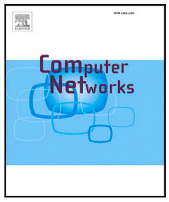
Computer Networks 228 (2023) 109755

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



Traffic-aware efficient consistency update in NFV-enabled software defined networking

Pan Li [a], Guiyan Liu [b,*], Songtao Guo [b,*], Yue Zeng [c]

[a] College of Electronic and Information Engineering, Southwest University, Chongqing, 400715, China
[b] College of Computer Science, Chongqing University, Chongqing, 400044, China
[c] Department of Computer Science and Technology, Nanjing University, Nanjing, 210023, China

ARTICLE INFO

Keywords:
Consistency update
Policy consistency
Transient congestion
Software defined networking
Network function virtualization

ABSTRACT

In network function virtualization(NFV)-enabled software defined networks, the controller needs to frequently update the flow forwarding rules in the data plane to adapt to dynamic changes in network topologies or service requests. However, inconsistent rule updates may lead to blackholes, loops, transient congestion or policy violations (e.g., packets do not traverse designated network functions in a specific order), resulting in service interruption and throughput degradation. Therefore, this paper proposes an effective rule consistent update mechanism to avoid the above four problems simultaneously, while improving network throughput and satisfying user requests. Specifically, we first build three effective models to avoid blackholes, loops, and policy violations. Then, considering that network function nodes may change the sizes of their processed flows, we build a congestion avoidance model based on traffic changes to avoid congestion, which can reduce unnecessary rule update delays and packet loss. Subsequently, we prove that the consistent update problem constructed above is NP-hard, and then design an effective heuristic rule consistent update algorithm to obtain the rule update sequence that can simultaneously avoid blackholes, loops, congestion, and policy violations. Extensive trace-driven simulation results show that compared with the existing update methods, our proposed method can improve the success rate by up to 20.6% and reduce the maximum link utilization by up to 7.5%.

1. Introduction

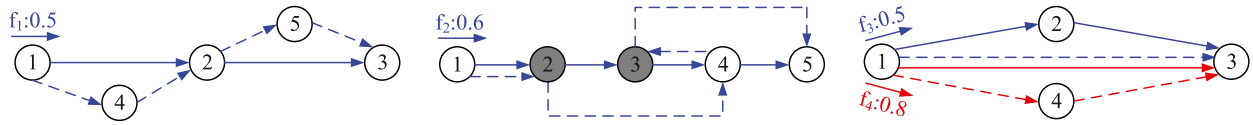
Software defined networking (SDN) has been widely used in load balancing and failure recovery by separating the control and data planes, which greatly simplifies network management [1]. To improve network performance and implement network policies, service providers may apply network functions based services, such as firewalls, VPN agents, and intrusion detection systems [2]. However, traditional network functions are implemented by dedicated hardware devices, with disadvantages such as high cost and difficulty in scaling [3]. Network function virtualization (NFV) changes the implementation of network functions by running software on general hardware, making deployment of network functions more flexible [4]. Benefiting from SDN and NFV, service providers can efficiently manage networks and flexibly deploy network functions, thereby reducing capital expenditures and operating expenses [5].

In NFV-enabled SDNs, the routing paths of flows may be updated frequently due to load balancing or topology changes to improve service performance. In a dynamic environment, cloud resources may need to be reconfigured to save energy [6] or costs [7], which may also result in updates to flow routing paths. Moreover, topology changes may be caused by node and link failures, in which case the routing paths of flows on failed links or nodes need to be changed in order to maintain service. In order to update the flow path, the SDN controller with a global network view in the control plane needs to instruct the switches in the data plane to update their flow forwarding rules. However, rule updates may be inconsistent due to the difference of the processing delay among switches and that of the communication delay between the switch and the controller [8,9].

Unfortunately, inconsistent rule updates may result in the following four problems. (1) Blackhole problem: A blackhole may occur, when packets arrive at a switch and the switch does not have corresponding forwarding rules, which may result in packet loss and traffic transmission interruption [10]. (2) Loop problem: A loop occurs when packets are forwarded back and forth among switches, which may result in packet loss [11]. (3) Policy violation problem: A policy is violated when packets do not pass through specified network function nodes in a specific order, which is unacceptable for the security-critical service [12].

* Corresponding authors.
E-mail addresses: gyliu@cqu.edu.cn (G. Liu), guosongtao@cqu.edu.cn (S. Guo).

https://doi.org/10.1016/j.comnet.2023.109755
Received 19 October 2022; Received in revised form 25 March 2023; Accepted 31 March 2023
Available online 6 April 2023



(a) Blackhole (b) Loop and policy violation (c) Transient congestion

Fig. 1. Examples are given to illustrate blackholes, loops, policy violations and transient congestion caused by inconsistent updates. The solid and dashed lines represent the old and new paths respectively. (1) Blackhole: If switch 1 has modified the rule of flow f_1, and switch 4 has not added its rule, switch 4 is regarded as a blackhole. (2) Loop: If switch 4 has updated the rule of flow f_2, and switch 3 still uses its old rule, a loop $4 \rightarrow 3 \rightarrow 4$ may occur. (3) Policy violation: Gray nodes 2 and 3 are network function nodes. If switch 2 has updated the rules of flow f_2 and switch 4 has not updated its rules, the packets of flow f_2 will arrive at the destination from $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$, causing network function node 3 to be bypassed. (4) Transient congestion: Assuming that the capacity of each link is 1 unit, and the flow sizes of f_3 and f_4 are 0.5 and 0.8. If switch 1 has updated the rule for flow f_3 but not for flow f_4, the total load on link $(1,3)$ is $0.5+0.8=1.3$, which exceeds the link capacity.

(4) Transient congestion problem: Transient congestion means that during rule updates, the total load on the link may exceed the link capacity, which may also lead to packet loss and performance degradation [1]. To elaborate on the above four issues, we give examples as shown in Fig. 1.

Currently, some existing works [13,14] have studied how to avoid blackholes and loops, and have proposed efficient heuristic algorithms to calculate the update order of rules. Moreover, several novel methods [1,8,9,15,16] have been proposed to avoid blackholes, loops and congestion simultaneously. However, these methods ignore policy consistency and may fail to meet user requirements. For example, when the access to the Intranet server needs to be strictly restricted, the incoming flows may first go through the firewall (FW) and then through the intrusion detection system (IDS) [5]. If FW or IDS is bypassed by packets, it may cause the network to be attacked, which is unacceptable for security-critical environments [12]. Therefore, some works [12,17–20] have studied how to avoid blackholes, loops and policy violations. However, the above works do not consider the simultaneous avoidance of blackholes, loops, congestion, and policy violations, which may lead to unmet service requirements and throughput reduction.

Therefore, in this paper, we propose a novel rule consistent update mechanism that can simultaneously avoid blackholes, loops, policy violations, and congestion, which helps improve network throughput and meet service requests. Specifically, in order to avoid blackholes, we divide the new path into several segments, and then build an update dependency graph according to these segments. The dependency graph represents the update order of rules. To avoid loops, we identify the critical nodes that determine whether the loop is formed, and then build a dependency graph based on the critical nodes. To avoid policy violations, we first analyze two cases where a waypoint (i.e., network function node) may be bypassed, i.e., one case is that a node is the precursor of a waypoint in the old path and the successor of that waypoint in the new path, and the other case is opposite. Then, we construct the corresponding dependency graph for the above situation. Moreover, to resolve conflicts between the above dependency graphs, we use a tag matching method that instructs the switch to apply which rules based on tags, which helps improve the success rate.

To avoid congestion, we build a congestion avoidance model based on traffic changes to ensure that incoming traffic on each link does not exceed its capacity. Because, after a flow is processed by waypoints, its size may be changed [21,22]. For example, the Citrix NetScaler SD-WAN WAN optimizer can compress the processed traffic to 20% of its original traffic, and the BCH(63,48) encoder for satellite communications can increases the traffic volume of the processed flow by 31% due to the checksum overhead [22]. If the traffic change is ignored when avoiding congestion, it may lead to a long update time[1] or link congestion.[2] Thus, we introduce traffic change factors and delay the update of some flows on potentially congested links to avoid congestion, which can effectively reduce update time and packet loss. Subsequently, we design an effective rule consistent update algorithm to find an update sequence that can avoid blackholes, loops, congestion, and policy violations at the same time. The main contributions of this paper are as follows:

- We analyze the reasons for the problems of blackholes, loops, and policy violation during the update, and construct three corresponding dependency graphs to avoid these problems. Subsequently, we build a conflict avoidance model to resolve conflicts between dependency graphs, which can improve the success rate of consistent updates.
- We reveal the cause of congestion considering traffic change and build a congestion avoidance model based on traffic change factors, which can effectively reduce update time and packet loss. And then we prove that the consistent update problem constructed above is NP-hard.
- We design an effective heuristic rule update algorithm to obtain the rule update order that can simultaneously avoid blackholes, loops, congestion, and policy violations.
- Extensive trace-driven simulation results show that compared with the existing update methods, our proposed method can improve the success rate by up to 20.6% and reduce the maximum link utilization by up to 7.5%.

The rest of this paper is organized as follows. Section 2 discusses the related works of consistent update. In Section 3, we construct the corresponding avoidance models to solve the four problems of inconsistent update. In Section 4, we propose a rule consistent update algorithm. Section 5 introduces simulation settings and results analysis. Finally, Section 6 concludes this paper.

2. Related work

In this section, we introduce the existing consistent update methods, which can be roughly divided into three categories. The concise comparison between our proposed method and existing methods is listed in Table 1.

The first category is how to avoid blackholes and loops during the update. Basta et al. [23] studied how to avoid loops while minimizing

[1] For example, switch 1 depicted in Fig. 1(c) is a waypoint, and its traffic change factor is 0.7. The sizes of flows f_3 and f_4 are 0.5 and 0.8 respectively. If the traffic change factor is ignored, the load on link $(1,3)$ in the worst case is $0.5+0.8=1.3$, thus the update of flow f_3 will be delayed to avoid congestion. This is unnecessary if the traffic change factor is considered, which can prolong the unnecessary update time.

[2] For example, the traffic change factor of switch 1 in Fig. 1(c) is 1.1, and the sizes of flows f_3 and f_4 are 0.4 and 0.6 respectively. If the traffic change factor is ignored, they can be updated simultaneously, and the actual load link $(1,3)$ is $(0.4+0.6)*1.1=1.1$, which exceeds the link capacity and causes congestion.

Table 1
Related work.

Reference	Objective (avoid)					Methods		
	Blackhole	Loop	Congestion	Policy violation	Conflict	Ordered replacement	Tag matching	Combine ordered replacement and tag matching
[14]	✔	✔	✘	✘	✘	✔	✘	✘
CSU [15]	✔	✔	✔	✘	✘	✔	✘	✘
DRF [12]	✔	✔(partial)	✘	✔	✘	✔	✘	✘
DSF [12]	✔	✔	✘	✔	✘	✔	✘	✘
FLIP [19]	✔	✔	✘	✔	✔(partial)	✘	✘	✔
This paper	✔	✔	✔(traffic change)	✔	✔	✘	✘	✔

the number of controller-switch interactions. Zhou et al. [24] introduced a concept of relaxed loop-free to speed up the updating. But these methods ignore other consistent properties. To avoid blackholes and loops, Maity et al. [13] proposed a multilevel queuing approach to ensure package-level consistency and Li et al. [14] proposed a rule composition method based on abstract algebra to reduce the number of rule operations.

The second category is how to avoid blackholes, loops, and congestion during the update. Jin et al. [8] used the global resource dependency graph to dynamically update the rules to avoid blackholes, loops, and congestion, but this method is time-consuming. In order to reduce the update time, some works [1,25] divided the global update dependencies into local update dependencies, Coeus [16] eliminated redundant operations in update events, DART [26] reduced the data plane load, and Chronus^{+} [9] used the switch buffer to cache the overload flows. Based on [1], Song et al. [15] designed a consistent scheduling update (CSU) algorithm to further reduce the updating time by updating some of the flows that need to be moved in the potential congestion link in advance. In addition, Bera et al. [27] proposed an adaptive flow rule placement scheme to maximize the number of flows while increasing the visibility of the network. However, the above methods ignore that the size of a flow may change after being processed by the network function nodes, leading to packet loss and unnecessary update delays.

The third category is how to avoid blackholes, loops, and policy violations during the update. Reitblatt et al. [18] guaranteed blackhole-free, loop-free, and policy consistency by tagging packets on the ingress switch and installing matching rules on other switches along the path. However this approach requires the old and new rules to be installed on the switch simultaneously, which increases TCAM (ternary content addressable memory) overhead. To reduce TCAM overhead, Ludwig et al. [12,17] designed two novel update algorithms, DRF and DSF, to replace the old rules by calculating a specific order. The DSF algorithm ensures that there are no loops at any time, and the DRF algorithm only ensures that there are no loops in the current path from the source node. However, the applicability of the ordered replacement method (i.e., replacing the old rules by calculating a specific order) is limited, and the order guaranteeing both loop-free and policy consistency may not exist [19,20]. To resolve the conflict between policy consistency and loop-free, Vissicchio et al. [19,20] proposed an efficient update method called FLIP that combines ordered replacement and tag matching to avoid blackholes, loops, and policy violations. However, this method allows packets to traverse a limited number of loops when dealing with conflicts, which may result in packet loss.

In all, almost all of the previous works only focused on dealing with some of the four problems of blackholes, loops, congestion, and policy violations rather than consider solving these four problems simultaneously, which may lead to service interruption and packet loss. Therefore, in order to improve network throughput and meet user demands, we propose an effective rule update mechanism, which can simultaneously avoid blackholes, loops, congestion and policy violations.

Table 2
List of key notations.

Notation	Description
P_f^n	New path of flow f
P_f^o	Old path of flow f
G_f	The Segment node set of flow f
S_f	Segments of the new path of flow f
H_f	The loops formed by the old and new paths of flow f
W	The set of waypoints that each packet must pass through
R_f	Circles between dependency graphs of flow f
$y_f^{(u,v)}$	The size of flow f on the link (u,v)
y_f^{u-}, y_f^{u+}	The size of flow f before and after being processed by node u
$c(u,v)$	The capacity of link (u,v)
$\Gamma(u,v)$	The set of flows that need to be removed from the link (u,v), but have not yet been removed from the link (u,v)
$\Psi(u,v)$	The set of unchanged flows on link (u,v)
$\Phi(u,v)$	The set of new flows that have been moved into the link (u,v)

3. Problem formulation

In this section, we formulate the consistent update (CSUD) problem, which is able to avoid blackholes, loops, policy violations, and congestion during the update. In the following, rule updates on nodes are simplified as node updates. For convenience, we summarize important notations in Table 2.

3.1. Blackhole-free update

To ensure blackhole-free updates, we propose a blackhole avoidance method, which is mainly divided into three steps.

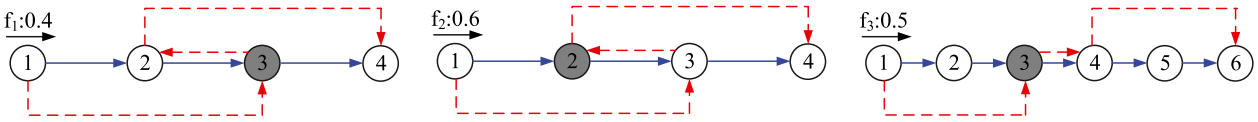
In the first step, we find the segment nodes G_f that control the flow f switching path, where the segment nodes [25] refer to common nodes of old path P_f^o and new path P_f^n with different next hops. That is, each segment node $g_f \in G_f$ is a common node of the old and new paths, and it has different next hops in the old and new paths. Similar to [4,5], the old and new paths are generated based on the Dijkstra algorithm [28].[3] Thus, the segment node set G_f can be expressed as

$$G_f = \{g_f | g_f \in P_f^o \cap P_f^n, n(g_f, P_f^o) \neq n(g_f, P_f^n)\}, \tag{1}$$

where $n(g_f, P_f^o)$ and $n(g_f, P_f^n)$ represent the next hops of node g_f in the old and new paths.

In the second step, we divide the new path P_f^n into several segments, called segment paths, where segment paths refer to the paths after dividing the new path based on segment nodes. For example, there is a new path $A \rightarrow B \rightarrow C$, segment node $G_f = B$, then the segment paths are $A \rightarrow B$, $B \rightarrow C$. After the division, each segment path $s \in S_f$ starts with a segment node or source node $P_f^n.start$ and ends with a segment node or destination node $P_f^n.end$. There are no overlapping

[3] Note that our solution is equally compatible with old and new paths generated by other algorithms.



(a) Case 1 (b) Case 2 (c) Case 3

Fig. 2. Bypass waypoint. The blue solid and red dashed lines represent the old and new paths of flow f_3, and the gray nodes are waypoints. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

nodes between segments, except that adjacent segments share the start or end node. Thus, the segment path set S_f can be expressed as

$$\begin{aligned} S_f = \{ & s \mid s \subseteq P_f^n, s.start \in G_f \cup P_f^n.start, \\ & s.end \in G_f \cup P_f^n.end, \forall s' \in S_f, \\ & s \cap s' = \emptyset \ or \ = s.end \equiv s'.start \\ & or \ = s.start \equiv s'.end \}. \end{aligned} \tag{2}$$

In the third step, we construct update dependency graphs based on segment paths S_f to avoid blackholes. In each segment path, as long as the nodes $s \setminus \{s.start, s.end\}$ in the segment path are updated before the starting node $s.start$, it can be guaranteed that there are no blackholes [25]. Therefore, the dependency graph in each segment path can be constructed as follows

$$d(s) = s.start \rightharpoonup \{n | n \in s \setminus \{s.start, s.end\}\}. \tag{3}$$

Algorithm 1 Blackhole Avoidance

Input: flow f, old path P_f^o, new path P_f^n
Output: blackhole-free dependency graph BH, segment paths S_f, segment nodes G_f

1: Get segment nodes G_f according to Eq. (1);
2: $k = P_f^n[0]$;
3: **while** $k \in P_f^n$ **do**
4: $s = k$;
5: $m =$ the next hop of k on P_f^n;
6: **while** $m \neq \emptyset \wedge m \notin G_f$ **do**
7: $s = s \rightarrow m$;
8: $k = m, m =$ the next hop of k on P_f^n;
9: **end while**
10: $s = s \rightarrow m, S_f = S_f \cup s$;
11: Get the blackhole-free dependency graph $d(s)$ according to Eq. (3);
12: $BH = BH \cup d(s), k = m$;
13: **end while**

The details of constructing the blackhole-free dependency graph are shown in Algorithm 1. First, we obtain the segment nodes G_f according to Eq. (1). Then, we traverse the new path of flow f, adding non-segment nodes to a segment until meeting a segment node or destination node (lines 3–10). We finally obtain the blackhole-free dependency graph $d(s)$ according to Eq. (3). For example, as shown in Fig. 1(a), the segment nodes of the flow f_1 are nodes 1 and 2. Then according to the segment nodes, we divide the new path of flow f_1 into two segments: $1 \rightarrow 4 \rightarrow 2$ and $2 \rightarrow 5 \rightarrow 3$. Finally, we obtain the blackhole-free dependency graphs of these two segments as $1 \rightharpoonup \{4\}$ and $2 \rightharpoonup \{5\}$. That is, the update of node 4 (or node 5) is earlier than the update of node 1 (or node 2).

3.2. Loop-free update

To ensure loop-free updates, we propose a loop avoidance method, which consists of three steps. First, we find the loops H_f formed by the old and new paths according to the method of strong connected components [1]. Then, we identify the critical nodes that determine whether the loop $h_f \in H_f$ is formed or not. Among them, h_f^x is the segment node in h_f and its next hop in the new path is not in h_f, h_f^l is the segment node in h_f and its next hop in the old path is not in h_f. Finally, we construct a loop-free dependency graph based on critical nodes, aiming to make node h_f^x update earlier than node h_f^l to avoid loops. The dependency graph is constructed as follows

$$d(h) = h_f^l \rightharpoonup h_f^x. \tag{4}$$

Algorithm 2 Loop Avoidance

Input: flow f, old path P_f^o, new path P_f^n, segment nodes G_f
Output: loop-free dependency graph L

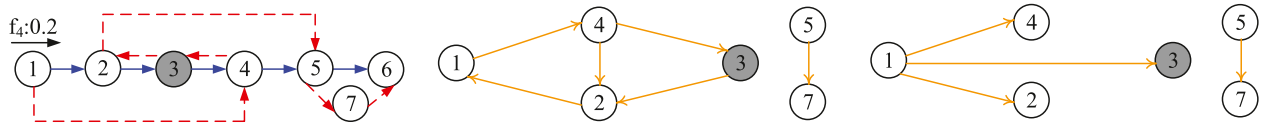
1: Calculate Loops H_f by strongly connected components;
2: **for** each $h_f \in H_f$ **do**
3: Find critical nodes h_f^l and h_f^x in the loop h_f;
4: Get the loop-free dependency graph $d(h)$ according to Eq. (4);
5: $L = L \cup d(h)$;
6: **end for**

The details of constructing the loop-free dependency graph are shown in Algorithm 2. First, the loops H_f formed by the new and old paths are obtained by strong connected components [25]. Then we find the critical nodes h_f^l and h_f^x in the loop h_f, and finally construct a loop-free dependency graph $d(h)$ according to Eq. (4). For example, in Fig. 1(b), the old and new paths of flow f_2 form a loop $4 \rightarrow 3 \rightarrow 4$, the critical nodes h_f^x and h_f^l in this loop are nodes 3 and 4, and then the loop-free dependency graph is $4 \rightharpoonup 3$. This means that node 4 can be updated only after node 3 is updated.

3.3. Policy consistency update

To ensure the consistent update of policies, we need to meet the following two conditions, (i) all waypoints will be passed by the packets (i.e., the waypoint inclusion constraint), and (ii) the order of traversing the waypoints by the packets during the update is unchanged (i.e., the waypoint order constraint). However, path changes may cause the above two constraints to be violated, that is, waypoints are bypassed, or the order of traversing waypoints is changed. We first discuss the situation where waypoints are bypassed, as shown in Fig. 2. Case 1: In the old path, the packets traverse node n before waypoint w, and in the new path, the packets traverse node n after the waypoint w. It can be found that if node n is updated earliest, the packets will bypass the waypoint w. For example, as shown in Fig. 2(a), the old and new paths of flow f_1 are $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 2 \rightarrow 4$, where waypoint $w = 3$, node $n = 2$. For waypoint 3, in the old path, the packets traverse node 2 before node 3, but in the new path, the packets traverse node 2 after node 3. If node 2 is updated earliest, the packets will reach the destination node 4 directly via nodes 1 and 2, which will cause the packets to bypass waypoint 3.

Case 2: Contrary to Case 1, in the old path, the packets visit waypoint w before node n, and in the new path, the packets visit the waypoint w after node n. We can find that if node n is updated last, the packets will bypass the waypoint w. For instance, in Fig. 2(b) [12,17], the old and new paths of flow f_2 are $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 2 \rightarrow 4$, where waypoint $w = 2$, node $n = 3$. For waypoint 2, in the old path, the packets visit the waypoint 2 before node 3, and in the new path, the packets visit the waypoint 2 after node 3. If node 3 is updated last, the



(a) Path (b) Update dependency graph (c) New update dependency graph

Fig. 3. Conflict between dependency graphs. The orange arrow indicates the update relationship between nodes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

packets will directly pass through nodes 1 and 3 to reach destination node 4, which will cause the waypoint 2 to be bypassed by the packets.

Case 3: In both the old and new paths, the packets traverse waypoint w before (or after) node n. In this case, the update of node n will not cause waypoint w to be bypassed. As shown in Fig. 2(c), the old and new paths of flow f_3 are $1 \to 2 \to 3 \to 4 \to 5 \to 6$ and $1 \to 3 \to 4 \to 6$, where waypoint $w = 3$, node $n = 1$ or 4. For waypoint 3, in both the old and new paths, the packets visit the waypoint 3 after node 1 and before node 4. We can find that whether node 1 or node 4 is updated first, the waypoint 3 will not be bypassed.

Based on the above analysis, we can find that *a waypoint may be bypassed when the order of the waypoint relative to other nodes changes in the old and new paths*. Next, we solve this problem.

For Case 1, to avoid packets bypassing the waypoints, for each waypoint $w \in W$, we first find the critical node c_f^w that determines whether the packets bypass the waypoint w. That is, in the old path, the packets traverse c_f^w before w, and vice versa for the new path. Then, we construct a waypoint-enforcement dependency graph to avoid waypoints being bypassed. In this dependency graph, the update of the critical node c_f^w is later than the update of the node $g_{f,o}^w$, and $g_{f,o}^w$ is the last segment node in the old path that controls the packets to enter the node c_f^w. Thus, the dependency graph is constructed as follows

$$d(w) = c_f^w \rightharpoonup g_{f,o}^w. \tag{5}$$

For Case 2, similar to case 1, for each waypoint $w \in W$, we first find the critical node c_f^w that determines whether the packet bypasses the waypoint w. That is, in the old path, the packets visit w before c_f^w, and vice versa for the new path. Then we construct a waypoint-enforcement dependency graph to avoid packets bypassing the waypoint. In this dependency graph, the update of the critical node c_f^w is earlier than the update of the node $g_{f,n}^w$, and $g_{f,n}^w$ is the latest segment node in the new path that controls the packets entering the node c_f^w. The dependency graph is constructed as follows

$$d(w) = g_{f,n}^w \rightharpoonup c_f^w. \tag{6}$$

By constructing the above dependency graph, we meet the waypoint inclusion constraint, that is, the packets do not bypass the waypoints. Next, we need to satisfy the waypoint order constraint, that is, the order of waypoints being traversed by the packets will not be changed. Interestingly, the waypoint order constraint can be satisfied after satisfying both the waypoint inclusion constraint and the loop-free constraint (i.e., the forwarding path has no loops), which has been proved in Theorem 8 in [12]. Therefore, the dependency graph we constructed can satisfy both the waypoint inclusion constraint and the waypoint order constraint, because the loop-free constraint has been satisfied in Section 3.2.

The details of constructing the waypoint-enforcement dependency graph are shown in Algorithm 3. For each waypoint $w \in W$, we first find the critical node c_f^w that determines whether the waypoint is bypassed. That is, if the packets visit w after node i in the old path, and the packets visit w before i in the new path, the node i is regarded as a critical node c_f^w (lines 3–4). Then we find node $g_{f,o}^w$, which is the last segment node in the old path that controls the packets to enter c_f^w, and get the waypoint-enforcement dependency graph according to Eq. (5) (lines 5–6). If the packets visit w before node i in the old path, and the packets visit w after i in the new path, node i is a critical node c_f^w (lines 8–9). Then we find node $g_{f,n}^w$, which is the last segment node in the new path that controls the packets to enter c_f^w, and get the waypoint-enforcement dependency graph according to Eq. (6) (lines 10–11). As an example in Fig. 2(a), for the waypoint $w = 3$, the nodes c_f^w and $g_{f,o}^w$ are nodes 2 and 1, and then the dependency graph $d(w)$ is $2 \rightharpoonup 1$. That is, only after node 1 is updated, node 2 can be updated. As shown in Fig. 2(b), for the waypoint $w = 2$, the nodes c_f^w and $g_{f,n}^w$ are nodes 3 and 1, and the dependency graph $d(w)$ is $1 \rightharpoonup 3$. That is, node 1 can be updated only after node 3 is updated.

Algorithm 3 Waypoint Enforcement

Input: flow f, old path P_f^o, new path P_f^n, segment nodes G_f, waypoint set W
Output: waypoint-enforcement dependency graph WP

```
1: for each w ∈ W do
2:   for each i ∈ P_f^n do
3:     if (index_old(i) < index_old(w)) ∧ (index_new(i) > index_new(w)) then
4:       c_f^w = i;
5:       Find node g_{f,o}^w;
6:       Get the waypoint-enforcement dependency graph d(w) according to Eq. (5);
7:     end if
8:     if (index_old(i) > index_old(w)) ∧ (index_new(i) < index_new(w)) then
9:       c_f^w = i;
10:      Find node g_{f,n}^w;
11:      Get the waypoint-enforcement dependency graph d(w) according to Eq. (6);
12:    end if
13:  end for
14:  WP = WP ∪ d(w);
15: end for
```

3.4. Conflict avoidance between dependency graphs

We construct dependency graphs according to Sections 3.1, 3.2, and 3.3 above to avoid blackholes, loops, and policy violations, but these dependencies may conflict. As shown in Fig. 3(a), the flow f_4 is migrated from $1 \to 2 \to 3 \to 4 \to 5 \to 6$ to $1 \to 4 \to 3 \to 2 \to 5 \to 7 \to 6$, where node 3 is the waypoint. According to Eqs. (1)–(6), the blackhole-free, loop-free and waypoint-enforcement dependency graphs we constructed are $d(s) = \{5 \rightharpoonup 7\}$, $d(h) = \{4 \rightharpoonup 3, 3 \rightharpoonup 2, 4 \rightharpoonup 2\}$ and $d(w) = \{2 \rightharpoonup 1, 1 \rightharpoonup 4\}$ respectively. Thus the rule update dependency graph $D_{f_4} = d(s) + d(h) + d(w)$ of flow f_4 is depicted in Fig. 3(b). We find that there are two circles $4 \rightharpoonup 3 \rightharpoonup 2 \rightharpoonup 1 \rightharpoonup 4$ and $4 \rightharpoonup 2 \rightharpoonup 1 \rightharpoonup 4$ in the dependency graph D_{f_4} because conflicts between $d(h)$ and $d(w)$. This situation will result in unable to find the correct update order to avoid blackholes, loops, and policy violations.

To eliminate the conflicts between the dependency graphs, we first use the method of strong connected components to find the conflicts between the dependency graphs, that is, the circles R_f in the dependency graph $D_f = d(s) + d(h) + d(w)$. Then, we eliminate the circles by removing the dependencies in the circles from the dependency graph D_f. However, removing these dependencies may result in failure to

ensure that there will be no blackholes, no loops, and policy consistency during the update. *To deal with this problem, we adopt the method of tag matching to ensure consistency*. The processing of tag matching is as follows. We first add matching rules on the matching nodes[4] $\{m_f\}$ to identify the tagged packets, and then update the rule (i.e., modify the existing old rule to implement the new one) on the tag node t_f to mark and forward packets. When dealing with our problem, the t_f is the first node that appears in the new path in a circle $r \in R_f$, and $\{m_f\}$ is other nodes in r except t_f. Note that a forwarding rule, such as an OpenFlow flow entry [29], can support first adding tags to packets and then forwarding them. For intuition, we construct a conflict avoidance dependency graph $d(r)$ to represent the process of tag matching, as shown below:

$$d(r) = t_f \rightharpoonup \{m_f \mid m_f \in r \setminus t_f\}. \tag{7}$$

Algorithm 4 Conflict Avoidance

Input: flow f, dependency graph D_f
Output: new dependency graph D_f, I_f

1: **while** True **do**
2: Calculate circles R_f in D_f by strong connected components;
3: **if** $R_f = \emptyset$ **then**
4: break;
5: **end if**
6: Select the circle $r \in R_f$ with the largest number of nodes;
7: $I_f = I_f \cup r$;
8: Delete the dependencies in the circle r from D_f;
9: Get the conflict avoidance dependency graph $d(r)$ according to Eq. (7);
10: $D_f = D_f \cup d(r)$;
11: **end while**

The details of constructing the conflict avoidance dependency graph are shown in Algorithm 4. The main processes of the algorithm are as follows. We get the circles R_f in the dependency graph D_f by strong connected components. If there are circles in D_f that means there are conflicts in D_f. To deal with conflicts, we select a circle $r \in R_f$ with the largest number of nodes. For the circle r, we first delete the dependencies in the circle r from the dependency graph D_f, and then get the conflict avoidance dependency graph $d(r)$ according to the Eq. (7) (lines 8–9). Repeat the above steps until there are no circles in the new dependency graph D_f.

Next, we give an example to facilitate the understanding of the above algorithm, as shown in Fig. 3(b). There are two circles in the dependency graph, such as $4 \rightharpoonup 3 \rightharpoonup 2 \rightharpoonup 1 \rightharpoonup 4$ and $4 \rightharpoonup 2 \rightharpoonup 1 \rightharpoonup 4$. For circle $4 \rightharpoonup 3 \rightharpoonup 2 \rightharpoonup 1 \rightharpoonup 4$, we first delete the dependencies in the circle from the dependency graph, and then get the tag node as node 1 and the matching nodes as nodes 2, 3 and 4. The conflict avoidance dependency graph is $1 \rightharpoonup \{2, 3, 4\}$, that is, we first add the matching rules on nodes 2, 3 and 4 to identify the label of node 1 (i.e., use the new path), and then update node 1 to tag the packets and forward them to node 4. The new dependency graph is $\{1 \rightharpoonup 2, 1 \rightharpoonup 3, 1 \rightharpoonup 4, 5 \rightharpoonup 7\}$, as shown in Fig. 3(c). We can find that there are no circles in the new dependency graph, so the conflict is resolved.

3.5. Congestion-free update

To avoid transient congestion, before flow f^* is migrated to link (u, v), it needs to be satisfied that even after flow f^* is moved into link (u, v), the total load on the link will not exceed the link capacity $c(u, v)$. This means that if the rules of flow f^* are updated, the link (u, v) will not be congested. This can be described as

$$\sum_{f \in \Gamma(u,v)+\Psi(u,v)+\Phi(u,v)} y_f^{(u,v)} + y_{f^*}^{(u,v)} \le c(u, v), \tag{8}$$

where $y_f^{(u,v)}$ represents the size of flow f on the link (u, v), $\Gamma(u, v)$ represents the set of flows that need to be removed from the link (u, v), but have not yet been removed from the link (u, v), $\Psi(u, v)$ represents the set of unchanged flows on link (u, v), and $\Phi(u, v)$ represents the set of new flows that have been moved into the link (u, v).

After each flow passes through a waypoint (such as the BCH encoder or WAN optimizer), its traffic size may change. We use y_f^{u-} and y_f^{u+} respectively to represent the size of the flow f before and after being processed by node u. That is, $y_f^{u+} = \alpha_u \cdot y_f^{u-}$, where α_u represents the traffic change factor after being processed by node u, and it can be predicted [21,22]. If node u is not a waypoint, $\alpha_u = 1$. The size of the flow f over the link (u, v) can be represented as $y_f^{(u,v)} = y_f^{u+}$. Then the Eq. (8) can be transformed into the following Eq. (9),

$$\sum_{f \in \Gamma(u,v)+\Psi(u,v)+\Phi(u,v)} y_f^{u+} + y_{f^*}^{u+} \le c(u, v) \tag{9}$$

3.6. Problem analysis

Theorem 1. *The proposed consistent update (CSUD) problem is NP-hard.*

Proof of Theorem 1. We consider a simplified version of the CSUD problem, that is, we only need to find a congestion-free update sequence. Given a network, its link capacity is c. There is a flow set $F = \{f_1, f_2, f_3, \ldots, f_m\}$, where the old paths of flow f_1 and rest flows $\{f_2, f_3, \ldots, f_m\}$ are $A \to B \to E$ and $A \to D \to F$. The new paths of flow f_1 and flows $\{f_2, f_3, \ldots, f_m\}$ are $A \to D \to E$ and $A \to B \to F$, respectively. We use $y_{f_i}^{u+} \in N$ to represent the size of flow f_i on the link (u, v), that is, for flow f_1 and flow set $H = \{f_2, f_3, \ldots, f_m\}$, there are $y_{f_1}^{A+} = c/2$ and $\sum_{i=2}^{i=m} y_{f_i}^{A+} = c$. In order to allow flow f_1 to migrate from $A \to B \to E$ to $A \to D \to E$ and meet the link capacity constraint, we need to select a flow subset H' from the flow set H. Let flows in the flow subset H' migrate from $A \to D \to F$ to $A \to B \to F$, and the total flow size in H' is $c/2$, that is, $\sum_{f \in H'} y_f^{A+} = c/2$. In order to find the flow subset H', we need to divide the flow set H into two sets whose total flow size is both $c/2$. This is equivalent to the partition problem [30], which requires that the set S of n integers be divided into two subsets S_1 and S_2 with the same sum. So the CSUD problem is an NP-hard problem.

4. Rule update algorithm

In this section, we design a rule consistency update algorithm to get the rule update sequence that can simultaneously avoid blackholes, loops, congestion, and policy violations.

4.1. Rule consistency update

In order to ensure consistent updates, we design a Rule Consistency Update (RCU) algorithm to get the rule update sequence, as shown in Algorithm 5. The RCU algorithm mainly consists of five steps. In the first step, we invoke Algorithm 1 to obtain the segment nodes G_f, segment set S_f and blackhole-free dependency graph BH. In the second step, we invoke Algorithm 2 to construct loop-free dependency graphs L. In the third step, we invoke Algorithm 3 to construct the waypoint-enforcement dependency graphs WP. In the fourth step, we resolve the conflicts in the dependency graph by invoking Algorithm 4, and obtain a new conflict-free dependency graph D_f and a new segment set S_f (lines 11–13). In the fifth step, Algorithm 6 is invoked to obtain an update sequence that can simultaneously avoid blackholes, loops, policy violations, and congestion.

[4] The matching node refers to the switches on the routing path that are configured with matching rules.

Algorithm 5 RCU: Rule Consistency Update

Input: Flow set F, old path P_f^o, new path P_f^n, waypoint set W
Output: Update sequence US

Let $BH = L = WP = D = US = \emptyset$;
for each $f \in F$ **do**
  Step 1: Construct blackhole-free dependency graph
  Run Algorithm 1 and obtain BH, S_f, G_f;
  Step 2: Construct loop-free dependency graph
  Run Algorithm 2 and obtain L;
  Step 3: Construct the waypoint-enforcement dependency graph
  Run Algorithm 3 and obtain WP;
  $D_f = \{BH + L + WP\}$;
  Step 4: Construct a new conflict-free dependency graph and obtain a new segment set
  Run Algorithm 4 and obtain conflict-free dependency graph D_f;
  Delete the segments in the I_f from S_f;
  Re-divide the new path according to I_f to get a new segment set S_f;
  $D = D \cup D_f$, $S = S \cup S_f$;
end for
Step 5: Get the update sequence to ensure consistent update
Run Algorithm 6 and obtain US;

4.2. Getting update sequence

Algorithm 6 Get Update Sequence

Input: Flow set F, old path P_f^o, new path P_f^n, segment set S, dependency graph D
Output: Update sequence US

Put the nodes without dependencies into us_0;
Delete us_0 from D, $US = US + us_0$;
Sort flow size in ascending order;
while $S \notin \emptyset$ **do**
  $j = j + 1$, $us_j = \emptyset$;
  for each $f \in F$ **do**
    for each $s \in S_f$ **do**
      if for each link $e \in s$, the Eq. (9) can be satisfied **then**
        if $s[0]$ has no precursor node in D_f **then**
          $us_j = us_j + s[0]$;
          Delete s from S_f and S;
        end if
      end if
    end for
  end for
  $US = US + us_j$;
  if $us_j = \emptyset$ **then**
    for each $s \in S_f$ **do**
      if link $e \in s$ does not satisfy the Eq. (9) **then**
        $y_f = \min_{e' \in P_f^n} \gamma(e')$, $j = j - 1$;
      end if
    end for
  end if
end while

Algorithm 6 is designed to transform the obtained rule update dependency graph into a rule update sequence while avoiding congestion. The main processes of the algorithm are as follows. We first update the nodes that have no dependencies (including matching nodes), and delete us_0 from the dependency graph D (lines 1–2). For nodes with dependencies, in order to ensure consistent updates, we need to take the following measures (lines 3–24). For each link $e \in s$ in the segment $s \in S_f$ of each flow $f \in F$, if the link load satisfies the Eq. (9) after the flow f moves into the link, and the segment node $s[0]$ has no dependency with other nodes, it means that the segment node can be updated, or mark the packets if the segment node is a tag node. We put $s[0]$ into the update sequence us_i and delete s from s_f (lines 6–15). If there are deadlocks [10] or there are conflicts between congestion-free and the three consistency attributes (i.e., blackhole-free, loop-free, and policy consistency), we use the rate limiting method [8,10]. That is, the rate of flow f is limited to the minimum available link bandwidth $\gamma(e')$ (lines 17–23). The above processes are repeated until all nodes are put into the update sequence.

4.3. Algorithm complexity analysis

In the network, there are n nodes, m flows, and w waypoints. We analyze the complexity of RCU algorithm as follows. As shown in Algorithm 5, the RCU algorithm consists of five stages. In the first stage, Algorithm 1 is invoked to divide the new path and construct blackhole-free dependency graphs. Since each path contains at most n nodes, it costs $O(n)$ to get the segment nodes, and $O(n^2)$ to construct blackhole-free dependency graphs. Thus, the total cost of the first stage is $O(n^2)$. In the second stage, Algorithm 2 is invoked, and it costs $O(n^2)$ to find the loops formed by the new and old paths, and $O(n^2)$ to construct loop-free dependency graphs. Therefore, the total cost of the second stage is $O(n^2)$. In the third stage, Algorithm 3 is invoked, and it costs $O(wn^2)$ to construct waypoint-enforcement dependency graphs. In the fourth stage, Algorithm 4 is invoked, and it costs $O(wn^2)$ to get the circles in the dependency graph. Since at least one circle is resolved in each round of the fourth stage, it needs to perform at most n rounds. In general, the number of circles is small, and it can be considered as a constant. Thus, the fourth stage costs $O(wn^2)$ to construct conflict-free dependency graphs. In Algorithm 5, the above four algorithms are invoked, and they cost $O(mwn^2)$ to construct blackhole-free, loop-free, waypoint-enforcement, and conflict-free dependency graphs for m flows.

In the fifth stage, Algorithm 6 is invoked. It first costs $O(mwn)$ to update nodes that have no dependencies, and then costs $O(m^2)$ to sort the flows. Next, for nodes with dependencies, it takes $O(nm^2 + wmn^2)$ to find nodes that can be updated. If there are deadlocks or conflicts between congestion-free and the above three consistency attributes, it takes $O(mn + n^2)$ to calculate and update the transmission bandwidth after the rate limit. Since at least one segment is updated or one conflict or deadlock is resolved in each round of the fifth stage, it needs to perform at most $K = mn$ rounds. The total cost of the fifth stage is $O(Knm^2 + Kwmn^2)$. To speed up the algorithm, we can set a constant threshold for the number of iterations K [10]. Thus, the total cost of the fifth stage is $O(nm^2+wmn^2)$. Based on the above analysis, the complexity of RCU algorithm is $O(nm^2 + wmn^2)$.

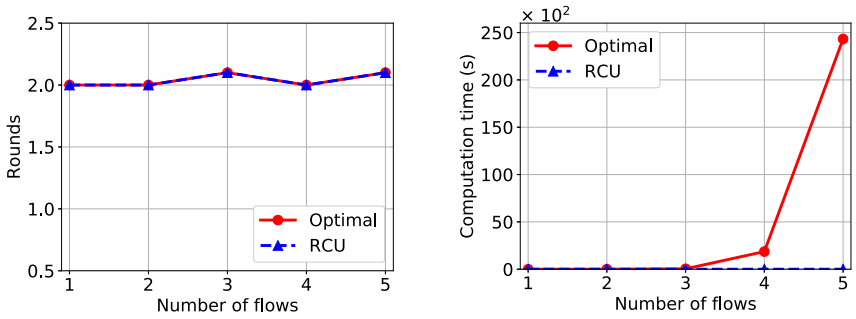
5. Performance evaluation

In this section, we first introduce the simulation settings and then discuss the simulation results in detail.

5.1. Simulation settings

Operating environment: We evaluate our algorithm in experimental and simulation settings. The experimental environment is based on our Ryu [31]/Mininet [32] running RYU NOS, OpenFlow v1.3 and Open-VSwitch v2.3. Besides, numerical simulations are implemented on a Python-based simulator. They are executed on a machine with 8G RAM, 3.2 GHz CPU and i7-8700 processor.

Topology: Our experiment is implemented in three topologies of Fat-tree, VL2 and Mesh. The 8-pod Fat-tree topology [9] has 16 core switches, 32 aggregation switches, 32 edge switches and 128 hosts



(a) Update rounds (b) Computation time

Fig. 4. Performance comparison between RCU algorithm and optimal solution.

respectively. The VL2 topology [33] has 5 intermediate switches, 20 aggregation switches, 50 ToR switches and 100 hosts. The Mesh topology [1,10] has 80 switches, and the degree of each node in the network is 4.

Parameters: The capacity of each link is set to 1 Gbps [1,10]. The number of waypoints varies from 1 to 3 [17], and the traffic factor of each waypoint is randomly selected from 0.8 to 1.2 [34].

Workloads: We use a public data set to generate traffic load [35,36]. The data set is collected from a network of 23 nodes, collected every 15 min for several months. And the data set contains the source node, destination node, and bandwidth of each flow. The number of flows varies from 100 to 300 to simulate various network loads. However, the three topologies we selected have more hosts than the data set. Therefore, we randomly generate the source and the destination nodes of a flow, and then select the bandwidth in the data set as the flow size. Similar to [4,5], we generate old and new paths based on the Dijkstra algorithm [28] for sequentially traversing waypoints.

Metrics: We first evaluate our algorithm with numerical simulations by comparing it with existing algorithms in terms of success rate, maximum link utilization, computation time and update rounds. This is because the above metrics are macroscopic behaviors and do not involve micro operations, such as message processing and flow rule updates. Further, we evaluate the total execution time of our algorithm in experimental environments based on Mininet and Ryu controller, as it involves microscopic behaviors.

Methods: We will compare our RCU algorithm with six closely related algorithms, including DRF [12], DSF [12], FLIP [19], CSU [15], NRCU and optimal solution. Among them, DRF and DSF algorithms are designed to avoid blackholes, loops, and policy violations. The DSF algorithm ensures that there are no loops at any time. The DRF algorithm weakens the loop-free condition, and only ensures that there are no loops in the current path from the source node. The FLIP algorithm is designed to avoid blackholes, loops, policy violations, and conflicts between loop-free and policy consistency. This method allows packets to traverse a finite number of loops while resolving conflicts. The CSU algorithm is designed to avoid blackholes, loops, and congestion. The NRCU algorithm is a special case of the RCU algorithm, and it does not consider traffic changes. The optimal solution is obtained by exploring the solution space using the branch and bound method [37]. We compare the above algorithms with RCU algorithm to emphasize the importance of jointly considering blackholes, loops, congestion, and policy violations. The detailed differences and correlations between our algorithm and these algorithms are discussed in Section 2, as shown in Table 1.

5.2. Comparison between RCU algorithm and optimal solution

In this subsection, we compare our RCU algorithm with the optimal solution in terms of update rounds and algorithm computation time, in Fat-tree topology. The experimental results are the average values of 10 tests, as shown in Fig. 4.

Table 3
Additional TCAM overhead (RCU vs FLIP).

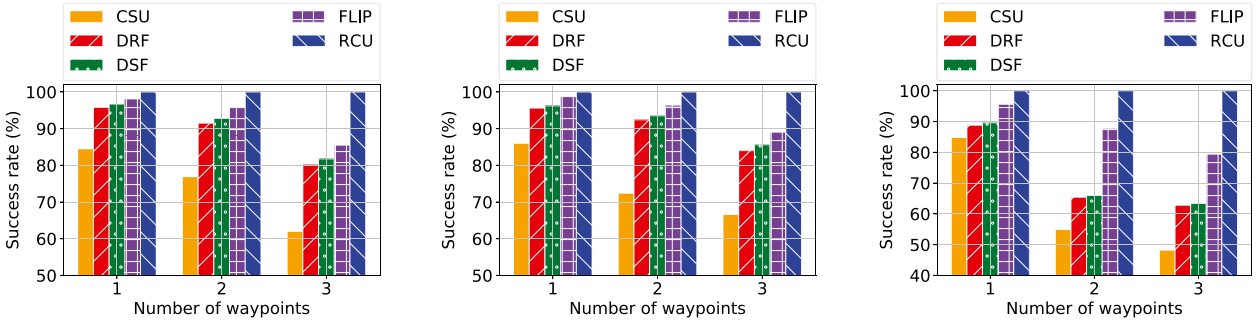
Topology \ Number of waypoints	$w = 1$	$w = 2$	$w = 3$
Fat-tree	0.9%	2.4%	4.1%
VL2	0.8%	2.7%	3.3%
Mesh	2.1%	3.5%	5.4%

As shown in Fig. 4(a), our algorithm can always hit the optimal solution under different numbers of flows. This result verifies that our algorithm has good performance. As shown in Fig. 4(b), the computation time of our algorithm is significantly shorter than that of obtaining the optimal solution, and the computation time of the optimal solution grows exponentially. This is because the solution space grows exponentially as the number of flows increases. Searching for an optimal solution in such a large solution space is very time-consuming. For example, when updating the paths of 5 flows, the computation time of our algorithm is 0.01s, while the computation time of the optimal solution is 24326.54 s. This is because the optimal solution is obtained by brutally exploring the whole solution space with an exponential scale update sequences. It should be noted that when the number of flows is large, our algorithm may not be able to hit the optimal solution, however obtaining the optimal solution in this case is very time-consuming.

5.3. Comparison of RCU algorithm with FLIP, DRF, DSF and CSU algorithms

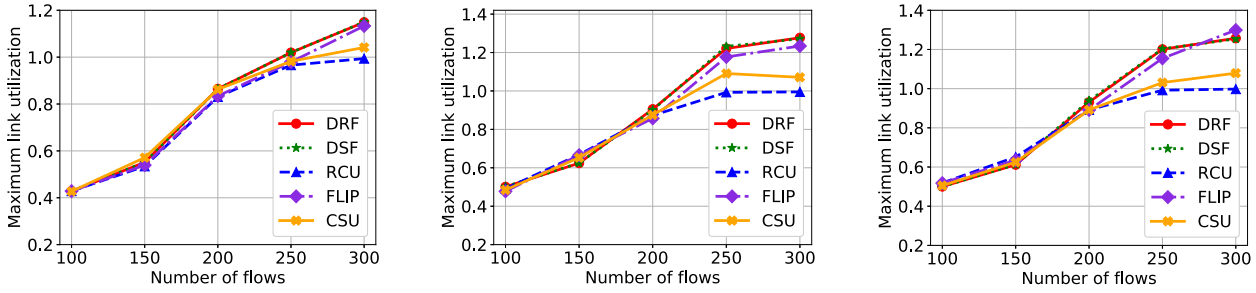
In this subsection, we evaluate the success rate, TCAM overhead, maximum link utilization, computation time, and update rounds of RCU, FLIP, DRF, DSF and CSU algorithms. The experimental results are the average values of 50 tests.

(1) **Success rate**: Fig. 5 shows the success rate of the different algorithms when the number of flows is 100. Success rate refers to the ratio that the obtained update sequence can ensure loop-free and policy consistency. As shown in Fig. 5(a), our algorithm always outperforms the others and is close to 1. While the success rate of other algorithms decreases as the number of waypoints increases. As shown in Fig. 5(c), the success rate of CSU, DRF, DSF and FLIP algorithms respectively is 48.2%, 62.8%, 63.4% and 79.4% when the number of waypoints is 3 (i.e., $w = 3$). This is because our algorithm can effectively guarantee loop-free and policy consistency, and it also avoids conflicts between them by constructing the conflict avoidance dependency graph. However, the CSU algorithm ignores policy consistency, and other algorithms do not resolve (DRF and DSF) or fully resolve (FLIP) the conflicts between loop-free and policy consistency. As the number of waypoints increases, the probability of conflicts between



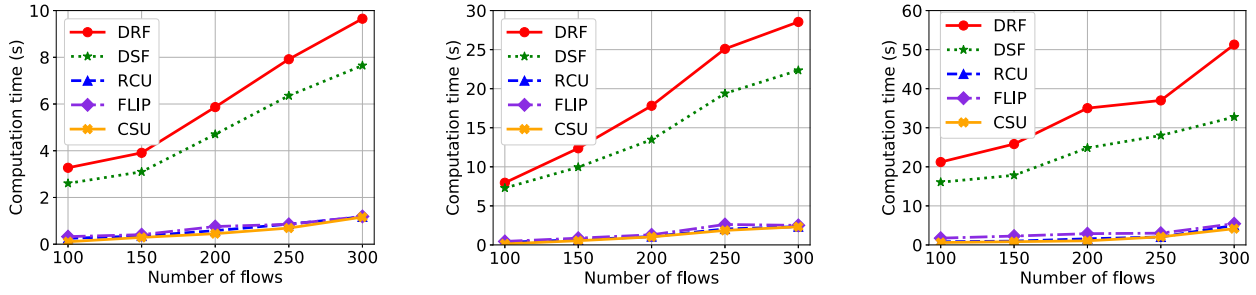
(a) Fat-tree topology (b) VL2 topology (c) Mesh topology

Fig. 5. Success rate in different topologies.



(a) Fat-tree topology (b) VL2 topology (c) Mesh topology

Fig. 6. Maximum link utilization in different topologies.



(a) w=1 (b) w=2 (c) w=3

Fig. 7. Calculate update sequence time in Fat-tree topology.

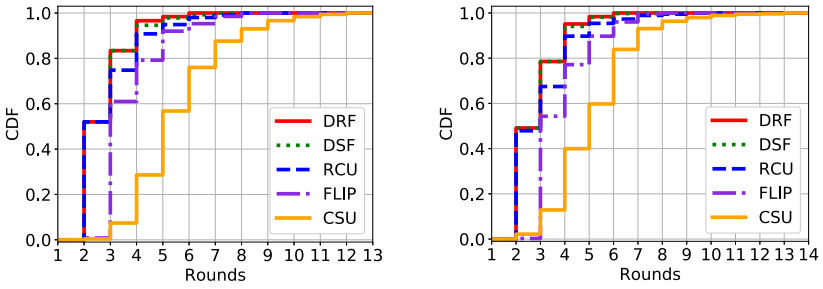
loop-free and policy consistency may also increase, so the success probability of CSU, DRF, DSF and FLIP algorithms may decrease.

In addition, we observe that the success rate of CSU, DRF, DSF and FLIP algorithms in Mesh is lower than that in Fat-tree and VL2, because there are fewer alternate links in Mesh than in Fat-tree and VL2 topologies. Hence, the probability of conflicts between loop-free and policy consistency increases, and the success rate of CSU, DRF, DSF and FLIP algorithms decreases. Finally, we show the extra TCAM used by our algorithm compared to FLIP, as shown in Table 3. The results show that our algorithm can effectively resolve the conflict between policy consistency and loop-free with a little extra TCAM resources.

(2) **Congestion**: Fig. 6 shows the change of the maximum link utilization with the increase of the number of flows, when the number of waypoints is 2. Once the maximum link utilization exceeds 1, congestion will occur, resulting in packet loss and throughput degradation. We can see from Fig. 6 that the RCU algorithm always guarantees that the maximum link utilization is less than or equal to 1, and the maximum link utilization of the DRF, DSF, FLIP and CSU algorithms is sometimes greater than 1 (that is, congestion occurs). Specifically, in Mesh with 300 flows, the maximum link utilization of the DRF, DSF, RCU, FLIP and CSU algorithms are 1.257, 1.254, 0.997, 1.299 and 1.079, respectively. This is because the DRF, DSF, and FLIP algorithms do not consider avoiding transient congestion. Although transient congestion is considered in the CSU algorithm, it ignores that the size of a flow may change after being processed by waypoints, which may still cause congestion. In our RCU algorithm, all these factors are considered.

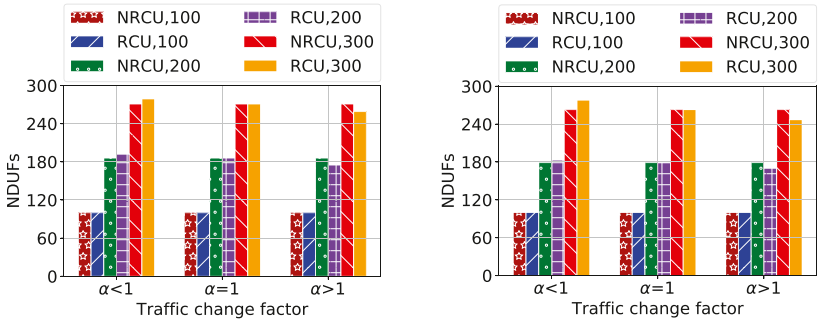
(3) **Computation time**: Fig. 7 shows the time required for all algorithms to calculate the update sequence in the Fat-Tree topology. Obviously, the computation time of RCU, FLIP and CSU algorithms is short, which is significantly outperforms DRF and DSF algorithms. As shown in Fig. 7(c), when the number of flows is 300, the time spent by CSU, RCU, FLIP, DRF and DSF algorithms to calculate the update sequence is 4.16s, 4.99s, 5.43s, 51.28s and 32.75s, respectively. This is because the DRF and DSF algorithms need to solve linear programs with many constraints, which are time-consuming. While the RCU, FLIP and CSU algorithms are efficient heuristics, they can get results quickly.

(4) **Update rounds**: Fig. 8 shows the number of rounds required to update the different number of flows when $w = 3$ in the Fat-tree topology. In Fig. 8(b), when the number of flows is 300 (i.e., $|F| = 300$), the RCU algorithm requires slightly more update rounds than



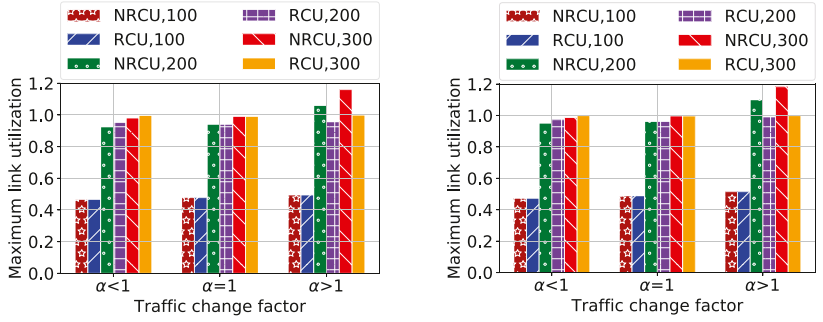
(a) $|F|$=100 (b) $|F|$=300

Fig. 8. Update rounds in different number of flows.



(a) Fat-tree topology (b) VL2 topology

Fig. 9. Number of directly-updated flows (NDUFs) in different topologies.



(a) Fat-tree topology (b) VL2 topology

Fig. 10. Maximum link utilization in different topologies.

the DRF, DSF and FLIP algorithms, and less update rounds than CSU algorithm. This is because RCU algorithm also needs to avoid transient congestion compared with DRF, DSF and FLIP algorithms. In order to avoid congestion during the update, the RCU algorithm needs to delay the update of some flows on the potentially congested link, so the number of update rounds increases. Furthermore, the CSU algorithm adopts the reverse order update method to avoid blackholes and loops, which requires more update rounds. Another observation result is that the number of update rounds required for the DRF algorithm is the least. The reason is that compared with other algorithms, the DRF algorithm weakens the loop-free condition, so the number of update rounds will be reduced.

5.4. Impact of traffic change factor on link utilization and update flow number

In this subsection, we evaluate the impact of the traffic change factor α on the number of directly-updated flows (that is, no dependency on other flows) and the maximum link utilization in the RCU and NRCU algorithms.

As shown in Fig. 9, when $\alpha < 1$, compared with the RCU algorithm, the NRCU algorithm that does not consider the flow sizes change (i.e., $\alpha = 1$) will get less directly-updated flows. This means that unnecessary waiting time for some flows is increased. The reason is that the flow sizes have become smaller after being processed by waypoints when $\alpha < 1$. The link is a potentially congested link when $\alpha = 1$, but the link may not be a potentially congested link when $\alpha < 1$, thus the flows on this link can be updated directly. When $\alpha = 1$, NRCU and RCU algorithms get the same result because the flow sizes have not changed.

When $\alpha > 1$, compared with the RCU algorithm, although the NRCU algorithm can obtain more flows that can be directly updated, as the number of flows increases, the maximum link utilization of the NRCU algorithm is over 1. As shown in Fig. 10(b), when the number of flows is 300, the maximum link utilization of the NRCU algorithm is 1.183, which will cause link congestion. This is because when $\alpha > 1$, the flow size becomes larger after being processed by waypoints. The link is not a potentially congested link when $\alpha = 1$, but the link may be a potentially congested link when $\alpha > 1$. Because NRCU algorithm does not consider the change of flow sizes, it will directly update the

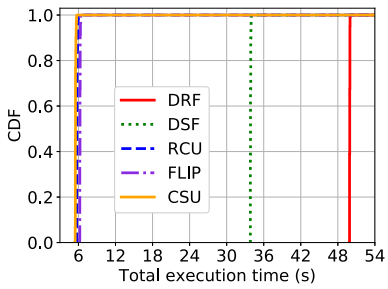


Fig. 11. Total execution time in Fat-tree topology.

flows on this potentially congested link, resulting in link congestion. However, the RCU algorithm will delay the update of some flows on this potentially congested link to avoid congestion, so the number of directly-updated flows is reduced, and the maximum link utilization is always less than or equal to 1.

In addition, we can observe from Fig. 9 that when the number of flows is 100, the number of directly-updated flows obtained by the RCU and NRCU algorithms basically does not change with the traffic change factor. This is because when the number of flows is 100, the network load is relatively light, and there are many alternative links in networks, so the probability of congestion is relatively small.

5.5. Experimental result

To further verify the practicality of our algorithm, we evaluate our algorithm in an experimental environment in terms of total execution time, which includes rule update time and algorithm computation time. We test the total execution time required to update 300 flows in the Fat-tree topology, and the results are shown in Fig. 11. Obviously the total execution time of our algorithm is short, it is close to CSU algorithm and FLIP algorithm, their updates are completed in about 6s. At the same time, the total execution time of DSF algorithm and DRF algorithm is about 34 s and 50 s respectively. This is because our algorithm has a shorter computation time, while the DSF algorithm and the DRF algorithm take a long time to run the mathematical optimizer to obtain the result. Also, the rule update time of the switch (tens of milliseconds per round) is significantly shorter than the computation time. The above results show that our algorithm has a short execution time, which helps the rules on switches to be updated quickly.

6. Conclusions

In this paper, we propose a rule consistency update mechanism that can simultaneously avoid blackholes, loops, congestion, and policy violations. First, we analyze the causes of blackholes, loops, congestion and policy violations during the update. Then, we construct update dependency graphs to avoid blackholes, loops, and waypoints being bypassed. In order to improve the success rate of consistent updates, we build a conflict avoidance model to resolve conflicts between dependency graphs. In addition, to avoid transient congestion and consider that the waypoint may change the size of its process flow, we propose a congestion avoidance model based on the traffic change factor. Subsequently, we prove that the consistent update problem is an NP-hard problem, and propose an effective heuristic RCU algorithm to obtain the update sequence that satisfies the above four consistency. Finally, we test the different performance indicators of our proposed algorithm in different situations, and find that compared with DRF, DSF, FLIP and CSU algorithms, our proposed algorithm can effectively improve the success rate and reduce the maximum link utilization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62272069, 62202073), China Postdoctoral Science Foundation (2021M700587), Chongqing Natural Science Foundation for Postdoctoral (cstc2021jcyj-bshX0077) and Fundamental Research Funds for the Central Universities (2022CDJXY-020).

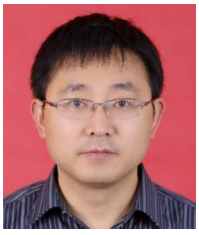
References

[1] W. Wang, W. He, J. Su, Y. Chen, Cupid: Congestion-free consistent data plane update in software defined networks, in: IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications, 2016, pp. 1–9.
[2] Y. Yao, S. Guo, P. Li, G. Liu, Y. Zeng, Forecasting assisted VNF scaling in NFV-enabled networks, Comput. Netw. 168 (2020) 107040.
[3] S.R. Zahedi, S. Jamali, P. Bayat, A power-efficient and performance-aware online virtual network function placement in SDN/NFV-enabled networks, Comput. Netw. 205 (2022) 108753.
[4] X. Fan, H. Xu, H. Huang, X. Yang, Real-time update of joint SFC and routing in software defined networks, IEEE/ACM Trans. Netw. 29 (6) (2021) 2664–2677.
[5] L. Liu, S. Guo, G. Liu, Y. Yang, Joint dynamical VNF placement and SFC routing in NFV-enabled SDNs, IEEE Trans. Netw. Serv. Manag. 18 (4) (2021) 4263–4276.
[6] V. Eramo, E. Miucci, M. Ammar, Study of reconfiguration cost and energy aware VNE policies in cycle-stationary traffic scenarios, IEEE J. Sel. Areas Commun. 34 (5) (2016) 1281–1297.
[7] V. Eramo, F.G. Lavacca, Optimizing the cloud resources, bandwidth and deployment costs in multi-providers network function virtualization environment, IEEE Access 7 (2019) 46898–46916.
[8] X. Jin, H.H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, R. Wattenhofer, Dynamic scheduling of network updates, Acm Sigcomm Comput. Commun. Rev. 44 (4) (2014) 539–550.
[9] X. He, J. Zheng, H. Dai, Y. Sun, W. Dou, G. Chen, Buffer-assisted network updates in timed SDN, IEEE Trans. Commun. 69 (10) (2021) 6822–6837.
[10] K. Wu, J. Liang, S. Lee, Y. Tseng, Efficient and consistent flow update for software defined networks, IEEE J. Sel. Areas Commun. 36 (3) (2018) 411–421.
[11] M. Tsai, C. Chuang, S. Chou, A. Pang, G. Chen, Enabling efficient and consistent network update in wireless data centers, IEEE Trans. Netw. Serv. Manag. 16 (2) (2019) 505–520.
[12] A. Ludwig, S. Dudycz, M. Rost, S. Schmid, Transiently policy-compliant network updates, IEEE/ACM Trans. Netw. 26 (6) (2018) 2569–2582.
[13] I. Maity, A. Mondal, S. Misra, C. Mandal, CURE: Consistent update with redundancy reduction in SDN, IEEE Trans. Commun. 66 (9) (2018) 3974–3981.
[14] G. Li, Y.R. Yang, F. Le, Y. Lim, J. Wang, Update algebra: Toward continuous, non-blocking composition of network updates in SDN, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 1081–1089.
[15] H. Song, S. Guo, P. Li, G. Liu, FCNR: Fast and consistent network reconfiguration with low latency for SDN, Comput. Netw. 193 (2) (2021) 108113.
[16] X. He, J. Zheng, H. Dai, C. Zhang, G. Li, W. Dou, W. Rafique, Q. Ni, G. Chen, Continuous network update with consistency guaranteed in software-defined networks, IEEE/ACM Trans. Netw. 30 (3) (2022) 1424–1438.
[17] A. Ludwig, S. Dudyzc, M. Rost, S. Schmid, Transiently secure network updates, in: The 2016 ACM SIGMETRICS International Conference, 2016, pp. 273–284.
[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for network update, Acm Sigcomm Comput. Commun. Rev. 42 (2012) 323–334.
[19] S. Vissicchio, L. Cittadini, FLIP the (flow) table: Fast lightweight policy-preserving SDN updates, in: IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications, 2016, pp. 1–9.
[20] S. Vissicchio, L. Cittadini, Safe, efficient, and robust SDN updates by combining rule replacements and additions, IEEE/ACM Trans. Netw. 25 (5) (2017) 3102–3115.
[21] Y. He, X. Zhang, Z. Xia, Y. Liu, K. Sood, S. Yu, Joint optimization of service chain graph design and mapping in NFV-enabled networks, Comput. Netw. 202 (2022) 108626.
[22] W. Ma, J. Beltran, D. Pan, N. Pissinou, Placing traffic-changing and partially-ordered NFV middleboxes via SDN, IEEE Trans. Netw. Serv. Manag. 16 (4) (2019) 1303–1317.

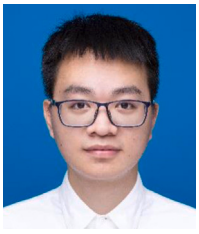
[23] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, S. Schmid, Efficient loop-free rerouting of multiple SDN flows, IEEE/ACM Trans. Netw. 26 (2) (2018) 948–961.
[24] H. Zhou, X. Gao, J. Zheng, G. Chen, Scheduling relaxed loop-free updates within tight lower bounds in SDNs, IEEE/ACM Trans. Netw. 28 (6) (2020) 2503–2516.
[25] P. Li, S. Guo, C. Pan, L. Yang, G. Liu, Y. Zeng, Fast congestion-free consistent flow forwarding rules update in software defined networking, Future Gener. Comput. Syst. 97 (8) (2019) 743–754.
[26] I. Maity, S. Misra, C. Mandal, DART: Data plane load reduction for traffic flow migration in SDN, IEEE Trans. Commun. 69 (3) (2021) 1765–1774.
[27] S. Bera, S. Misra, A. Jamalipour, FlowStat: Adaptive flow-rule placement for per-flow statistics in SDN, IEEE J. Sel. Areas Commun. 37 (3) (2019) 530–539.
[28] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1) (1959) 269–271.
[29] W. Wang, Y. Liu, J. Liu, Y. Li, H. Song, Y. Wang, J. Yuan, Consistent state updates for virtualized network function migration, IEEE Trans. Serv. Comput. 13 (6) (2020) 999–1006.
[30] S. Chopra, M.R. Rao, The partition problem, Math. Program. 59 (1–3) (1993) 87–115.
[31] Ryu, https://ryu.readthedocs.io/en/latest/.
[32] Mininet, http://mininet.org/.
[33] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, S. Sengupta, VL2: A scalable and flexible data center network, Commun. Acm 54 (3) (2009) 95–104.
[34] W. Ma, J. Beltran, Z. Pan, D. Pan, N. Pissinou, SDN-based traffic aware placement of NFV middleboxes, IEEE Trans. Netw. Serv. Manag. 14 (3) (2017) 528–542.
[35] S. Uhlig, B. Quoitin, J. Lepropre, S. Balon, Providing public intradomain traffic matrices to the research community, Acm Sigcomm Comput. Commun. Rev. 36 (1) (2006) 83–86.
[36] Traffic Matrices, https://totem.info.ucl.ac.be/dataset.html.
[37] Branch and Bound, https://en.wikipedia.org/wiki/Branch_and_bound.

Pan Li received the M.S. degree from Southwest University, Chongqing, China, in 2019. She is currently working toward the PhD's degree in computer science and technology, Southwest University. Her research interests include data center networks, network functions virtualization and software defined networking.

Guiyan Liu received the Ph.D. degree in Computational Intelligence and Information Processing from Southwest University of China in 2020. She is currently an assistant researcher with the Chongqing University of China. Her research interests include data center networks, software defined networking, network function virtualization and 5G.



Songtao Guo received the BS, MS, and Ph.D. degrees in computer software and theory from Chongqing University, Chongqing, China, in 1999, 2003, and 2008, respectively. He was a professor from 2011 to 2012 at Chongqing University and a professor from 2013 to 2018 at Southwest University. He is currently a full professor at Chongqing University, China. He was a senior research associate at the City University of Hong Kong from 2010 to 2011, and a visiting scholar at Stony Brook University, New York, from May 2011 to May 2012. His research interests include wireless sensor networks, wireless ad hoc networks, data center networks and mobile edge computing. He has published more than 100 scientific papers in leading refereed journals and conferences. He has received many research grants as a principal investigator from the National Science Foundation of China and Chongqing and the Postdoctoral Science Foundation of China.



Yue Zeng received the M.S. degree in the department of electronic information engineering from Southwest University, Chongqing, China, in 2019. He is currently working toward the Ph.D. degree in the department of computer science and technology in Nanjing University, China. His research interests include network functions virtualization, software defined networking, machine learning for networking, distributed computing, and edge computing.