *in* OPT-HP *defined in Lemma 4 does not contain any bottom-to-top edges.*

Proof: In Fig. 4(a), the nodes can be divided into two parts, V_{in} and V_{out}. The source node s belongs to V_{in} part while the destination node d locates in V_{out} part. Notice that there are two types of edges that connect V_{in} to V_{out}, the right-to-left edges and the in-to-out edges. Suppose that hp_1 contains a bottom-to-top edge, $(V_{out}^{2^{i-1}}(u), V_{in}^{1}(v))$. Since hp_1 does not contain any right-to-left edges according to Lemma 4, $hp_1(s, V_{out}^{2^{i-1}}(u))$ must pass through an in-to-out edge. Denote it by $(V_{in}^{2^{i-1}}(j), V_{out}^{1}(j))$. Since there is one path from $V_{in}^{1}(v)$ to $V_{out}^{1}(j)$ in op and there is also an path from $V_{out}^{1}(j)$ to $V_{in}^{1}(v)$ in hp_1, $G_{op} = op \cup hp_1$ contains a cycle. It contradicts that OPT-HP is a relaxed loop-free helper path sequence. Therefore, hp_1 does not contain any bottom-to-top edges. ■

Now we can prove Lemma 2 based on Lemma 3-5.

Proof of Lemma 2: According to Lemma 4-5, the first helper path hp_1 in OPT-HP does not contain any right-to-left edges and bottom-to-top edges. Therefore, hp_1 consists of the downward edges and the rightward edges, like hp' (red lines in Fig. 4(e)). Suppose that $hp_1 = hp'$ in the optimal solution OPT-HP $= \{hp_1, hp_2, \ldots, hp_r\}$. We claim that $\{hp^*, hp_2, \ldots, hp_r\}$ is another optimal solution.

First we prove that $\{hp^*, hp_2, \ldots, hp_r\}$ is a helper path sequence. Obviously hp^* is a helper path. Notice that hp^* only contains new edges on node $V_{in}^{l}(2^{k-i})$ where $1 \le l \le 2^{i-1}$. The corresponding old edges on these nodes are right-to-left edges, but they are not contained in hp_j where $2 \le j \le r$ according to Lemma 4. Therefore, we have that $\{hp^*, hp_2, \ldots, hp_r\}$ is a helper path sequence.

Next, we only need to prove that $hp^* \cup hp_2$ is acyclic if $hp' \cup hp_2$ is acyclic. For a better understanding of the proof, the graph of $\mathcal{I}_k^i$ in Fig. 4(a) is abstracted to a mesh network in Fig. 4(f). The source node s is modeled as $(0, h)$ and the destination node d is modeled as $(w, 0)$. All right-to-left edges are deleted because no helper path in OPT-HP contains them. All bottom-to-top edges are omitted in this mesh network. The top-right red lines represent hp^* and the bottom-left red lines represent hp' in Fig. 4(f). Suppose that $hp^* \cup hp_2$ contains a cycle. Based on Lemma 3, we have that $\exists u, v \in hp^* \cap hp_2$, there is a path from u to v in hp^* and there is a path from v to u in hp_2. As is shown in Fig. 4(f), hp^* consists of two parts: the horizontal segment and the vertical segment. Since neither hp^* nor hp_2 contains right-to-left edges, no cycle in $hp^* \cup hp_2$ contains the node in the vertical segment of hp^* (i.e., $V_{in}^{l}(2^{k-i})$ or $V_{out}^{l}(2^{k-i})$ for some $1 \le l \le 2^{i-1}$). Therefore, we have that u, v must be located in the horizontal segment of hp^*. Let $u = (x_1, h)$ and $v = (x_2, h)$ as shown in Fig. 4(f) where $x_1 < x_2$. Since all edges except the bottom-to-top edges are downward or rightward, the subpath from v to u in hp_2 (the right blue lines in Fig. 4(f)) must contain a bottom-to-top edge, denoted by $((x_4, 0), (x_1, h))$. As is shown by Fig. 4(f), the subpath from v to $(x_4, 0)$ in hp_2 must intersect with hp'. Denote by (x_6, y_6) the rightmost one of the intersection points, which satisfies $x_6 < w$. In addition, the subpath from u to d in hp_2 must also contain a bottom-to-top edge since otherwise it will intersect with the subpath from v to $(x_4, 0)$ in hp_2. Denote by $(x_3, 0)$ the left end point of this bottom-to-top edge, which satisfies $x_3 < x_4$. Similarly, the subpath from u to $(x_3, 0)$ (the left blue lines in Fig. 4(f)) will intersect with hp'. Denote by (x_5, y_5) the rightmost one of the intersection nodes, which satisfies $x_1 \le x_5 \le x_3 < x_4$. Therefore, we find a subpath from (x_6, y_6) to (x_5, y_5) in hp_2. After hp' passes (x_5, y_5), it must intersect with the subpath from v to $(x_4, 0)$ in hp_2 (the right blue lines) in order to reach the destination d. Therefore, there is a subpath from (x_5, y_5) to (x_6, y_6) in hp'. Hence, $hp' \cup hp_2$ contains a cycle, which induces a contradiction. ■

Algorithm 1 Shortest Path First Algorithm

```
Input: the old path $op$, the new path $np$.
Output: $hp_1, hp_2, \dots, hp_r$.
1: $i \leftarrow 0$.
2: $hp_i \leftarrow op$.
3: while $G_i = hp_i \cup np$ has a cycle do
4:     Delete all the non-common nodes in $hp_i$ and $np$.
5:     Merge $u$ with $v$ in $hp_i$ and $np$ for the common edges $(u,v)$.
6:     for $e \in G_i$ do
7:         if $e$ is a backward edge of $np$ then
8:             $w(e) \leftarrow \infty$.
9:         else if $e$ is a forward edge of $np$ then
10:            $w(e) \leftarrow 0$.
11:        else
12:            $w(e) \leftarrow 1$.
13:    $hp_{i+1} \leftarrow$ one of the shortest paths from $s$ to $d$.
14:    $i \leftarrow i + 1$.
15: $r \leftarrow i$.
16: $hp_1, hp_2, \dots, hp_r \leftarrow$ UNFOLD$(op, hp_1, \dots, hp_r, np)$.
17: return $hp_1, hp_2, \dots, hp_r$.
```

Function UNFOLD$(op, hp_1, \dots, hp_r, np)$

```
1: $i \leftarrow 0$.
2: $hp'_i \leftarrow op$.
3: for $i \leftarrow 0$ to $r - 1$ do
4:     $hp'_{i+1} \leftarrow hp_{i+1}$.
5:     for $v \in hp_{i+1}$ do
6:         unfold $v$ to the corresponding edge before merged.
7:     for $(u,v) \in hp_{i+1}$ do
8:         if $(u,v) \in hp_i$ then
9:             $p_{u,v} \leftarrow$ the path from $u$ to $v$ in $hp'_i$.
10:        else
11:            $p_{u,v} \leftarrow$ the path from $u$ to $v$ in $np$.
12:        replace $(u,v)$ in $hp'_{i+1}$ with $p_{u,v}$.
13: return $hp'_1, hp'_2, \dots, hp'_r$.
```

Therefore, we have completed the proof of Theorem 1. Since $\mathcal{I}_k^1 = \mathcal{I}_k$, we have that any algorithm requires at least $k-1$ helper paths for $\mathcal{I}_k$ where the number of nodes is $n = 2^k$. According to Lemma 1, we can further obtain the lower bound for the relaxed loop-free update problem.

Theorem 2: Any algorithm requires $\Omega(\log n)$ rounds to guarantee relaxed loop-freedom in the worst case.

IV. A Fast Relaxed Loop-Free Update Algorithm

In this section, we propose Savitar, a fast relaxed loop-free update algorithm that exactly touches the tight lower bound.

A. Decreasing the Number of Helper Paths

We begin with two related definitions to support our algorithm, which are also used in [8], [19].

Definition 9 (Forward Edge): For paths p_a and p_b with exactly the same node set, an edge (u, v) in p_b is called a forward edge of p_b if there is a path from u to v in p_a.

Definition 10 (Backward Edge): For paths p_a and p_b with exactly the same node set, an edge (u, v) in p_b is called a backward edge of p_b if there is a path from v to u in p_a.

In Fig. 1(a), let p_a be the old path op and p_b be the new path np. Then (s, v_4) and (v_2, d) are the forward edges of np while (v_3, v_2) and (v_4, v_3) are the backward edges of np.

According to the above definitions, we know that $G_{op} = op \cup hp_1$ is acyclic if hp_1 consists of the edges from op and the forward edges of np. Based on this observation, we propose an algorithm to compute a relaxed loop-free helper path sequence. Algorithm 1 first initializes $hp_0 = op$ (lines 1-2) and tends to obtain hp_{i+1} based on hp_i and np (lines 4-15). If $G_i = hp_i \cup np$ does not contain any cycles, we have found a feasible helper path sequence and this algorithm will terminate. Otherwise, Algorithm 1 simplifies hp_i and np by deleting the non-common nodes from their node sequences and merging the common edges between these two paths (line 4-5). After the simplification, hp_i and np have the same node set and the weight of each edge $w(e)$ is assigned according to the type of the edges (line 6-14). Finally, let hp_{i+1} be one of the shortest paths from s to d in the weighted graph G_i. Because all helper paths have been simplified in lines 4-5, Algorithm 1 finally uses a function named UNFOLD to obtain the original paths. A simple example for Algorithm 1 is presented in Fig. 6.

In each iteration (lines 3-17), hp_{i+1} only consists of the edges in hp_i and the forward edges in np because the weight of the backward edges of np is set as ∞. Therefore, $hp_i \cup hp_{i+1}$ does not contain any cycles for $0 \le i \le r$ where $hp_0 = op$ and $hp_{r+1} = np$. Since the simplification operations (lines 4-5) do not influence the cycle, unfolding all helper paths to the ones before simplification (line 19) will also not introduce any cycles. Therefore, Algorithm 1 outputs a relaxed loop-free helper path sequence. In addition, the number of helper paths is also bounded by $\lfloor \log_2 n \rfloor - 1$, where n is the number of nodes in the network. We prove it by Lemma 6 and Theorem 3.

Lemma 6: Let $U = [1, n]$ be an interval and $\mathcal{F}$ be a family of subintervals of U. Define the length of the interval $I = [a, b]$ as $|I| = b - a$. If $\bigcup_{I \in \mathcal{F}} I = U$, then there is a subfamily $\mathcal{F}^* \subseteq \mathcal{F}$ which satisfies

(1) $|\bigcup_{I \in \mathcal{F}^*} I| \ge \lfloor n/2 \rfloor$;

(2) $|I_1 \cap I_2| = 0$ for $\forall I_1, I_2 \in \mathcal{F}^*$.

Proof: We prove this lemma by constructing two subfamilies $\mathcal{F}_1, \mathcal{F}_2 \subseteq \mathcal{F}$ which satisfy the following properties

(i) $|I_1 \cap I_2| = 0$ for $\forall I_1, I_2 \in \mathcal{F}_1$;

(ii) $|I_1 \cap I_2| = 0$ for $\forall I_1, I_2 \in \mathcal{F}_2$;

(iii) $\bigcup_{I \in \mathcal{F}_1 \cup \mathcal{F}_2} I = U$.

The construction process is as follows:

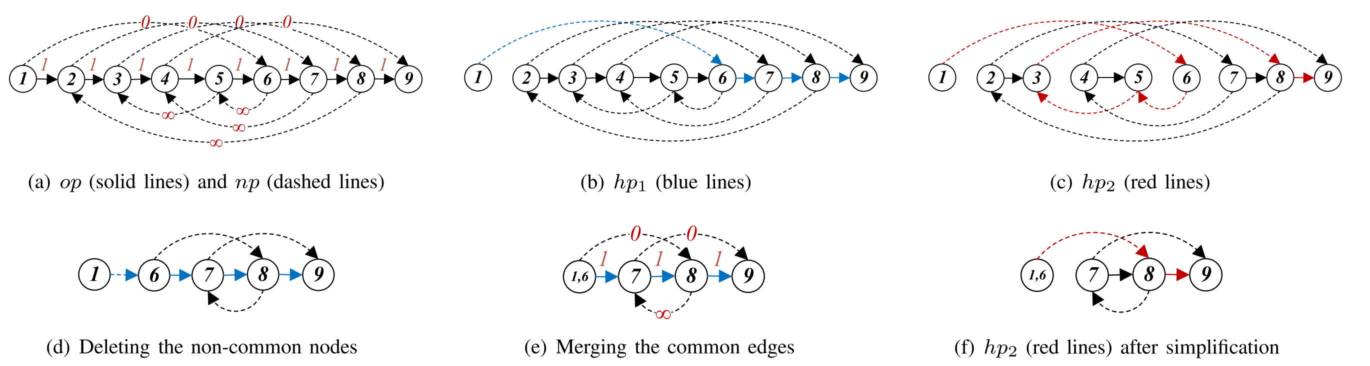
step 1: $\mathcal{F}_1 \leftarrow \emptyset$, $\mathcal{F}_2 \leftarrow \emptyset$, $l \leftarrow 1$, $r \leftarrow 1$;

step 2: if $r < n$, then select the interval $I = [a, b]$ satisfying $l \le a \le r$ and $b = \max_{[a,b] \in \mathcal{F}, l \le a \le r} b$ into $\mathcal{F}_1$, and next $l \leftarrow r$, $r \leftarrow b$;

step 3: if $r < n$, then select the interval $I = [a, b]$ satisfying $l \le a \le r$ and $b = \max_{[a,b] \in \mathcal{F}, l \le a \le r} b$ into $\mathcal{F}_2$, and next $l \leftarrow r$, $r \leftarrow b$;

step 4: repeat step 2-3 until $r = n$.

Before proving the correctness of the above construction process, we introduce some intermediate variables to help



(a) op (solid lines) and np (dashed lines) (b) hp_1 (blue lines) (c) hp_2 (red lines)

(d) Deleting the non-common nodes (e) Merging the common edges (f) hp_2 (red lines) after simplification

Fig. 6. An example of Algorithm 1. The old path $op = 1 \rightarrow 2 \rightarrow \ldots \rightarrow 9$ and the new path $np = 1 \rightarrow 6 \rightarrow 5 \rightarrow 3 \rightarrow 8 \rightarrow 2 \rightarrow 7 \rightarrow 4 \rightarrow 9$. In the first iteration, assign the weight of each edge in Fig. 6(a). Let hp_1 be one of the shortest paths, e.g., $hp_1 = 1 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$ in Fig. 6(b). In the second iteration, delete the non-common nodes $2, 3, 4, 5$ from the node sequence of hp_1 and np and get Fig. 6(d). Then merge the common edges $(1, 6)$ and get Fig. 6(e). Assign the weight of each edge and let hp_2 be one of the shortest paths, e.g., $\{1, 6\} \rightarrow 8 \rightarrow 9$ in Fig. 6(f). Unfold hp_2 by unfolding node $\{1, 6\}$ to edge $(1, 6)$ and replacing edge $(\{1, 6\}, 8)$ with path $\{1, 6\} \rightarrow 5 \rightarrow 3 \rightarrow 8$ since it belongs to np, and get Fig. 6(c).

the analysis. Denote by $I_1^i = [a_1^i, b_1^i]$ ($I_2^i = [a_2^i, b_2^i]$) the interval added into $\mathcal{F}_1$ ($\mathcal{F}_2$) in step 2 (step 3) of the i-th iteration, and denote by $\mathcal{F}_1^i$ ($\mathcal{F}_2^i$) the corresponding subfamily $\mathcal{F}_1$ ($\mathcal{F}_2$) obtained after step 2 (step 3). Denote l, r after step 2 of the i-th iteration by l_{2k-1}, r_{2k-1}, and denote l, r after step 3 of the i-th iteration by l_{2k}, r_{2k}. When the construction process terminates, it may happen that one interval is added into $\mathcal{F}_1$ but no interval is added into $\mathcal{F}_2$ in the last iteration because the conditional statement in step 3 is not satisfied (due to $r = n$) at this time. In order to simplify the analysis, hypothetically a virtual interval $I^* = [n, n]$ is added into $\mathcal{F}_2$ in this case, such that there are the same number of intervals in $\mathcal{F}_1$ and $\mathcal{F}_2$.

First we claim that the following propositions always hold after the k-th iteration

(a) $l_{2k-1} = b_2^{k-1}, r_{2k-1} = b_1^k, l_{2k} = b_1^k, r_{2k} = b_2^k$;
(b) $a_2^{k-1} \le a_1^k \le b_2^{k-1} \le a_2^k \le b_1^k \le b_2^k$;
(c) $r_{2k-2} < r_{2k-1} < r_{2k}$;
(d) $\bigcup_{I \in \mathcal{F}_1^k \cup \mathcal{F}_2^k} I = [1, r_{2k}]$.

Define the initial value where $a_1^0, b_1^0, a_2^0, b_2^0, l_0, r_0 = 1$. We prove these propositions by using mathematical induction.

Base case: It can be easily verified that these propositions hold when $k = 1$ by executing the first iteration.

Inductive step: Assume that these propositions always hold in the first $k-1$ iterations. Now consider the k-th iteration. Before step 2, we have that $l = l_{2k-2}$ and $r = r_{2k-2}$. In step 2, one interval $I_1^k = [a_1^k, b_1^k]$ is added into $\mathcal{F}_1$. After executing $l \leftarrow r$ and $r \leftarrow b$ in step 2, we have that $l_{2k-1} = b_2^{k-1}$ and $r_{2k-1} = b_1^k$ hold. Similarly, after adding one interval $I_2^k = [a_2^k, b_2^k]$ into $\mathcal{F}_2$ and executing $l \leftarrow r, r \leftarrow b$ in step 3, we have that $l_{2k} = b_1^k$ and $r_{2k} = b_2^k$ hold. Therefore, Proposition (a) holds after the k-th iteration.

Notice that the interval $I_1^k = [a_1^k, b_1^k]$ satisfies $l_{2k-2} \le a_1^k \le r_{2k-2}$ in step 2. Next, we will prove that $r_{2k-2} < b_1^k$ holds. According to the hypothesis, we have that $\bigcup_{I \in \mathcal{F}_1^{k-1} \cup \mathcal{F}_2^{k-1}} I = [1, r_{2k-2}]$ holds. Due to $\bigcup_{I \in \mathcal{F}} I = [1, n]$ and $r_{2k-2} < n$, there is at least one interval $I^* = [a^*, b^*]$ in $\mathcal{F}$ that satisfies $a^* \le r_{2k-2} < b^*$. We claim that $a^* \ge l_{2k-2}$ holds. Suppose by way of contradiction that $a^* < l_{2k-2}$ holds. Then, we have that $\exists 1 \le i < k$, either $l_{2i-2} \le a^* \le r_{2i-2}$ or $l_{2i-1} \le a^* \le r_{2i-1}$ holds, where $l_{2i-2} = b_1^{i-1}, r_{2i-2} = l_{2i-1} = b_2^{i-1}$ and $r_{2i-1} = b_1^i$. For the former, $I^* = [a^*, b^*]$ should be added into $\mathcal{F}_1$ instead of $I_1^i = [a_1^i, b_1^i]$ in step 2 of the i-th iteration because of $l_{2i-2} \le a^* \le r_{2i-2}$ and $b^* > r_{2k-2} = b_2^{k-1} \ge b_1^i$; For the latter, $I^* = [a^*, b^*]$ should be added into $\mathcal{F}_2$ instead of $I_2^i = [a_2^i, b_2^i]$ in step 3 of the i-th iteration because of $l_{2i-1} \le a^* \le r_{2i-1}$ and $b^* > r_{2k-2} = b_2^{k-1} \ge b_2^i$. Therefore, we have that $l_{2k-2} \le a^* \le r_{2k-2} < b^*$ holds and the interval $I_1^k = [a_1^k, b_1^k]$ added into $\mathcal{F}_1$ in step 2 satisfies $l_{2k-2} \le a_1^k \le r_{2k-2} < b_1^k$. Similarly, the interval $I_2^k = [a_2^k, b_2^k]$ added into $\mathcal{F}_2$ in step 3 satisfies $l_{2k-1} \le a_2^k \le r_{2k-1} < b_2^k$. Due to $l_{2k-2} = b_1^{k-1}, r_{2k-2} = l_{2k-1} = b_2^{k-1}, r_{2k-1} = b_1^k$ and $a_2^{k-1} \le b_1^{k-1}$ according to the hypothesis, we have that Proposition (b) holds after the k-th iteration.

In the proof of Proposition (b), we have that $l_{2k-2} \le a_1^k \le r_{2k-2} < b_1^k$ and $l_{2k-1} \le a_2^k \le r_{2k-1} < b_2^k$ hold. Due to $r_{2k-1} = b_1^k$ and $r_{2k} = b_2^k$, we have that Proposition (c) also holds after the k-th iteration.

According to the hypothesis, we have that

$$\begin{aligned}\bigcup_{I \in \mathcal{F}_1^k \cup \mathcal{F}_2^k} I &= \Big(\bigcup_{I \in \mathcal{F}_1^{k-1} \cup \mathcal{F}_2^{k-1}} I\Big) \cup I_1^k \cup I_2^k \\ &= [1, r_{2k-2}] \cup [a_1^k, b_1^k] \cup [a_2^k, b_2^k] \\ &= [1, r_{2k}].\end{aligned}$$

The last equality holds because of $a_1^k \le r_{2k-2} = b_2^{k-1} \le a_2^k \le b_1^k \le b_2^k = r_{2k}$. Therefore, we have that Proposition (d) holds after the k-th iteration.

Conclusion: All of Propositions (a)-(d) hold after the k-th iteration.

Based on Proposition (b), we have that the adjacent intervals $I_2^{k-1} = [a_2^{k-1}, b_2^{k-1}]$ and $I_2^k = [a_2^k, b_2^k]$ in $\mathcal{F}_2$ satisfy $a_2^{k-1} \le b_2^{k-1} \le a_2^k \le b_2^k$, which indicates that Property (ii) holds. Similarly, the adjacent intervals $I_1^{k-1} = [a_1^{k-1}, b_1^{k-1}]$ and $I_1^k = [a_1^k, b_1^k]$ in $\mathcal{F}_1$ also satisfy $a_1^{k-1} \le b_1^{k-1} \le a_1^k \le b_1^k$ based on Proposition (b). Therefore, Property (i) is also satisfied. According to Proposition (c), the variable r gradually increases during the iterations. Therefore, the construction process can terminate successfully until r reaches the value of n. At this time, we have that $r_{2k} = n$ holds and Property (iii) is satisfied according to Proposition (d).

Since $\mathcal{F}_1$ and $\mathcal{F}_2$ satisfy Properties (i) (ii) (iii), we have

$$\left|\bigcup_{I\in\mathcal{F}_1} I\right| \geq \left\lceil\frac{n-1}{2}\right\rceil \text{ or } \left|\bigcup_{I\in\mathcal{F}_2} I\right| \geq \left\lceil\frac{n-1}{2}\right\rceil.$$

Due to $\lceil\frac{n-1}{2}\rceil = \lfloor\frac{n}{2}\rfloor$, we complete the proof by letting

$$\mathcal{F}^* = \arg\max_{\mathcal{F}_i}\left|\bigcup_{I\in\mathcal{F}_i} I\right|.$$

■

Theorem 3: For any update instance, the number of helper paths that are computed by Algorithm 1 is at most $\lfloor\log_2\ n\rfloor - 1$*, where* n *is the number of nodes in the network.*

Proof: Consider each iteration in Algorithm 1. After the simplification in lines 4-5 in Algorithm 1, the i-th helper path hp_i and the new path np have the same node set. Use $1, 2, \ldots, n$ to renumber the nodes along hp_i. We have $hp_i = 1 \rightarrow 2 \rightarrow \ldots \rightarrow n$. In addition, we can find that all forward edges (u_f, v_f) of np satisfy $v_f > u_f$ and all backward edges (u_b, v_b) of np satisfy $v_b < u_b$. Replace each forward edge (u, v) of np with an interval $I = [u, v]$. Let U be the universal set $[1, n]$ and $\mathcal{F}$ be a family of the intervals generated by the forward edges of np. We claim that $\bigcup_{I\in\mathcal{F}} I = U$ is true. In order to prove it, suppose by way of contradiction that it does not hold. Then there is an interval $I^* = [a, b] \neq \emptyset$ that satisfies $I^* \subseteq U$ and $I^* \cap (\bigcup_{I\in\mathcal{F}} I) = \emptyset$. However, it means that there is no edge from $\{1, 2, \ldots, \lfloor a\rfloor\}$ to $\{\lceil b\rceil, \ldots, n-1, n\}$ in np, which induces a contradiction.

According to Lemma 6, we can obtain a set of the forward edges $\mathcal{S}$ (i.e., the subfamily $\mathcal{F}^*$) where each forward edge does not overlap with each other. Let p be a path from node 1 to node n that contains all the forward edges of np in $\mathcal{S}$, but the remaining edges of p are from hp_i. Notice that each forward edge (u, v) skips $v-u-1$ nodes. Denote by $|p|$ the number of nodes in p. Therefore, we have

$$|p| = n - \sum_{(u,v)\in\mathcal{S}} (v-u-1) = n - \left|\bigcup_{I\in\mathcal{F}^*} I\right| + |\mathcal{S}|.$$

Denote by $w(p)$ the sum of the weight of each edge in p where the weight function is defined in lines 6-12 in Algorithm 1. Since the weight of all the forward edges of np is set as 0, we can get

$$w(p) = |p| - 1 - |\mathcal{S}| = n - 1 - \left|\bigcup_{I\in\mathcal{F}^*} I\right|.$$

Since hp_{i+1} is one of the shortest path, we have $w(hp_{i+1}) \leq w(p)$. Consider the simplification in lines 4-5 in next iteration and denote hp_{i+1} after the simplification by hp^*_{i+1}. Notice that $w(hp_{i+1})$ is greater than or equal to the number of edges in hp^*_{i+1} because all the forward edges of np in $\mathcal{S}$ will be merged into one node by the simplification. Therefore, we have

$$|hp^*_{i+1}| \leq w(hp_{i+1}) + 1 \leq n - \left|\bigcup_{I\in\mathcal{F}^*} I\right|.$$

Now consider $\mathcal{F}_1$ and $\mathcal{F}_2$ that are constructed in the proof of Lemma 5. We claim that there must always be an interval $I_1 \in \mathcal{F}_1$ and an interval $I_2 \in \mathcal{F}_2$, which satisfy $I_1 \cap I_2 \geq 1$. Suppose by way of contradiction that it does not hold. For any $I = [u, v] \in \mathcal{F}$, both u and v are integers. Then we have $I_1 \cap I_2 = 0$ for any $I_1, I_2 \in \mathcal{F}_1 \cup \mathcal{F}_2$. Due to $(\bigcup_{I\in\mathcal{F}_1\cup\mathcal{F}_2} I) = U$, all the forward edges in $\mathcal{F}_1 \cup \mathcal{F}_2$ have made up a path from node 1 to node n, which is exactly the new path np. Therefore, there is no cycle in $hp_i \cup np$ and the iteration should terminate in line 3 in Algorithm 1. Hence, the above assumption does not hold and we can have

$$\left|\bigcup_{I\in\mathcal{F}_1} I\right| + \left|\bigcup_{I\in\mathcal{F}_2} I\right| \geq n - 1 + 1 = n.$$

Therefore, we get

$$\left|\bigcup_{I\in\mathcal{F}^*} I\right| = \max\left(\left|\bigcup_{I\in\mathcal{F}_1} I\right|, \left|\bigcup_{I\in\mathcal{F}_2} I\right|\right) \geq \left\lceil\frac{n}{2}\right\rceil.$$

Hence, we have $|hp^*_{i+1}| \leq n - \lceil\frac{n}{2}\rceil = \lfloor\frac{n}{2}\rfloor$. In other words, the number of the common nodes between hp_i and np will be halved in each iteration. Notice that no helper path is required when $n = 2, 3$. Therefore, Algorithm 1 will terminate after finding at most $\lfloor\log_2\ n\rfloor - 1$ helper paths. ■

B. Accelerating the Updates From hp_i to hp_{i+1}

Given a relaxed loop-free sequence with r helper paths, we can construct a feasible relaxed loop-free update sequence with $2r + 1$ rounds according to Lemma 1. However, $2r + 1$ rounds are not always essential actually and we can accelerate the updates from hp_i to hp_{i+1}.

Let UPD_u record whether u has been updated in Algorithm 2. In order to update hp_i to hp_{i+1}, Algorithm 2 first checks whether one round is feasible (line 7). If so, Algorithm 2 updates all the nodes that need to be updated in order to migrate hp_i to hp_{i+1} (lines 8-10). Meanwhile, Algorithm 2 also greedily updates other nodes in this round that will not introduce a cycle that s can reach (lines 11-15). Otherwise, Algorithm 2 uses two rounds. In the first round, all nodes which are not in hp_i will be updated (lines 17-19). In the second round, all the common nodes in $hp_i \cap hp_{i+1}$ that need to be updated in order to migrate hp_i to hp_{i+1} will be updated, and a part of the non-common nodes that will not introduce any loops that s can reach are also updated (lines 24-28). Notice that for $0 \leq i \leq r-1$, $hp_{i+1} \in hp_i \cup np$ in the helper path sequence computed by Algorithm 1. Therefore, if an old edge $(u, v) \notin hp_i$, then we have $(u, v) \notin hp_j$ for $j > i$. In other words, greedily updating some nodes whose old edges do not belong to hp_i or hp_{i+1} is safe and feasible in Algorithm 2 (lines 11-15, lines 17-19 and lines 24-28).

Take the instance in Fig. 6 as an example again. Since there is no cycle in $\mathcal{G}_0 \cup hp_1$, we can update op to hp_1 in one round. Therefore, we get $U_1 = \{1\}$ after executing lines 8-10 in Algorithm 2. According to line 11-15, node $2, 3, 4$ can also be updated in this round, i.e., $U_1 = \{1, 2, 3, 4\}$. After updating the nodes in U_1, we can find that $\mathcal{G}_1 \cup hp_2$ has a cycle $5 \rightarrow 6 \rightarrow 5$ that s can reach and thus we use 2 rounds to update hp_1 to hp_2. In the first round, update all the nodes that are not in hp_1 and have not been updated (lines 17-19), i.e., node 5 in this example. In the second round, node $6, 7$ will be updated according to lines 24-28. Therefore, we have $U_2 = \{5\}$ and $U_3 = \{6, 7\}$. Finally, node 8 is updated in U_4 in order to update hp_2 to np. Therefore, Algorithm 2 uses 4 rounds to schedule relaxed loop-free updates for this instance, while the construction mechanism in Lemma 1 needs 5 rounds.

Algorithm 2 Helper Path Migration Algorithm

```
Input: $op, hp_1, hp_2, \ldots, hp_r, np$.
Output: $U_1, U_2, \ldots, U_m$.
1: $j \leftarrow 1$, $hp_0 \leftarrow op$, $hp_{r+1} \leftarrow np$, $\mathcal{G}_0 \leftarrow op$.
2: for $u \in U \backslash d$ do
3:   $\mathrm{UPD}_u \leftarrow F$.
4: for $i \leftarrow 0$ to $r$ do
5:   $\mathcal{G}_i^{i+1} \leftarrow \mathcal{G}_i \cup hp_{i+1}$.
6:   $U_j \leftarrow \emptyset$.
7:   if $\mathcal{G}_i^{i+1}$ has no cycle reached by $s$ then
8:     for $(u, v) \in hp_{i+1}$ do
9:       if $\mathrm{UPD}_u = F$ and $(u, v) \in np$ then
10:        $U_j \leftarrow U_j \cup \{u\}$, $\mathrm{UPD}_u \leftarrow T$.
11:    for $u \in U \backslash hp_{i+1}$ do
12:      $v \leftarrow next_{new}(u)$.
13:      if $\mathrm{UPD}_u = F$ and $\mathcal{G}_i^{i+1} \cup \{(u, v)\}$ has no cycle reached by $s$ then
14:        $U_j \leftarrow U_j \cup \{u\}$, $\mathrm{UPD}_u \leftarrow T$.
15:        $\mathcal{G}_i^{i+1} \leftarrow \mathcal{G}_i^{i+1} \cup \{(u, v)\}$.
16:  else
17:    for $u \in U \backslash hp_i$ do
18:      if $\mathrm{UPD}_u = F$ then
19:        $U_j \leftarrow U_j \cup \{u\}$, $\mathrm{UPD}_u \leftarrow T$.
20:    $\mathcal{G}_i \leftarrow \mathcal{G}_i$ after updating the nodes in $U_j$.
21:    $\mathcal{G}_i^{i+1} \leftarrow \mathcal{G}_i \cup hp_{i+1}$.
22:    $j \leftarrow j + 1$.
23:    $U_j \leftarrow \emptyset$.
24:    for $(u, v) \in hp_i$ do
25:      $t \leftarrow next_{new}(u)$.
26:      if $\mathrm{UPD}_u = F$ and $(u, v) \notin hp_{i+1}$ and $\mathcal{G}_i^{i+1} \cup \{(u, t)\}$ has no cycle reached by $s$ then
27:        $U_j \leftarrow U_j \cup \{u\}$, $\mathrm{UPD}_u \leftarrow T$.
28:        $\mathcal{G}_i^{i+1} \leftarrow \mathcal{G}_i^{i+1} \cup \{(u, t)\}$.
29:  $\mathcal{G}_i \leftarrow \mathcal{G}_i$ after updating the nodes in $U_j$.
30:  $j \leftarrow j + 1$.
31: $m \leftarrow j - 1$.
32: return $U_1, U_2, \ldots, U_m$.
```

Finally, we claim that Savitar touches the tight lower bound according to Lemma 1 and Theorem 3.

Theorem 4: For any update instance, Savitar can use at most $2\lfloor \log_2 \ n \rfloor - 1$ *rounds to schedule relaxed loop-free updates, where* n *is the number of nodes in the network.*

V. Experimental Evaluation

In this section, we conduct both a prototype implementation and extensive simulations to evaluate our algorithm.

Benchmark Schemes: we compare the following loop-free update algorithms in the experiment.

1) **Greedy:** the algorithm in [10] which updates a maximal set of nodes without introducing any loops in each round.
2) **Peacock:** the algorithm in [19] which updates the disjoint forward edges starting with the maximum forward distance in each odd round.
3) **Savitar:** the fast update algorithm in this article.
4) **OPT:** the mixed integer programming to minimize the number of rounds which is solved by Gurobi 8.1.0 [2].

The small-scale update scenarios used in this experiment are generated from the real network topologies in Topology Zoo [5], which contains over two hundred and fifty different networks. We select two different nodes as the source and destination, and compute two different paths from source to destination as the old routing path and the new routing path by using a common search algorithm (e.g., DFS). However, the size of most networks in Topology Zoo is small, where the number of nodes is less than 100. Correspondingly, we generate random Erdős-Rényi graphs [22] with 5000 nodes by using ZER algorithm [6] and obtain the large-scale update instances from the random graphs based on the above approach.

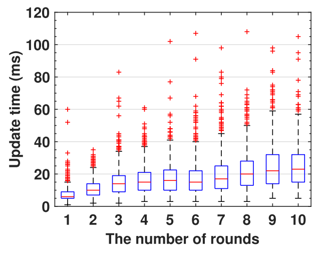
Due to the limitation of the experimental platform, we cannot conduct an experiment for thousands of update instances with large size to compare the update time of the above algorithms. However, all algorithms based on the node-ordering protocol aim to decrease the update time by minimizing the number of rounds. Therefore, we first conduct a prototype implementation of this protocol to study the relationship between the update time and the number of rounds, and then evaluate these algorithms by comparing the latter.

A. A Prototype Implementation

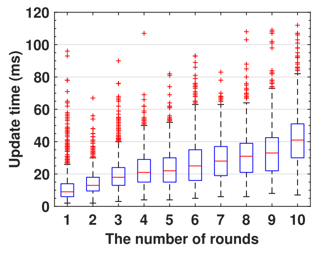
Implementation: we develop a prototype implementation of the node-ordering protocol by using a Floodlight v1.2 [1] controller with Openflow v1.3 [4] on Mininet v2.3 [3]. We add a `RoutingUpdate` module with three REST APIs into Floodlight, `INSTALL`, `UPDATE` and `CLEAR`. The `INSTALL` API provides the interface of the installation of the initial route rules, the `UPDATE` API commands the controller to update the initial routing path to a customized one and the `CLEAR` API deletes a certain rule from the switches. We set the `MATCH` field in the switches only based on the source and destination IP address. In each round, Floodlight first sends an `OFPT_FLOW_MOD` message with the new rule and then an `OFPT_BARRIER_REQUEST` message to each switch that is updated in this round. Upon receiving all the `OFPT_BARRIER_REPLY` messages, the controller starts the updates of next round.

Setup: we perform this experiment on a PC with an Intel i7-7700HQ octa-core processor and 16 GB memory. Each data point is obtained from 1000 runs in order to get the distribution of the update time.

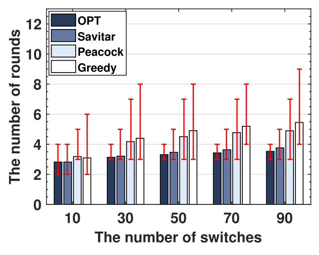
Results: Fig. 7(a) and Fig. 7(b) present the results when the number of switches is 20 and 40, respectively. Both these two figures show that the update time is directly proportional to the number of rounds. Therefore, minimizing the number of rounds can significantly decrease the update time. Comparing Fig. 7(a) and Fig. 7(b), we can find that the update time increases with the number of switches even for the same number of rounds. In addition, the difference of the update time among the different number of rounds also increases as the size of the update instance enlarges. Therefore, decreasing the number of rounds is more important for the update



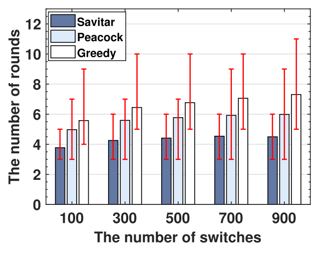
(a) Update time (20 switches).



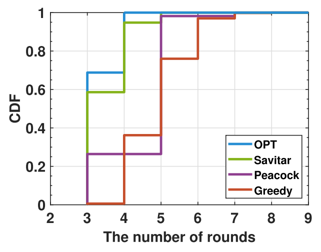
(b) Update time (40 switches).



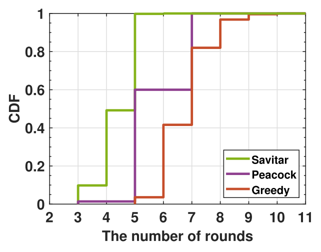
(c) Rounds (small scale).



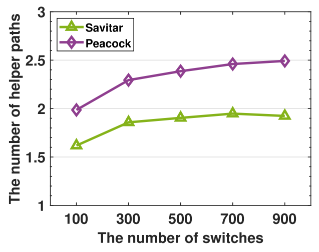
(d) Rounds (large scale).



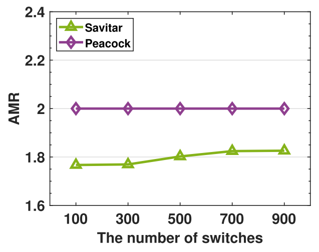
(e) CDF (50 switches).



(f) CDF (500 switches).



(g) Helper paths.



(h) AMR.

Fig. 7. Performance comparison.

instances with large size in practice, which is exactly achieved by our algorithm.

B. Extensive Simulations

In order to observe the difference among the above algorithms for more complex update instances, we conduct extensive simulations to compare the number of rounds computed by different update algorithms. Each data point in this experiment is obtained from 500 different update instances with the same number of switches.

Results: Fig. 7(c) and Fig. 7(d) show the results on a small scale and a large scale, respectively. We cannot obtain the results of OPT on the large scale due to the long running time that it requires. In Fig. 7(c), we can find that Savitar has achieved near optimal performance. Specifically, Savitar only increases the mean number of rounds by 0%, 2.55%, 4.64%, 5.95%, 6.68% compared with OPT, respectively, while implementing the latter requires very long running time and cannot apply to the real-time network updates. In addition, Savitar performs much better than Peacock and Greedy on both a small scale and a large scale. On a small scale, Savitar decreases the mean number of rounds by 20.93% and 25.22% on average compared with Peacock and Greedy, respectively. On a large scale, Savitar decreases the mean number of rounds by 23.96% and 35.06% on average, respectively. For all the update instances in the experiment, Savitar requires at most 6 rounds even when there are 900 switches, while Peacock requires 9 rounds and Greedy requires 11 rounds in the worst case. Therefore, besides the theoretical results, Peacock does not perform well in practice, which is far from OPT and achieves only a slight improvement compared with Greedy while the latter guarantees strong loop-freedom.

In order to derive a better comparison, we present the distribution of the number of rounds for these 500 different update instances where the number of switches is 50 and 500 in Fig. 7(e) and Fig. 7(f), respectively. The distribution computed by Savitar is close to that of OPT. In these 500 update instances with 50 switches, OPT can solve 70% of them and Savitar can solve 58% of them in 3 rounds, respectively. In contrast, Peacock and Greedy can only solve 26% and 1% of these instances in 3 rounds, respectively. When the number of switches increases to 500, Savitar presents greater advantages. Savitar can use at most 4 rounds to solve 50% of these 500 update instance, while it is less than 20% and equal to 0% for Peacock and Greedy, respectively. Moreover, the percentage of the update instances that requires 6 rounds or more is less than 1% for Savitar, while it is about 40% and 60% for Peacock and Greedy, respectively.

In order to schedule relaxed loop-free updates, Savitar consists of two subroutines, one is to compute a relaxed loop-free helper path sequence and the other is to schedule relaxed loop-free updates from hp_i to hp_{i+1}. Actually, Peacock can also be regarded as an algorithm including the above two subroutines. Therefore, we present how Savitar accelerate the loop-free updates by comparing the results of Savitar and Peacock in both the two subroutines. Fig. 7(g) shows the average number of helper paths computed by Savitar and Peacock. We can find that Savitar saves 20.27% unnecessary helper paths on average compared with Peacock. Fig. 7(h) compares the Average Migration number of Rounds to migrate hp_i to hp_{i+1} (we denote it by AMR). The migration from *op* to hp_1 is not included when computing AMR since it requires only one round for any algorithm, i.e.,

$$\mathrm{AMR} = \frac{|\mathrm{R}| - 1}{|\mathrm{HP}|}$$

where R (HP) is a relaxed loop-free update (helper path) sequence computed by the corresponding algorithm. Peacock always requires 2 rounds in order to update hp_i to hp_{i+1} ($i \geq 1$) while Savitar will first check whether one round is feasible, as is shown in Algorithm 2. Fig. 7(h) presents the results and we can find that Savitar saves 10.11% unnecessary rounds in each migration from hp_i to hp_{i+1}.

In addition, we also use the above algorithms to solve the constructed instance $\mathcal{I}_k$ in Sec. III from $k = 3$ to $k = 8$, where the number of nodes is $n = 2^k,\ k \in \mathbb{N}^*$. Savitar and Greedy present the same results, which requires $3, 4, 5, 6, 7, 8$ rounds in turn for these instance, while Peacock requires $5, 7, 9, 11, 13, 15$ rounds. Due to its prohibitively long running time, OPT can only solve $\mathcal{I}_k$ when $k \leq 6$ ($n \leq 64$), which requires $3, 4, 5, 6$ rounds from $k = 3$ to $k = 6$.

VI. Related Work

Loop-Free Route Updates: The two-phase commit protocol [23] is an alternative solution to guarantee loop-freedom. It avoids forwarding loops by ensuring that packets are forwarded along the old routing path or the new routing path, but never a mixture of them at any time point. At its expense, this protocol requires that the switches has to preserve both old and new rules during updates and thus consumes more Ternary Content Addressable Memory (TCAM). In addition, the VLAN field in the packet head may be occupied by this protocol, which is not friendly with the applications that also use this field. Timed SDNs can keep more accurate synchronization and trigger updates at specific time point [21], which can be used to avoid forwarding loops during network updates [27], [29]. However, running Network Time Protocol (NTP) will cause extra overhead in order to synchronize the clocks of all the switches in Timed SDNs.

In contrast, the node-ordering protocol can guarantee loop-free network updates by controlling the update order of all switches, which is not only easy to deploy but also avoids additional resource consumption. The works [10], [11], [20] presented how to find a safe subset of the switches which cannot introduce any loops in each round while all of them require $\Omega(n)$ rounds in the worst case, where n is the number of switches. A new algorithm named FLIP that combines the advantages of the two-phase commit protocol and the node-ordering protocol was designed to accelerate the loop-free updates in [24], but it still cannot avoid extra flow table space overhead. The work [26] proposed a dynamic ordered update scheme to avoid forwarding loops and packet duplication during multicast routing updates. Due to the inherent hardness to guarantee (strong) loop-freedom, a weaker notion of loop-freedom, named relaxed loop-freedom, was introduced in [8], [19]. Our work extends the works [8], [19] by proposing a faster relaxed loop-free update algorithm and solving some open problems.

Congestion-Free Route Updates: In addition to forwarding loops, transient congestion may also occur in some links during network updates, where new flows have arrived while old flows are not migrated yet [9], [17]. SWAN [14] and zUpdate [18] attempt to find a congestion-free update plan by solving a set of linear programs (LPs) in Wide Area Networks (WAN) and Data Center Networks (DCN), respectively. It has been proved that such a congestion-free update plan always exists if all links have a certain slack capacity [14]. However, this condition cannot be always guaranteed in practice. The work [7] proposed a polynomial-time algorithm to judge whether a congestion-free update sequence exists or not even though there are some full links in the network. Different from the above, the work [31] tended to minimize the transient congestion during network updates when considering that congestion-free update plans may not always exist especially in the network with large flows. Besides linear programs, the dependency graph is also one of the main mechanisms to compute a congestion-free update sequence [13], [16], [25], which can accelerate the network updates based on dynamic scheduling. How to conduct congestion-free updates in timed SDN was studied in [28], [30], where the resource dependency graph based on time-extended networks was constructed and a greedy algorithm was developed to compute the update plans.

VII. Conclusion

In this article, we investigated a fundamental problem of scheduling fast relaxed loop-free updates in software defined networks. We solved the long-standing open problem whether $O(1)$-round schedules always exist to guarantee relaxed loop freedom by deriving an $\Omega(\log n)$-round lower bound, which was first proposed and discussed in [19]. In addition, we also presented an algorithm named Savitar that can schedule relaxed loop-free updates within tight lower bounds and accelerate the updates both theoretically and practically.

Our future works will focus on more interesting open problems. Although minimizing the number of rounds to guarantee strong loop freedom has been proved NP-hard [8], [19], it still remains unknown whether this problem is also NP-hard for relaxed loop freedom. Besides, whether we can design some effective approximation algorithms is also one interesting research direction.

Acknowledgment

The authors also would like to thank Tao Chen, Guanhao Wu and Tao Liao for their contributions on the experimental evaluation of this article.

References

[1] *Floodlight*. Accessed: Dec. 6, 2018. [Online]. Available: http://floodlight.openflowhub.org/

[2] *Gurobi*. Accessed: Dec. 18, 2018. [Online]. Available: http://www.gurobi.com/

[3] *Mininet*. Accessed: Dec. 6, 2018. [Online]. Available: http://mininet.org

[4] *Openflow Switch Specification*. Accessed: Dec. 6, 2018. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[5] *Topology Zoo*. Accessed: Dec. 15, 2018. [Online]. Available: http://www.topology-zoo.org/

[6] B. Bollobás, *Random Graphs*. Cham, Switzerland: Springer, 1998, pp. 215–252.

[7] S. Brandt, K.-T. Forster, and R. Wattenhofer, "On consistent migration of flows in SDNs," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.

[8] K.-T. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid, "Loop-free route updates for software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 328–341, Feb. 2018.

[9] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1435–1461, 2nd Quart., 2019.

[10] K.-T. Forster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *Proc. IFIP Netw. Conf. (IFIP Netw.) Workshops*, May 2016, pp. 1–9.

[11] K.-T. Forster and R. Wattenhofer, "The power of two in consistent network updates: Hard loop freedom, easy flow migration," in *Proc. 25th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Aug. 2016, pp. 1–9.

[12] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, *Software-Defined Networking (SDN): Layers and Architecture Terminology*, document RFC 7426, 2015, pp. 1–35.

[13] X. He *et al.*, "Coeus: Consistent and continuous network update in software-defined networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Jul. 2020, pp. 1509–1518.

[14] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 15–26.

[15] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM Int. Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2013, pp. 3–14.

[16] X. Jin *et al.*, "Dynamic scheduling of network updates," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 539–550.

[17] D. Li, S. Wang, K. Zhu, and S. Xia, "A survey of network update in SDN," *Frontiers Comput. Sci.*, vol. 11, no. 1, pp. 4–12, Feb. 2017.

[18] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "ZUpdate: Updating data center networks with zero loss," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 411–422.

[19] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proc. ACM Symp. Princ. Distrib. Comput. (PODC)*, 2015, pp. 13–22.

[20] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. 12th ACM Workshop Hot Topics Netw. (HotNets-XII)*, 2013, pp. 1–7.

[21] T. Mizrahi and Y. Moses, "Software defined networks: It's about time," in *Proc. IEEE 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[22] S. Nobari, X. Lu, P. Karras, and S. Bressan, "Fast random graph generation," in *Proc. 14th Int. Conf. Extending Database Technol. (EDBT/ICDT)*, 2011, pp. 331–342.

[23] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2012, pp. 323–334.

[24] S. Vissicchio and L. Cittadini, "FLIP the (Flow) table: Fast lightweight policy-preserving SDN updates," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.

[25] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free consistent data plane update in software defined networks," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.

[26] G. Wu, X. Gao, T. Chen, H. Zhou, L. Kong, and G. Chen, "Shifter: A consistent multicast routing update scheme in software-defined networks," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 346–355.

[27] J. Zheng, G. Chen, S. Schmid, H. Dai, and J. Wu, "Chronus: Consistent data plane updates in timed SDNs," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 319–327.

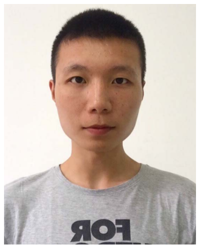
[28] J. Zheng, G. Chen, S. Schmid, H. Dai, J. Wu, and Q. Ni, "Scheduling congestion- and loop-free network update in timed SDNs," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2542–2552, Nov. 2017.

[29] J. Zheng *et al.*, "Scheduling congestion-free updates of multiple flows with chronicle in timed SDNs," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 12–21.

[30] J. Zheng *et al.*, "Congestion-free rerouting of multiple flows in timed SDNs," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 968–981, May 2019.

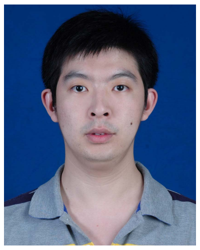
[31] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proc. IEEE 23rd Int. Conf. Netw. Protocols (ICNP)*, Nov. 2015, pp. 1–10.

[32] H. Zhou, X. Gao, J. Zheng, and G. Chen, "A tight lower bound for relaxed loop-free updates in SDNs," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 91–100.



Hao Zhou received the B.S. degree in computer science and technology from Nanjing University, China, in 2017, and the M.S. degree in computer science and technology from Shanghai Jiao Tong University, China, in 2020. His research interests include data center, software defined networks, and cloud computing. He focuses on utilizing approximation algorithm, online algorithm, and machine learning to solve the network optimization problem.

Xiaofeng Gao (Member, IEEE) received the B.S. degree in information and computational science from Nankai University, China, in 2004, the M.S. degree in operations research and control theory from Tsinghua University, China, in 2006, and the Ph.D. degree in computer science from The University of Texas at Dallas, USA, in 2010. She is currently a Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. She has published more than 200 peer-reviewed articles in the related area, including well-archived international journals such as the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, and also in well-known conference proceedings such as WWW, SIGKDD, INFOCOM, ICDE. Her research interests include wireless communications, data engineering, and combinatorial optimizations. She has served on the Editorial Board of *Discrete Mathematics, Algorithms and Applications*, and as the PCs and peer reviewers for a number of international conferences and journals.



Jiaqi Zheng (Member, IEEE) received the B.S. and M.S. degrees in computer science and technology from the Nanjing University of Aeronautics and Astronautics, and the Ph.D. degree in computer science and technology from Nanjing University. He was a Research Assistant with The City University of Hong Kong in 2015 and a Visiting Scholar with Temple University in 2016. He is currently a Research Assistant Professor from the Department of Computer Science and Technology, Nanjing University, China. His research area is computer networking, particularly data center networks, SDN/NFV, and machine learning systems. He received the Best Paper Award from IEEE ICNP 2015 and Outstanding Doctoral Dissertation Award from CCF. He is a member of the ACM.

Guihai Chen (Member, IEEE) received the B.S. degree in computer software from Nanjing University in 1984, the M.E. degree in computer applications from Southeast University in 1987, and the Ph.D. degree in computer science from The University of Hong Kong in 1997. He had been invited as a Visiting Professor from the Kyushu Institute of Technology, Japan; The University of Queensland, Australia; and Wayne State University, USA. He is currently a Distinguished Professor with Shanghai Jiao Tong University. He has published more than 350 peer-reviewed articles, and more than 200 of them are in well-archived international journals such as the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, the ACM/IEEE TRANSACTIONS ON NETWORKING, and *ACM Transactions on Sensor Networks*, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNEXT, and AAAI. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture, and data engineering. He is a CCF Fellow. He received several best paper awards including ICNP 2015 Best Paper Award. His articles have been cited for more than 15 000 times according to Google Scholar.