Optimal Consistent Network Updates in Polynomial Time

Pavol Černý[1], Nate Foster[2], Nilesh Jagnik[1], and Jedidiah McClurg[1]

[1]University of Colorado Boulder
[2]Cornell University

Abstract. Software-defined networking (SDN) enables controlling the behavior of a network in software, by managing the the forwarding rules installed on switches. However, it can be difficult to ensure that certain properties are preserved during periods of reconfiguration. The widely-accepted notion of *per-packet consistency* requires every packet to be forwarded using the new configuration or the old configuration, but not a mixture of the two. A (partial) order on switches is a *consistent order update* if updating the switches in that order guarantees per-packet consistency. A consistent order update is *optimal* if it allows maximal parallelism, where switches may be updated in parallel if they are incomparable in the order. This paper presents a polynomial-time algorithm for computing optimal consistent order updates. This contrasts with other recent results, which show that for other properties (e.g., loop-freedom and waypoint enforcement), the optimal update problem is NP-complete.

1 Introduction

Software-defined networking (SDN) replaces conventional network management interfaces with higher-level APIs. SDN can be used to build a variety of applications, but it can be difficult for operators to *correctly* and *efficiently* reconfigure the network—i.e., update the global set of forwarding rules installed on switches (known as a *configuration*). Even if the initial and final configurations are correct, naïvely updating individual switches (known as *switch-updates*) can lead to incorrect transient behaviors such as forwarding loops, blackholes, bypassing a firewall, etc. Switch-updates can often be parallelized, but this too can cause incorrect behavior. Hence, we need a partial order on switch-updates which ensures that correctness properties hold before, during, and after the update.

Consistent order updates. This paper investigates the problem of computing a *consistent order update*. Given an initial and final network configuration, a consistent order update is a partial order on switch-updates, such that if the switches are updated according to this order, an important consistency property called *per-packet consistency* [16] is guaranteed throughout the update process. This property guarantees that each packet traversing the network will follow a single global configuration: either the initial one, or the final one, but not a mixture of the two. In particular, this means that if the initial and the final configurations are loop-free and blackhole-free, prevent bypassing a firewall, etc., then so do all of the intermediate configurations.

Optimal consistent order updates. In implementing a consistent order update, we would generally prefer to use one that is optimal. A consistent order update is *optimal* if it allows the most parallelism among all consistent order updates. Formally, recall that a consistent order update is a partial order on switch-updates—an optimal partial order is one where the length of the longest chain in the order is the smallest among all possible correct partial orders. Intuitively, this means the update can be performed in the smallest number of "rounds," where rounds are separated by waiting for in-flight packets to exit the network and by waiting for all the switch updates from the previous rounds to finish.

Single flow vs. multiple flows. A *flow* is a restriction of a network configuration to packets of a single type, corresponding to values in packet headers. A packet type might include the destination address, protocol number (TCP vs. UDP), etc. We show that if we consider flows to be *symbolic* (i.e., represented by predicates over packet headers, potentially matching multiple flows), then the problem is CO-NP-hard. In this paper, we focus on the problem of updating an *individual* flow—i.e., we are interested in the situation where the flows to be updated can be enumerated. Furthermore, as we are looking for efficient consistent order updates, we focus on the case where each switch can be updated at most once, from its initial to its final configuration.

Main result. Our main result is that for updating a single flow, there is a polynomial-time algorithm, with $O(n^2(n+m))$ complexity where n is the number of switches and m the number of links. The result is interesting both theoretically and practically. On the theoretical side, recent papers have presented complexity results for network updates. However, for many other consistency properties (loop-freedom, waypoint enforcement) and network models, the optimal network update problem is NP-hard [4, 6, 9, 10, 11, 12]. The same is true for results that study these problems with a model which is the same as ours (single flows, update every switch at most once). In contrast, we provide a *positive* result that there exists a polynomial-time algorithm for optimal order updates for a single flow, with respect to the per-packet consistency property. The consistency properties studied in these papers (loop-freedom and waypoint enforcement) are weaker than per-packet consistency, which offers a trade-off: enforcing only (for instance) loop-freedom allows more updates to be found, but it is an (exponentially) harder problem. In practice, network operators might wish to update

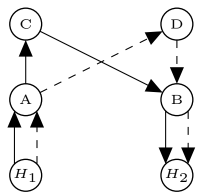

Fig. 1: Trivial update.

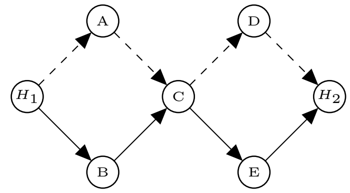

Fig. 2: Double diamond: no consistent update order exists.

only a small number of flows, and here our polynomial-time algorithm would be advantageous. A potential limitation is that if many flows are considered separately, it could lead to large forwarding tables.

Algorithm. Our algorithm models a network configuration as a directed graph with unlabeled edges, and an update from an initial configuration to a final configuration as a sequence of individual switch-updates—i.e., updating the outgoing edges at each switch. In order to determine whether a switch n can be updated while properly respecting the per-packet consistency property, we define a set of conditions on the paths *upstream* and *downstream* from n. We show that these conditions can be checked in $O(n(n+m))$ time. In this way, the algorithm produces a partial order on switches, representing the consistent order update (if such an order does not exist, our algorithm reports a failure). Additionally, we show that if the partial order is constructed greedily (i.e., all nodes that can be updated are immediately updated in parallel), it results in an *optimal* consistent order update. The challenging part of the proof is to show that this algorithm is complete (i.e., always finds a consistent order update if one exists) and optimal.

2 Overview

This section presents a number of simple examples to help develop further intuition about the consistent order updates problem and the challenges that any solution must address.

Consistent order updates. Consider Figure 1. In the initial configuration C_i (denoted by *solid* edges), the forwarding-table rules (outgoing edges) on each switch are set up such that host H_1 is sending packets to H_2 along the path $H_1 \rightarrow A \rightarrow C \rightarrow B \rightarrow H_2$. Let us assume that switch C is scheduled for maintenance, meaning we must first transition to configuration C_f (denoted by the *dashed* edges). Note that the two configurations differ only for nodes A and D. If the node A is updated before node D, packets from H_1 will be dropped at D. On the other hand, updating D before A leads to a consistent order update. Note that since we model networks as graphs, we will use the terms *switch* and *node* interchangeably based on the context, and similarly for the terms *edge* and *forwarding rule*. *Path* will be used to describe a sequence of adjacent edges.

In Figure 2, regardless of the order in which we update nodes, there will always be inconsistency. Note that here the nodes A and D can be updated first, but a problem arises due to nodes H_1 and C. Specifically, if C is updated before

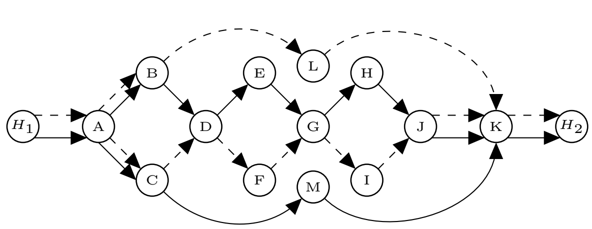

Fig. 3: Removable double diamond.

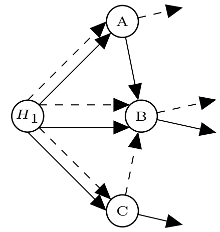

Fig. 4: Wait example.

H_1, then the network is in a configuration containing a path $H_1 \rightarrow B \rightarrow C \dashrightarrow D \dashrightarrow H_2$, which is not in either C_i or C_f. In other words, H_1 cannot be updated unless the (downstream) path from C to H_2 is first updated. On the other hand, C cannot be updated unless the (upstream) path from H_1 to C is first updated. We refer to this case as a *double diamond*. If we consider the notion of dependency graphs [13], where there is an edge from a node x to node y if the update of y can only be executed after the update of x, then our double diamond example corresponds to a cyclic dependency graph between H_1 and C.

Unfortunately, the presence of a double diamond (cyclic dependency) does not necessarily indicate that there cannot be a solution. Consider Figure 3, where there is a double diamond between D and J. Updating B removes the old traffic to D, and then after updating B, the nodes D, E, G, F, H, I, J have no incoming traffic. At this point, these nodes can be updated without violating per-packet consistency. Thus, the circular dependency has been eliminated, allowing a valid update order such as $[A, H_1, K, L, B, D, E, F, G, H, I, J, C, M]$. This shows that an approach (such as [7, 18]) based on a static dependency graph might miss some cases where a consistent order update exists—this is a limitation that is not exhibited by our algorithm.

Waits. As mentioned, it may be impossible to parallelize certain updates—we may need to make sure that some node x is updated before another node y. In other words, we may need to *wait* during the sequence of switch-updates to ensure that such updates are executed one after the other. This requirement can arise because when updating a node, we may need to ensure that (1) all of the previous switch-updates have been completed, and (2) all of the packets that were in the network since before the previous update have exited the network. The former type we call a *switch-wait*, and the latter a *packet-wait*.

In Figure 3, we see that L must be updated before updating B. To ensure that edges outgoing from L are ready, we must wait after sending the update command to L, in order to ensure that its forwarding rules have been fully installed. In other words, we say that there is a *switch-wait* required between updates of L and B. After updating B, the switch D becomes disconnected, but there may still be some packets in transit on the $B \rightarrow D$ path. Before updating D, we must ensure that packets along these old removed paths have been flushed from the network. For this reason, we say that a *packet-wait* is needed between updates of nodes D and B.

If we are interested only in finding a correct sequence of updates, we can wait (for an amount of time larger than the maximum switch-wait and packet-wait duration) after every node update. However, waits may not be necessary after every update if we update switches from separate parts of the network. For the Figure 3 example, the correct sequence with a *minimal* number of waits is $[A, H_1, K, L, ⓢ, B, ⓟ, D, E, F, G, H, I, J, ⓢ, C, M]$, where ⓟ denotes a packet-wait and ⓢ denotes a switch-wait. In this example, nodes A, H_1, K, L can be updated in parallel. Similarly, nodes D, E, F, G, H, I can be updated in parallel, etc. There are three waits, meaning this consistent order update requires *four* switch-update *rounds*.

The example in Figure 4 highlights the relationship between switch-waits and packet-waits. Observing that the configurations are roughly symmetrical, let us examine the relationship between nodes A, B, C. The correct order of updates between these nodes is $H_1, A, ⓟ, B, ⓢ, C$. This is because there must be a *switch-wait* between the updates of B and C, due to the presence of a C_f path $C \dashrightarrow B$. There must be a *packet-wait* between updates of switches A and B, due to the presence of a C_i path $A \rightarrow B$.

As is common in various other works (e.g., [9]), in this paper, we do not distinguish between packet-waits and switch-waits, and only use the term *wait*—our goal is to maximize the parallelism of switch-updates, i.e., minimize the number of switch-update rounds.

3 Network Model

Network and Configurations. A topology of a network is a graph $G = (N, E)$, where N is a set of nodes, and E is a set of directed edges. A configuration $C \in \mathcal{P}(E)$ is a subset of edges in E. A *proper* configuration is one that (a) has one source H_1, and (b) is acyclic. Here, a source is a designated node with no incoming edges, representing the point where packets enter the network. Note that cycles in a configuration are undesirable, as this would mean that traffic might loop forever in the network. We first consider the case with one source, and in Section 6, we describe a simple reduction for the case of multiple sources. Our goal is to transition from an initial configuration C_i to a final configuration C_f by updating individual nodes. We will consider C_i and C_f to be fixed throughout the paper, and assume that both are proper.

Updates. Let u be a node, and let C be a configuration. We define a function $out(C, u)$ which returns the set of edges from C whose source is u. The function $upd_1(C, u)$ returns the configuration C' such that $C' = (C \setminus out(C_i, u)) \cup out(C_f, u)$, that is, node u is updated to the final configuration in C'. Let R be the set of all sequences of nodes in N without repetition. We extend upd_1 to sequences of nodes by defining the function upd that, given a configuration C and a sequence of nodes S, returns a configuration $C' = upd(C, S)$. The function upd is defined by $upd(C, \varepsilon) = C$ (where ε is the empty sequence), and $upd(C, uS) = upd(upd_1(C, u), S)$. We consider sequences without repetition, because our goal is to find sequences that update every node at most once.

Paths. Given a configuration C, a C-path is a directed path (finite or infinite) whose edges are in C. For a path p, we write $p \in C$ if p is a C-path. A C_i-only path is one which is in C_i and not in C_f. Similarly, a C_f-only path is in C_f but not C_i. The function *nodes* takes a path q as an argument and returns a set Q of all nodes on a path. Let s and t be two nodes, and let C be a configuration. The function $\mathit{paths}(s,t,C)$ returns the set of all paths between s and t in configuration C. A path p in a configuration C is *maximal* if it is either (a) finite, and its last node has no outgoing edges in C, or (b) infinite. The function $\mathit{maxpaths}(s,C)$ returns the set of all maximal paths starting at node s in configuration C.

Path and Configuration Consistency. We say that a path p is *consistent* if and only if $p \in \mathit{maxpaths}(H_1, C_i) \vee p \in \mathit{maxpaths}(H_1, C_f)$, and a configuration C is *consistent* if and only if $\forall p \in \mathit{maxpaths}(H_1, C)$ we have that p is consistent. Intuitively, all maximal paths starting at H_1 are maximal paths in either the old configuration or the new configuration—this corresponds to per-packet consistency [16]. If C_i and C_f are proper, then so is every consistent configuration.

Waits. Let $U = u_1 u_2 \cdots u_k$ be a sequence of node updates. Let $C_j = \mathit{upd}(C_i, U_j)$ be the configuration reached after updating a sequence $U = u_1 u_2 \cdots u_j$ for $1 \le j \le k$, and let $C_0 = C_i$. For l, u such that $0 \le l \le u \le k$, let C_l^u be the configuration obtained as a union of configurations $C_l \cup \cdots \cup C_u$. We say that a *wait is needed* between u_j and u_k in U if and only if the configuration C_{j-1}^k is not consistent. To illustrate, let us return to the example in Figure 4 (note that we no longer distinguish between packet-waits and switch-waits). As mentioned, after updating H_1 and A, we need a wait before updating B. Let the configuration C_v be the union of all the intermediate configurations until after the update to B. Then C_v has the path $H_1 \to A \to B \to$, where we take the solid edge from A to B and a dashed outgoing edge from B, meaning a wait is needed. In this case, using the union of the configurations captures the reason for the wait.

Consistent update sequence. For any set of nodes S, let $\pi(S)$ be the set of sequences that can be formed by nodes in S, without repetition. Let $Z = S_1 S_2 \cdots S_k$ be a sequence such that each S_i is a subset of N. Let $\pi(Z)$ be the set of sequences defined by $\{r_1 r_2 \cdots r_k \mid r_1 \in \pi(S_1) \wedge r_2 \in \pi(S_2) \wedge \cdots \wedge r_k \in \pi(S_k)\}$.

The sequence $Z = S_1 S_2 \cdots S_k$ is a *consistent update sequence* if and only if

1. The sets $S_1, S_2, \cdots, S_k$ partition the set of nodes N. This ensures that $\forall U \in \pi(Z)$, we have $\mathit{upd}(C_i, U) = C_f$, i.e., after updating u, we are in C_f.
2. $\forall U \in \pi(Z)$, for every prefix U' of U, C=$\mathit{upd}(C_i, U')$ is a consistent configuration.
3. $\forall U \in \pi(Z)$, let $U' = u_1 u_2 \cdots u_j$ and $U'' = u_1 u_2 \cdots u_k$ be prefixes of u, s.t. $k > j$, then if a wait is needed between u_j, u_k in U, then u_j, u_k are in different sets S and S'.

Consistent Order Update Problem. Given an initial configuration C_i and the final configuration C_f, the *consistent order update problem* is to find a consistent update sequence if there exists one.

	Upstream (Condition for $paths(H_1, s, C_c)$)	Downstream (Condition for $maxpaths(s, C_c)$)
A	$Y_a(s) = \nexists\, p \in paths(H_1, s, C_c)$	$Z_a^{\dagger}(s) = (out(s, C_f) = \varnothing) \vee \forall p \in maxpaths(s, upd(C_c, s)) : p \in maxpaths(s, C_f)$
B	$Y_b(s) = \neg Y_a(s) \wedge \forall p \in paths(H_1, s, C_c) : p \in paths(H_1, s, C_i) \wedge p \in paths(H_1, s, C_f)$	$Z_b(s) = \forall p \in maxpaths(s, upd(C_c, s)) : p \in maxpaths(s, C_i) \vee p \in maxpaths(s, C_f)$
C	$Y_c(s) = \neg Y_a(s) \wedge \neg Y_b(s) \wedge \forall p \in paths(H_1, s, C_c) : p \in paths(H_1, s, C_f)$	$Z_c(s) = \forall p \in maxpaths(s, upd(C_c, s)) : p \in maxpaths(s, C_f)$
D	$Y_d(s) = \neg Y_a(s) \wedge \neg Y_b(s) \wedge \forall p \in paths(H_1, s, C_c) : p \in paths(H_1, s, C_i)$	$Z_d(s) = \forall p \in maxpaths(s, upd(C_c, s)) : p \in maxpaths(s, C_i)$
E	$Y_e(s) = \neg Y_a(s) \wedge \neg Y_b(s) \wedge \neg Y_c(s) \wedge \neg Y_d(s)$	$Z_e(s) = \forall p \in maxpaths(s, upd(C_c, s)) : p \in maxpaths(s, C_i) \wedge p \in maxpaths(s, C_f)$

Fig. 5: Necessary conditions for updating a node s in current configuration C_c

Optimal Consistent Order Update Problem. Given C_i and C_f, if a consistent update sequence exists, the *optimal consistent update problem* is to find a consistent update sequence of minimal length.

4 OrderUpdate Algorithm

This section presents an algorithm (Algorithm 1) that solves the consistent order update problem. It works by repeatedly finding and updating a node that can be updated without violating consistency. For clarity, we focus first on correctness. Section 5 presents an improved version that finds an optimal update.

Correct Sequence. A *correct* sequence of node updates $T = t_1 t_2 \cdots t_{|N|}$ refers to a consistent update sequence of singleton sets $Z = S_1 S_2 \cdots S_{|N|}$ s.t. $\forall j \in [1, |N|] : S_j = \{t_j\}$. Algorithm 1 uses a subroutine at Line 6 (in this section, the subroutine is Algorithm 2—in Section 5 we will replace it with Algorithm 3 to achieve optimality) to find a correct update sequence. It takes C_i and C_f as inputs and returns two sequences of nodes, R and R_w. Sequence R is the solution to the consistent order update problem (a sequence of singleton sets). Sequence R_w contains information about the placement of waits, which will be the same as R in this section, since we initially wait after every node update.

4.1 Necessary Conditions for Updating a Node

To determine which node updates lead to consistent configurations, we assume the network is in a consistent configuration C_c, and identify a set of necessary conditions that must hold for the update to preserve consistency. We classify nodes into five categories based on the types of paths that are incoming to them from H_1. The classification is given in the left-hand side of Figure 5.

Upstream Paths and Candidate Nodes. Paths from source H_1 to a node s are called *upstream* paths to s (in some configuration). The condition on these paths is called the upstream condition. If a node satisfies the upstream condition for one of the five categories/types, it is known as a *candidate* of that type.

Downstream Paths and Valid Nodes. Downstream paths from a node s are maximal paths starting at s (in some configuration). For each of the upstream conditions, there is a downstream condition which must be satisfied, in order to ensure that all maximal paths starting from H_1 in $upd(C_c, s)$ through s are consistent. If a candidate node satisfies the corresponding downstream condition, it is called *valid*. A node which is not valid is called *invalid*. Note that upstream paths to s are the same in C_c and $upd(C_c, s)$.

Lemma 1. In a consistent configuration C_c, if a valid node s is updated, then $upd(C_c, s)$ is consistent.

Proof Sketch. Figure 5 identifies nodes as Types A-E based on upstream conditions. The upstream conditions are exhaustive and mutually exclusive, meaning each node is a candidate of exactly one of the types. For each type described in Figure 5, our downstream condition ensures that updating preserves consistency. Upstream paths to a node may be fully contained in C_i or C_f (Type C and Type D respectively). For these cases, we need to ensure that downstream paths are also contained in C_i and C_f respectively. They may be in $C_i \cap C_f$ or $C_i \cup C_f$ (Type B and Type E respectively). For these cases, we need to ensure that downstream paths are in $C_i \cup C_f$ (for Type B) and $C_i \cap C_f$ (for Type E). Type A is a special case, as nodes of this type (also referred to as *disconnected nodes*) do not have any upstream paths. These nodes can be updated without the requirement of a downstream condition. However, we enforce a downstream condition (denoted $Z_a^\dagger$ in the table) in order to streamline the proofs. ☐

The proof of this and other theorems/lemmas are in the extended version [3]. Using Lemma 1, each node updated by OrderUpdate leads to a valid intermediate configuration. So, we change from C_i to C_f without going through an inconsistent state, and since we wait between all updates, we obtain a consistent sequence.

Theorem 1. Any sequence R of nodes produced by Algorithm 1 (using subroutine Algorithm 2) is correct.

4.2 Careful Sequences

Previously, we said that Type A candidates (disconnected nodes) do not require a downstream condition to be updated. However, Algorithm 1 imposes a downstream condition on disconnected nodes for them to be valid and updated. We refer to sequences that respect this downstream condition (i.e., update only valid nodes) as *careful* sequences. Let s be a node and C be a configuration, and define $valid_1(C, s)$ to be *true* if and only if s in valid in configuration C. We extend $valid_1$ to a sequence of nodes by defining *valid* as $valid(\varepsilon, C) = true$ (where ε is the empty sequence) and $valid(C, uS) = valid(upd(C, u), S) \wedge valid_1(C, u)$.

Careful Sequence A *careful* sequence $T = t_1 t_2 \cdots t_{|N|}$ is a correct sequence of nodes s.t. $\forall l \in [1, |N|] : valid(upd(C_i, t_1 t_2 \cdots t_{l-1}), t_l)$.

Algorithm 1: *OrderUpdate*

Input: set of all nodes (N), initial configuration (C_i), final configuration (C_f)
Result: consistent order of node updates (R), updates before which there are waits (R_w)

1 $R = R_w = P_0 \leftarrow \varnothing; k \leftarrow 1$ `// initialize` R, R_w, P_0 `and` k
2 $C_c \leftarrow C_i$ `//` C_c `starts with the initial value of` C_i
3 **while** $C_c \neq C_f$ **do** `// stop when` C_c `and` C_f `are equal`
4 $\quad U \leftarrow \{s \mid s \in N \wedge ((Y_a(s) \wedge Z_a(s)) \vee (Y_b(s) \wedge Z_b(s)) \vee (Y_c(s) \wedge Z_c(s)) \vee (Y_d(s) \wedge Z_d(s)) \vee (Y_e(s) \wedge Z_e(s)))\}$ `// valid nodes`
5 $\quad$ **if** $U = \varnothing$ **then** EXIT `// no consistent order of updates exists`
6 $\quad s = PickAndWait()$ `// by default, use Algorithm 2`
7 $\quad C_c \leftarrow (C_c \setminus out(s, C_i)) \cup out(s, C_f)$ `// update` C_c
8 $\quad N \leftarrow N - \{s\}$ `// remove updated nodes from node list`
9 **return** (R, R_w)

Algorithm 2: *SequentialPickAndWait*

$s = Pick(U)$ `// pick any valid node`
$R_w \leftarrow R_w.s$ `// by default, there is a wait after every update`
$R \leftarrow R.s$ `// append` s `to the end of result` R

Theorem 2. If a correct sequence of updates exists, then a careful sequence also exists.

4.3 Completeness of the OrderUpdate Algorithm

The OrderUpdate Algorithm (with the SequentialPickAndWait subroutine) is complete, i.e., if there exists any correct sequence, we find one. We can observe that if two nodes a and b are both valid in configuration C_c, then $upd(C_c, ab)$ and $upd(C_c, ba)$ are both consistent configurations. This property holds for any number of nodes and for all *careful* sequences, but not for all *correct* sequences. We prove this behavior in the following lemma, which is the key to confirming completeness of the OrderUpdate Algorithm.

Lemma 2. If $T = UVnY$ is a careful sequence, and $valid(upd(C_i, U), n)$, then $T' = UnVY$ is also careful.

In other words, Lemma 2 shows that if there are multiple valid nodes in some configuration C, then these nodes can be updated in any order. This is because once a node becomes valid, it does not become invalid. This is why we introduced careful sequences, because this lemma is not true for arbitrary correct sequences. Using this lemma, we can prove the completeness of Algorithm 1 (with the Algorithm 2 subroutine).

Theorem 3. Algorithm 1, using subroutine Algorithm 2, generates a correct order of updates R if one exists, and otherwise fails (in Line 5).

Running Time. Let $|V|$ be the number of nodes and $|E|$ be the number of edges in G. In each iteration of its outer loop, Algorithm 1 using *SequentialPickAndWait* (Algorithm 2) as a subroutine, makes a list of valid nodes and picks one to update. The set of valid nodes U in Line 4 can be found using a graph search on

Algorithm 3: *OptimalPickAndWait*

```
1 if k = 1 then                                // we do not need a wait before first node
2 |  P_0 ← U                                           // all nodes initially valid are P_0
3 if P_0 = ∅ then                                // we have to pick a lower priority node
4 |  P_0 ← U                                // all nodes in U become P_0 after waiting.
5 |  s = Pick(P_0); R ← R.s; R_w ← R_w.s; k ← k + 1; // pick P_0 node, append s
  |  to result R, add wait, increment number of rounds k
6 else
7 |  s = Pick(P_0); R ← R.s                 // pick any P_0 node, add s to result R
```

C_c for each node, which takes $O(|V|(|V| + |E|))$ steps. The loop runs $|V|$ times and updates each node, so the overall runtime is $O(|V|^2(|V|+|E|))$. This analysis relies on the fact that the graph search is implemented in a way that goes through each edge and node a constant number of times. Once a node has been visited, it is marked F, I, or B, based on whether the maximal paths downstream from it are maximal paths starting from it in C_i, C_f, or both. This ensures that we avoid visiting the node (and its outgoing edges) again.

5 Optimal OrderUpdate Algorithm

Thus far, we solved the consistent order update problem by generating a consistent sequence with only singleton sets. This corresponds to requiring a wait at every step of the update sequence, which does not allow any parallelism. However, we have seen in Section 2 that some nodes can be updated in parallel. In Section 3, we defined when a wait is needed in the sequence of updates. In this section, we provide a sequence of updates where there is a wait if and only if it is needed, solving the optimal version of the problem. We use Algorithm 1, but replace the subroutine *SequentialPickAndWait* (Algorithm 2) with *OptimalPickAndWait* (Algorithm 3). The algorithm returns a solution for the optimal consistent update problem in the following format.

Correct Waited Sequence. A correct waited sequence is a tuple (T, W) of node sequences without repetition, where W is a subsequence of T and $(T, W) = (t_1 t_2 \cdots t_{|N|}, w_1 w_2 \cdots w_{k-1})$, such that a consistent update sequence $S_1 S_2 \cdots S_k$ can be formed by taking $S_1 = \{t_1, \cdots, t_m\}$ where $t_{m_1} = w_1$, $\forall i \in (1, k) : S_i = \{t_{l_i}, \cdots, t_{m_i}\}$ where $t_{l_i} = w_{i-1}$ and $t_{m_i} = w_i$, and $S_k = \{t_{l_k}, \cdots, t_{|N|}\}$ where $t_{l_k} = w_{k-1}$.

Intuitively, T specifies a correct sequence of updates, with some waits, while W specifies the nodes, immediately before which a wait is placed. If we simply group the nodes between i-th and $(i + 1)$-st waits into a set S_{i+1} we obtain the consistent update sequence of Section 3. Considering solutions to the problem in the form of a sequence of nodes and waits simplifies the arguments we use to prove correctness and optimality.

Minimal Correct Waited Sequence. A *minimal correct waited sequence* is a correct waited sequence (T, W) such that $|W|$ is minimal. Since we always pick valid nodes, we need to prove that if a minimal correct waited sequence exists, then there exists a minimal correct waited sequence that updates only valid nodes.

Careful Waited Sequence. A *careful waited sequence* of updates $(T, W) = (t_1 t_2 \cdots t_{|N|}, w_1 w_2 \cdots w_{k-1})$ is a correct waited sequence s.t. $\forall j \in [1, |N|] : \mathit{valid}(\mathit{upd}(C_i, t_1 \cdots t_{j-1}), t_j)$ A *minimal careful waited sequence* is a careful waited sequence (T, W) s.t. $|W|$ is minimal. We prove the following for such sequences.

Theorem 4. If a minimal correct waited sequence exists, then a minimal careful sequence exists as well.

5.1 Condition for Waits

Partial Careful Waited Sequence. Given careful waited sequence $Z = (T = t_1 \cdots t_{|N|}, W = w_1 \cdots w_{k-1})$, a partial careful waited sequence is $Z' = (T' = t_1 \cdots t_r, W' = w_1 \cdots w_s)$ such that T' is a prefix of T and W' is a prefix of W. We start with a partial careful waited sequence with no nodes, and at every step adds a node while ensuring that the obtained sequence is a partial careful waited sequence, i.e., can be extended to a careful waited sequence.

Wait Condition. Consider a function *wait* that takes a partial careful waited sequence $S = (t_1 t_2 \cdots t_r, w_1 w_2 \cdots w_s)$ and node n s.t. $\mathit{valid}(C_i, U t_1 \cdots t_r)$ as an argument and returns *true* if there needs to be a wait before its update. Specifically: $\mathit{wait}(n, S) = \mathit{true}$ if and only if node $\exists x \in [1, r] : \neg \mathit{valid}(\mathit{upd}(C_i, t_1 \cdots t_x), n) \wedge \neg(\exists y \in [1, s], \exists z \in (x, r] : w_y = t_z)$, i.e., in the partial careful waited sequence, there must be a wait before updating a valid node n if and only if it was not valid until its dependencies were updated, and there was no wait after their update. In this case, n must be updated in a new round, after a wait.

We now show *completeness* of the wait condition, i.e., if a wait is needed (as defined in §3) after updating S and before updating n, then $\mathit{wait}(n, S)$ is true.

Lemma 3. If (1) n is the node picked for update, and (2) the partial careful waited sequence built before updating n is $S = (t_1 t_2 \cdots t_r, w_1 w_2 \cdots w_s)$, and (3) $w_s = t_y$ for some $y \in [1, r]$, and (4) we define $\forall x \in [1, r] : C_{t_x} = \mathit{upd}(C_i, t_1 \cdots t_x)$, and then $\mathit{wait}(n, S) \leftrightarrow C_{t_y} \cup \cdots \cup C_{t_r} \cup \mathit{upd}(C_{t_r}, n)$ is inconsistent.

5.2 Algorithm for Optimal Consistent Order Updates

The *OptimalPickAndWait* (Algorithm 3) subroutine minimizes waits, solving the optimal consistent update problem. We minimize waits by assigning priority P_0 (higher priority) or P_1 (lower priority) to nodes. Let S be a partial sequence. A node is in P_0 if and only if $\neg \mathit{wait}(n, S)$, i.e., P_0 nodes do not require waiting before update. A node is in P_1 if and only if $\mathit{wait}(n, S)$, i.e., we must wait before updating a P_1 node. We greedily update P_0 nodes first.

Correctness and optimality follow from the correctness argument in the previous section, and from Lemma 3. Intuitively, updating a node in P_0 which does not need a wait allows the P_1 list to build up. This means we need to place a single wait for as many P_1 nodes as possible. When we place a wait in the partial careful waited sequence, every valid node that was in P_1 moves to P_0. The last key property needed for the following theorems is that once a node acquires priority P_0, it retains priority P_0.

Theorem 5. Algorithm 1 with Algorithm 3 as its subroutine on Line 6 produces a correct waited sequence.

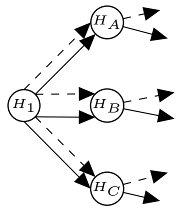

Fig. 6: Multiple sources.

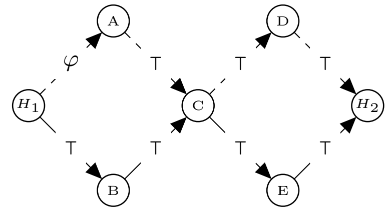

Fig. 7: Double diamond with symbolic forwarding rules.

Theorem 6. Algorithm 1 with Algorithm 3 as its subroutine on Line 6 produces a correct and optimal waited sequence of updates, if one exists.

Running Time. The OrderUpdate Algorithm with the *OptimalPickAndWait* subroutine has the same time complexity that it had with the *SequentialPickAndWait* subroutine. The *OptimalPickAndWait* subroutine introduces a priority-based node selection mechanism—after every wait, it simply moves nodes from the valid set U to the higher priority list P_0, which requires only $O(|N|)$ additional steps in each iteration.

6 Discussion

Multiple hosts and sinks. We can extend our single-source approach to a network with multiple sources $H_A, H_B, H_C, \cdots$. To do this, we assume that there is a master source H_1, and every actual source is connected to H_1, as shown in Figure 6. This approach works because we update every node only once, meaning we cannot artificially disable and then re-enable some sources and keep others.

Multiple packet types. Our approach can be applied when there are multiple (discrete) packet types, as long as each forwarding rule matches on a *single* packet type—in this case, we compute an update for each packet type, and perform these (rule-granularity) updates independently. In the more realistic case with *symbolic* forwarding rules (i.e., matching based on *first-order formulae over packet header fields*), deciding whether a consistent update exists is co-np-hard. Specifically, there is a reduction from SAT to this problem. We can consider each edge in a configuration as being labeled by a formula, and only packets whose header fields satisfy this formula can be forwarded along that edge. Consider a double diamond (Figure 7) with one edge labelled by φ, and all other edges labelled with *true* ($\top$). We have seen that a consistent update for this double diamond example is not possible in the situation where packets (of any type) can flow along all of the edges, so we can see that *there exists a consistent update if and only if φ is unsatisfiable.* This completes the reduction.

7 Related Work

Consistency. Our core problem is motivated by earlier work by Reitblatt et al. [16] that proposed *per-packet consistency* and provided basic update mechanisms.

Exponential Search-Based Network Update Algorithms. There are various approaches for producing a sequence of switch updates guaranteed to respect certain path-based consistency properties (e.g., properties representable using temporal logic, etc.). For example, McClurg et al. [15] use counter-example guided search and incremental LTL model checking, FLIP [17] uses integer linear programming, and CCG [19] uses custom reachability-based graph algorithms. Other works such as Dionysus [7], zUpdate [8], and Luo et al. [12], seek to perform updates with respect to quantitative properties.

Complexity results. Mahajan and Wattenhofer [13] propose dependency-graphs as a representation for network updates, and propose properties that can be solved using this general approach, including loop freedom, which is handled in a minimal way. Yuan et al. [18] detail general algorithms for building dependency graphs and using these graphs to perform a consistent update. Förster et al. [6] show that for *blackhole-freedom*, computing an update with a minimal number of rounds is NP-hard (assuming memory limits on switches). They also show NP-hardness results for rule-granular loop-free updates with maximal parallelism. Per-packet consistency in our problem is stronger than loop and blackhole freedom, but we consider solutions where each switch is updated *once*, and where a switch update replaces the entire old forwarding table with the new one.

Förster and Wattenhofer [5] examine loop-freedom, showing that maximizing the number for forwarding rules updated simultaneously is NP-hard. Ludwig et al. [10] show how to minimize the number of update rounds with respect to loop-freedom. They show that deciding whether a k-round schedule exists is NP-complete, and they present a polynomial algorithm for computing a weaker variant of loop-freedom. Amiri et al. [1] present an NP-hardness result for greedily updating a maximal number of forwarding rules in this context. Additionally, Ludwig et al. [9] investigate optimal updates with respect to a stronger property, namely *waypoint enforcement* in addition to loop freedom. They produce an update sequence with a minimal number of waits, using mixed-integer programming. Ludwig et al. [11] show that the decision problem is NP-hard.

Mattos et al. [14] propose a relaxed variant of per-packet consistency, where a packet may be processed by several subsequent configurations (rather than a *single* one), and present a polynomial graph-based algorithm for computing updates. Dudycz et al. [4] show that simultaneously computing *two* network updates while minimizing the number of switch updates ("touches") is NP-hard. Brandt et al. [2] give a polynomial algorithm to decide if a congestion-free update is possible when flows are "splittable" and/or not restricted to be integer.

8 Conclusion

We presented a polynomial-time algorithm to find a consistent update order for a single packet type. We then described a modification to the algorithm which finds a consistent update order with a minimal number of waits. Finally, we proved that this modification is correct, complete, and optimal.

References

[1] Saeed Akhoondian Amiri, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Transiently Consistent SDN Updates: Being Greedy is Hard. *SIROCCO*, 2016.

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

[3] Pavol Černý, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. Optimal Consistent Network Updates in Polynomial Time (Extended Version). *arXiv:1607.05159*, 2016.

[4] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't Touch This: Consistent Network Updates for Multiple Policies. *DSN*, 2016.

[5] Klaus-Tycho Förster and Roger Wattenhofer. The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration. *ICCCN*, 2016.

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

[7] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic Scheduling of Network Updates. *SIGCOMM*, 2014.

[8] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. *SIGCOMM*, 2013.

[9] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. *HotNets*, 2014.

[10] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling Loop-free Network Updates: It's Good to Relax! *PODC*, 2015.

[11] Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. Transiently Secure Network Updates. *SIGMETRICS*, 2016.

[12] Shouxi Luo, Hongfang Yu, Long Luo, and Le Min Li. Arrange Your Network Updates as You Wish. *IFIP*, 2016.

[13] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. *HotNets*, 2013.

[14] Diogo Menezes Ferrazani Mattos, Otto Carlos Muniz Bandeira Duarte, and Guy Pujolle. Reverse Update: A Consistent Policy Update Scheme for Software Defined Networking. *IEEE Communications Letters*, 2016.

[15] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient Synthesis of Network Updates. *PLDI*, 2015.

[16] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. *SIGCOMM*, 2012.

[17] Stefano Vissicchio and Luca Cittadini. FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates. *INFOCOM*, 2016.

[18] Yifei Yuan, Franjo Ivančić, Cristian Lumezanu, Shuyuan Zhang, and Aarti Gupta. Generating Consistent Updates for Software-Defined Network Configurations. *HotSDN*, 2014.

[19] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing Customizable Consistency Properties in Software-Defined Networks. *NSDI*, May 2015.