Congestion-Free Rerouting of Network Flows: Hardness and an FPT Algorithm

Esra Ceylan[1,2] Krishnendu Chatterjee[2] Stefan Schmid[1] Jakub Svoboda[2]
[1] TU Berlin, Germany [2] ISTA, Austria

***Abstract*—Given the increasingly stringent requirements on the performance and efficiency of communication networks, over the last years, great efforts have been made to render networks more flexible and programmable. In particular, modern networks support a flexible rerouting of flows, e.g., depending on the dynamically changing traffic or network conditions. However, the underlying algorithmic problems are still not well-understood today.**

In this paper, we revisit the k-Network Flow Update problem that asks for a schedule to reroute k unsplittable flows from their current paths to the given new paths, in a congestion-free manner in a capacitated network. We show that the problem is already NP-hard for three flows on directed acyclic graphs. Our main contribution is an efficient algorithm for sparse networks; specifically the algorithm is fixed parameter tractable in the number of flows and the treewidth of a graph that is the union of all flows. Our results also settle the open complexity question in the literature.

I. Introduction

With the popularity of datacentric applications and machine learning, the traffic in communication networks is growing explosively, especially to, from and inside datacenters. Accordingly, over the last years, great efforts have been made to improve the performance and resource efficiency of communication networks [20]. In particular, networks are becoming more and more programmable and flexible, supporting the dynamic adaption of resources and flows. For example, in order to make optimal use of their infrastructure, internet service providers can leverage such flexibilities to operate their networks more adaptively, e.g., performing dynamic traffic engineering or reacting to changes in security policies [24].

How to update networks and reroute flows in a fast and consistent manner however introduces its own challenges, and the underlying algorithmic and optimization problems are still not well understood. Indeed, the problem has already been studied in hundreds of publications, see the survey by Foerster et al. [10]. Most existing results revolve around heuristics.

This paper revisits the k-Network Flow Update problem, a fundamental reconfiguration problem arising in the context of such dynamic communication networks. In a nutshell, we are given a set of k unsplittable flows in a capacitated network connecting n routers, called *nodes*.

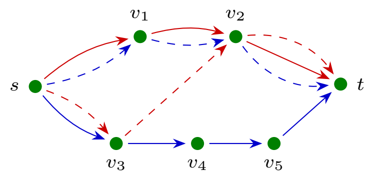

Fig. 1. Example of a flow network with two update flow pairs in red P_r and blue P_b. The old flows are indicted with a solid line, the new flows with dashed ones. The demand of each flow is 1. Edges (s, v_3) and (v_2, t) have capacity 2, the remaining edges have capacity 1.

The task is to find a feasible schedule for rerouting these flows from their current paths to their given new paths, by changing the forwarding rules at the nodes.

Figure 1 gives an example of the network flow update problem with two unsplittable flows where a rerouting of the flows is possible. For further details see Section II.

Amiri et al. [2] showed that the problem is NP-hard for six flows even if the union of the old and new paths are acyclic. For two flow pairs they present a polynomial-time algorithm.

Our Contributions: Motivated by the performance benefits of more adaptable communication networks, we study fast and congestion-free rerouting algorithms for network flows. We first close the gap in the literature and present a tight characterization of the complexity of the k-Network Flow Update problem by showing that it is NP-hard already for $k = 3$ (see Theorem 2). In our reduction not only the flows are acyclic, but also the graph is very simple since removing one edge makes the whole graph acyclic.

We then present an efficient algorithm for sparse networks, specifically, a fixed-parameter tractable algorithm parameterized by the treewidth of the graph and the number of flows.

Related Work: How to reroute flows in a congestion-free manner is a fundamental question, which is receiving much attention recently [10]. In general, the problem studied in this paper belongs to the family of combinatorial reconfiguration problems which asks for a transformation of one configuration into another, subject to some (reconfiguration) rules. Such reconfiguration problems have already been studied in many contexts, including puzzles and games (such as Rubik's cube) [32], satisfiability [14], independent sets [15], vertex coloring [6], or matroid bases [18], to just name a few.

The problem of how to consistently update routes has attracted much interest recently [5], [19], [23], [27], [30], in particular in the context of software-defined networks, and it is motivated by unpredictable router update times [19], [22]. For an overview, we refer to a recent survey by Foerster et al. [10]. In a seminal work by Reitblatt et al. [30], a strong *per-packet consistency* notion has been studied, which is well-aligned with the strong consistency properties usually provided in traditional networks [7]. Mahajan and Wattenhofer [27] started exploring the benefits of relaxing the per-packet consistency property, while *transiently* providing only essential properties like loop-freedom. The authors also present a first algorithm that quickly updates routes in a transiently loop-free manner. Their study was recently refined, where the authors also establish hardness results [3], [11], [12]. Furthermore, they focus on the problem of minimizing the number of scheduling rounds [25], initiate the study of multiple policies [9], and introduce additional transient routing constraints related to waypointing [24], [26]. However, none of these papers considers bandwidth capacity constraints.

Congestion is known to negatively affect application performance and user experience. The seminal work by Hongqiang et al. [23] on congestion-free rerouting was extended in several papers, using static [4], [16], [31], [35], dynamic [34], or time-based [28], [29] approaches. Vissicchio et al. presented FLIP [33], which combines per-packet consistent updates with order-based rule replacements, in order to reduce memory overhead. Moreover, Hua et al. [17] initiated the study of adversarial settings with packet tampering and packet dropping.

However, bandwidth capacity constraints have so far mainly been considered in strong, per-packet consistent settings, and for splittable flows. This is both costly (splittable flows introduce overheads) as well as restrictive (per-packet consistent updates require traffic marking and render many problem instances infeasible). Amiri et al. [1] presented a polynomial-time algorithm for acyclic flow graphs, to compute feasible (possibly very long) update schedules. In follow-up work [2], they presented a polynomial-time algorithm that computes the fastest update schedule for two flows, and also showed that the problem is already NP-hard for six flows.

Paper Outline: We introduce our model formally in Section II. In Section III, we prove that k-NETWORK FLOW UPDATE is NP-hard already for $k = 3$. In Section IV, we present a fixed-parameter tractable algorithm for the problem with bounded treewidth and bounded number of flow pairs.

Due to space constraints, proofs for results or additional material marked with $(*)$ are deferred to the full version of the paper.

II. MODEL

The problem considered in this paper can be described in terms of edge-capacitated directed graphs. Without loss of generality, we consider directed graphs with only one source vertex and one terminal vertex.

Definition 1 (Flow Network)**.** A flow network (G, c, s, t) consists of a directed graph G, a capacity function $c\colon E(G) \to \mathbb{R}_0^+$ assigning a non-negative capacity $c(e)$ to every edge $e \in E(G)$, and two vertices $s, t \in V(G)$, where s is the source and t the terminal.

An (s,t)-flow F of demand $d(F) \in \mathbb{N}$ is a directed path from s to t in a flow network such that $d(F) \leq c(e)$ for each $e \in E(F)$. Given a set of (s,t)-flows $\mathcal{F} = \{F_1, \dots, F_k\}$ we call $\mathcal{F}$ valid, if $\sum_{i\in[k]\colon e\in E(F_i)} d(F_i) \leq c(e)$ holds for each $e \in E(G)$.

Definition 2 (Update Flow Network)**.** An update flow pair $P = (F^o, F^u)$ consists of two (s,t)-flows, the old flow F^o and the update (or new) flow F^u, both having equal demand, i.e., $d(F^o) = d(F^u)$. Accordingly, we denote a family of update flow pairs by $\mathcal{P} = \{P_1, \dots P_k\}$.

Given a flow network (G, c, s, t) and a family $\mathcal{P}$ of update flow pairs with $P_i = (F_i^o, F_i^u)$, we say $(G, c, s, t, \mathcal{P})$ is an update flow network if

1) $V(G) = \bigcup_{i\in[k]} V(F_i^o \cup F_i^u)$,
2) $E(G) = \bigcup_{i\in[k]} E(F_i^o \cup F_i^u)$, and
3) $\{F_1^o, \dots, F_k^o\}$ and $\{F_1^u, \dots, F_k^u\}$ are valid.

Figure 1 gives an example of an update flow network with two update flow pairs shown in red and blue.

A flow in an update flow network can be rerouted by updating the outgoing edges (the forwarding rules) of the vertices along its path, i.e., by blocking the outgoing edge of the old flow and by allowing traffic along the outgoing edge of the new flow.

Definition 3 (Update)**.** Given an update flow network $(G, c, s, t, \mathcal{P})$, an update is a pair $(v, P) \in V(G) \times \mathcal{P}$. An update (v, P) with $P = (F^o, F^u)$ is resolved by deactivating the current outgoing edge (from F^u) of the flow P starting at v (note that there is only one) and activating the outgoing edge of F^u starting at v. The deactivated edge of F^o is no longer used by the flow pair P, but now the newly activated edge of F^u is.

For a flow pair $P = (F^o, F^u) \in \mathcal{P}$ and set of updates $U \subseteq V(G) \times \mathcal{P}$, graph $P(U)$ denotes the update flow network consisting of vertices $V(F^o \cup F^u)$ and the edges of P that are active after resolving all updates in U.

We are now able to determine, for a given set of updates, which edges we can and which we cannot use for routing. In the end, we want to describe a process of reconfiguration steps, starting from the *initial state*, in which no update has been resolved, and finishing in a state where the only active edges are exactly those of the new flows of every update flow pair.

Definition 4 (Transient Flow)**.** The flow pair P is called transient for some set of updates $U \subseteq V(G)\times\mathcal{P}$, if $P(U)$ contains a unique valid (s,t)-flow $T_{P,U}$. If there is a family $\mathcal{P} = \{P_1, \dots P_k\}$ of update flow pairs, we call $\mathcal{P}$ a transient family for a set of updates $U \subseteq V(G) \times \mathcal{P}$, if and only if every $P \in \mathcal{P}$ is transient for U.

A transient flow looks like a path of active edges for flow F, which starts at the source vertex and ends at the terminal vertex. Note that there may be some active edges connected to this path, but they cannot be used to route the flow since $T_{P,U}$ is unique after resolving U. In order to ensure transient consistency, the updates of these outgoing edges need to be scheduled over time.

Definition 5 (Update Sequence)**.** An update sequence $(\sigma_i)_{i\in[|V(G)\times\mathcal{P}|]}$ is an ordering of $V(G)\times\mathcal{P}$. We denote the set of updates that are resolved after step i by $U_i = \bigcup_{j=1}^{i}\sigma_j$, for all $i \in [|V(G) \times \mathcal{P}|]$.

Definition 6 (Feasible Update Sequence)**.** An update sequence σ is feasible, if for each $i \in [|V(G) \times \mathcal{P}|]$, the family $\mathcal{P}$ is transient for U_i.

A feasible update sequence for the example given in Figure 1 is the following: (v_2, P_b), (v_2, P_b), (v_3, P_r), (s, P_b), (s, P_r).

k-NETWORK FLOW UPDATE
Input: Update flow network $(G, c, s, t, \mathcal{P})$, $k = |\mathcal{P}|$.
Question: Is there a feasible update sequence?

III. NP-HARDNESS

To prove that the k-NETWORK FLOW UPDATE is NP-hard even for three flow pairs, we first define blocks and dependency graphs. The dependency graph concisely captures the constraints imposed by the flows. Then, we prove that for any graph D satisfying some conditions, we can create a flow network such that its dependency graph is D. Finally, we reduce the NP-hard problem INDEPENDENT SET [13] to k-NETWORK FLOW UPDATE.

Blocks: Let (F^o, F^u) be an update flow pair where $F^o \cup F^u$ is acyclic. We denote by $\preceq$ a topological ordering $v_1, v_2, \dots, v_{|F^o\cup F^u|}$ of the vertices in $V(F^o \cup F^u)$, where $s = v_1$ and $t = v_{|F^o\cup F^u|}$. Furthermore, let $z_1, \dots, z_{|F^o\cap F^u|}$ be the vertices in $V(F^o \cap F^u)$, ordered w.r.t. $\preceq$.

The jth block of the update flow pair F, denoted b_j, is the subgraph of $F^o \cup F^u$ induced by the set $\{v \in V(F^o \cup F^u) \mid z_j \preceq v \prec z_{j+1}\}$, where $j \in [|F^o \cap F^u| - 1]$. The flow pair (F^o, F^u) restricted to block b_j is denoted by $(F^o_{b_j}, F^u_{b_j})$.

In order to reroute the old flow to the new flow in a block b_j all vertices in b_j need to be updated w.r.t. (F^o, F^u). Hence, we say block b_j of (F^o, F^u) is updated if all vertices in b_j are updated w.r.t. (F^o, F^u). In Figure 1 the blue flow P_b consists only of one block, whereas the red flow P_r consists of two blocks.

Dependency graph: Different flow pairs can share edges. Each edge has a maximum capacity which introduces dependencies on how the blocks need to be updated. We capture this with the dependency graph. The dependency graph D of an update flow network is a directed hypergraph. Since we consider only three acyclic flow pairs in this section, this gives rise to three types of (hyper-)edges, i.e., $E(D) = E_\alpha \cup E_\beta \cup E_\gamma$. We construct D as follows:

1) Create for each block b of each flow pair in G one vertex b. We use b for a block and a vertex in $V(D)$ but it will be clear to which we are referring to.
2) Let $b_1, b_2, b_3 \in V(D)$. We add an
   a) α-edge $b_1 \to b_2$ if there is an edge $e \in E(F^u_{b_1}) \cap E(F^o_{b_2})$ with $c(e) < d(F^u_{b_1}) + d(F^o_{b_2})$.
   b) β-edge $b_1 \to \{b_2, b_3\}$ if there exists an edge $e \in E(F^u_{b_1}) \cap E(F^o_{b_2}) \cap E(F^o_{b_3})$ with
      i) $c(e) < d(F^u_{b_1}) + d(F^o_{b_2}) + d(F^o_{b_3})$,
      ii) $d(F^u_{b_1}) + d(F^o_{b_2}) \leq c(e)$, and
      iii) $d(F^u_{b_1}) + d(F^o_{b_3}) \leq c(e)$.
   c) γ-edge $\{b_1, b_2\} \to b_3$ if there exists an edge $e \in E(F^u_{b_1}) \cap E(F^u_{b_2}) \cap E(F^o_{b_3})$ with
      i) $c(e) < d(F^u_{b_1}) + d(F^u_{b_2}) + d(F^o_{b_3})$,
      ii) $d(F^u_{b_1}) + d(F^o_{b_3}) \leq c(e)$, and
      iii) $d(F^u_{b_2}) + d(F^o_{b_3}) \leq c(e)$.

Since for each flow pair the union of the two flows is acyclic, we observe that the two (resp. three) blocks involved in an α- (resp. β- or γ-)edge are distinct and belong to different flow pairs.

The edges in the dependency graph give information about the order in which the blocks in G can be updated. We observe that block b_1 can only be updated if: (i) for all α-edges $b_1 \to b_2$, block b_2 is updated; (ii) for all β-edges $b_1 \to \{b_2, b_3\}$, block b_2 or b_3 is updated; and (iii) for all γ-edges $\{b_1, b_2\} \to b_3$, block b_2 is not updated or b_3 is updated.

Not every graph only with these types of edges is the dependency graph of some update flow network. However, we show that for specific graphs D, we can always construct an update flow network $(G, c, s, t, \mathcal{P})$ such that its dependency graph is isomorphic to D. The next lemma proves this for graphs where the vertices can be partitioned into layers where the edges only point to the next two layers, e.g., see Figure 4.

Lemma 1. *Let D be a graph only with α-, β-, and γ-edges satisfying the following properties: It holds $V(D) = L_0 \dot\cup \dots \dot\cup L_\ell$ with $\ell \bmod 3 = 0$ and for each*

1) α-edge $b_1 \to b_2$ it holds that if $b_1 \in L_i$ for $i \in [\ell]$

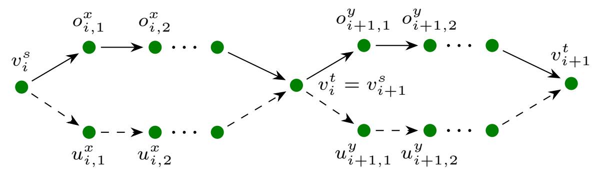

Fig. 2. Creating the flows for two vertices $b_i, b_{i+1} \in V(D)$. The old flow is drawn with solid edges and the new path with dashed edges.

then $b_2 \in L_{i-1} \cup L_{i-2}$ or if $b_1 \in L_0$ then $b_2 \in L_\ell$;

2) *β-edge $b_1 \to \{b_2, b_3\}$ it holds that if $b_1 \in L_i$ for $i \in [\ell] \setminus \{1\}$ then $b_2 \in L_{i-1}, b_3 \in L_{i-2}$ or $b_2 \in L_{i-2}, b_3 \in L_{i-1}$; and*
3) *γ-edge $\{b_1, b_2\} \to b_3$ it holds that if $b_3 \in L_i$ for $i \in [\ell - 2]$ then $b_1 \in L_{i+1}, b_2 \in L_{i+2}$ or $b_1 \in L_{i+2}, b_2 \in L_{i+1}$.*

Then, there exists an update flow network $(G, c, s, t, \mathcal{P})$ with $|\mathcal{P}| = 3$, the union of the old and new flow for each flow pair is acyclic, and $|V(G)| \in \mathcal{O}(|V(D)|)$ such that D is isomorphic to the dependency graph of $(G, c, s, t, \mathcal{P})$.

Proof. The proof consists of three steps: (i) Color $V(D)$ with three colors and assign them to three flows, (ii) create for each $b \in V(D)$ a block in the flow network, and (iii) connect the blocks and model the dependencies.

Coloring $V(D)$: First, we need a 3-coloring of the vertices of D. The color of each vertex indicates to which of the three flow pairs it belongs. Hence, we need a coloring where the two (resp. three) vertices of every α- (resp. β- or γ-) edge are colored differently. We color the vertices of layer $i \in \ell \cup \{0\}$ with color c_1 (resp. c_2 or c_3) if $i \bmod 3 = 0$ (resp. $i \bmod 3 = 1$ or $i \bmod 3 = 2$).

By Statement (1) vertices incident to an α-edge have different colors with this coloring. By Statement (2) and (3) each β- and γ-edge is incident to three distinct and consecutive layers. Hence, our coloring assigns to each of three vertices incident to a β- and γ-edge distinct colors. This means we obtain proper coloring.

Creating the flows: The three colors of our coloring correspond to the three flow pairs in our network. We show how to create the flows for each color.

First, we choose an arbitrary ordering $b_1, b_2, \ldots$ of the vertices in $V(D)$ of one color. Then, we create for each $b_i \in V(D)$ an old and a new path each of length $2\left(\delta^-_D(b_i) + \delta^+_D(b_i) + 1\right)$, where $\delta^-_D(b_i)$ (resp. $\delta^+_D(b_i)$) is the in-degree (resp. out-degree) of b_i in D. The old path consists of the vertices $v^s_i, o^x_{i,1}, o^x_{i,2}, \ldots, v^t_i$ and the new path of $v^s_i, u^x_{i,1}, u^x_{i,2}, \ldots, v^t_i$. These two flows have exactly two common vertices, the start-vertex v^s_i and the end-vertex v^t_i. The index x of vertices $o^x_{i,1}, o^x_{i,2}, u^x_{i,1}, u^x_{i,2}$ corresponds to a neighbor $x \in N^-_D(b_i) \cup N^+_D(b_i)$.

Next, we merge the created blocks according to the ordering $b_1, b_2, \ldots$ to obtain a continuous flow as follows: For two consecutive blocks b_i and b_{i+1}, we merge v^t_i with v^s_{i+1} (see Figure 2).

Connecting the flows: We have to connect the three flows to create the desired dependencies. We take an arbitrary ordering of the edges in $E(D)$, consider one edge after the other and do the following (see Figure 3):

1) For an α-edge $b_1 \to b_2$, we merge vertices $u1^{b_2}_{b_1}$, $o1^{b_1}_{b_2}$ as well as $u2^{b_2}_{b_1}$, $o2^{b_1}_{b_2}$ and set the capacity of the edge $(u1^{b_2}_{b_1}, u2^{b_2}_{b_1})$ to 1.
2) For a β-edge $b_1 \to \{b_2, b_3\}$, we merge vertices $u1^{b_2}_{b_1}$, $o1^{b_1}_{b_2}$, $o1^{b_2}_{b_3}$ as well as $u2^{b_2}_{b_1}$, $o2^{b_1}_{b_2}$, $o2^{b_2}_{b_3}$ and set the capacity of the edge $(u1^{b_2}_{b_1}, u2^{b_2}_{b_1})$ to 2.
3) For a γ-edge $\{b_1, b_2\} \to b_3$, we merge vertices $u1^{b_2}_{b_1}$, $u1^{b_1}_{b_2}$, $o1^{b_2}_{b_3}$ as well as $u2^{b_2}_{b_1}$, $u2^{b_1}_{b_2}$, $o2^{b_2}_{b_3}$ and set the capacity of the edge $(u1^{b_2}_{b_1}, u2^{b_2}_{b_1})$ to 2.

Finally, we have to connect the three flows to a common source s and terminal t.

We created an update flow network with $|\mathcal{P}| = 3$ and $|V(G)| \in \mathcal{O}(|V(D)|)$. It is straightforward that this models the necessary dependencies. The union of each flow pair is acyclic because each edge in D uses at most three blocks that belong to different flows and when rerouting a flow it does not intersect any other rerouting of this flow pair. □

Theorem 2. *The k-*Network Flow Update *problem is* NP*-hard even if the following holds: (i) $k = 3$, (ii) each flow pair is acyclic with demand 1, and (iii) the capacities of the flow network are in $\{1, 2\}$.*

Proof sketch. We provide a polynomial-time reduction from Independent Set [13]. Given an instance $(\hat{G}, h)$ of Independent Set we will create a dependency graph D that satisfies the conditions of Lemma 1. Hence, we can also create from D an update flow network. In this dependency graph D we will only be able to update blocks corresponding to independent vertices of $\hat{G}$ in the beginning. The main idea is that we can only resolve all the blocks in D if we can update at least h blocks representing an independent set in $\hat{G}$ first.

We proceed as follows: (i) Create a dependency graph D out of $\hat{G}$, (ii) build a gadget to count the number of blocks corresponding to independent vertices of $\hat{G}$, and (iii) the proof of correctness of the reduction.

Create Dependency Graph D: Let $(\hat{G}, h)$ be an Independent Set instance. We define $\hat{n} :- |V(\hat{G})|$. First, create block u. For each vertex $\hat{v}_i \in V(\hat{G})$, create blocks v_i and v'_i. For each edge $\{\hat{v}_i, \hat{v}_j\} \in E(\hat{G})$, create block $w_{i,j}$. Let $\{u\}$ be layer 0, denoted by L_0, all blocks $w_{i,j}$ are in layer 1, denoted by L_1, all blocks v'_i are in L_2, and all blocks v_i are in L_3 (see Figure 4).

Between these blocks, we add the following edges: For every $i \in [\hat{n}]$, we add the α-edge $v_i \to v'_i$. For every edge $\{\hat{v}_i, \hat{v}_j\} \in E(\hat{G})$ (suppose $i < j$) we add the γ-edge $\{v'_i, v_j\} \to w_{i,j}$ and the α-edge $w_{i,j} \to u$.

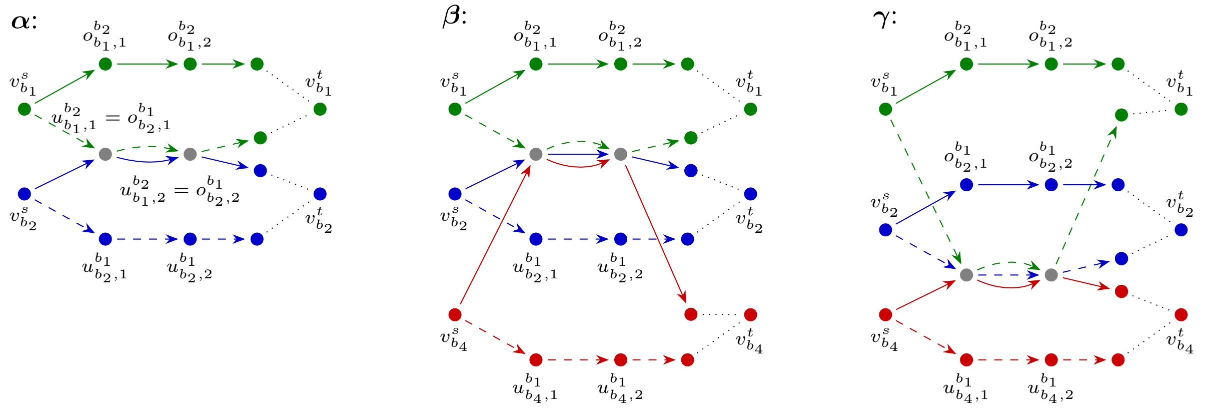

Fig. 3. Connecting the flows. Left: α-edge $b_1 \to b_2$. Middle: β-edge $b_1 \to \{b_2, b_3\}$. Right: γ-edge $\{b_1, b_2\} \to b_3$.

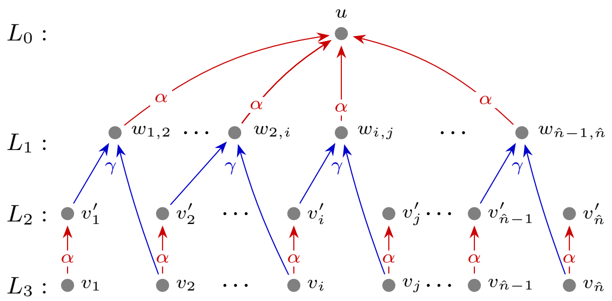

Fig. 4. The first four layers of the dependency graph D in Theorem 2, where $\{\hat{v}_1, \hat{v}_2\}, \{\hat{v}_2, \hat{v}_i\}, \{\hat{v}_i, \hat{v}_j\}, \{\hat{v}_{n-1}, \hat{v}_n\} \in E(\hat{G})$. Red arcs correspond to α-edges and blue arcs to γ-edges.

Claim 2.1 ($*$). *If u is not updated, then all vertices in L_3 that can be updated represent independent vertices in $\hat{G}$.*

Before we construct the layers to count the number of updated blocks corresponding to independent vertices in $\hat{G}$, we describe the OR-gadget. Given blocks $X = \{x_1, x_2, \ldots, x_\ell\}$ in one layer of D, let $\mathsf{OR}(X)$ be a gadget that (i) contains at most $2\ell+1$ vertices, (ii) spans at most $2\lceil \log \ell \rceil$ layers, and (iii) has one vertex v in the last layer where v can be updated if and only if at least one vertex from X was updated.

We describe the construction inductively: For $\ell = 1$, we have $X = \{x_1\}$ and the construction consists of one additional block x'_1 in a new layer and an α-edge $x'_1 \to x_1$. This fulfills the first two conditions and x'_1 can only be updated if x_1 is updated.

For $\ell \geq 2$, we create for every $t \in [\lfloor \ell/2 \rfloor]$ a block x'_{2t} with the α-edge $x'_{2t} \to x_{2t}$ and a block s_t with the β-edge $s_t \to \{x_{2t-1}, x'_{2t}\}$. If ℓ is odd, then we also add the α-edge $s_{\lceil \ell/2 \rceil} \to x_\ell$. With $S = \{s_1, s_2, \ldots, s_{\lceil \ell/2 \rceil}\}$, we then construct $\mathsf{OR}(S)$.

We can prove by induction that this construction of $\mathsf{OR}(X)$ satisfies all desired properties ($*$).

Construction of Counting Layers ($*$): We construct further layers that count the number of updated vertices in L_3. There are always some auxiliary layers and then a special counting layer called $\mathcal{L}_i$ which consists of blocks split into groups. Every group contains 2^i blocks and there are $\hat{n}/2^i$ groups in $\mathcal{L}_i$. For counting layer $\mathcal{L}_i$, a block in the j-th group at position p is denoted by $x_{j,p}$, where $p \in [2^i]$ and $j \in [\frac{\hat{n}}{2^i} - 1] \cup \{0\}$.

In the light of Claim 2.1, every $\mathcal{L}_i$ counts the size of the independent set vertices for some subset of blocks. The final counting layer $\mathcal{L}_{\log \hat{n}}$ counts the number of updated blocks in L_3, thus the independent set in $\hat{G}$.

Claim 2.2 ($*$). *Vertex $x_{j,p} \in \mathcal{L}_i$ can only be updated if at least p vertices in $\{v_{2^i \cdot j}, v_{2^i \cdot j+1}, \ldots, v_{2^i \cdot (j+1)-1}\} \subseteq L_3$ are updated.*

Finally, we connect block $u \in L_0$ with $v_{0,h} \in \mathcal{L}_{\log \hat{n}}$ by the α-edge $u \to v_{0,h}$. This construction of D satisfies the conditions of Lemma 1, has $\mathcal{O}(\hat{n}^2)$ vertices, and the number of layers is divisible by 3. Every edge spans at most three consecutive layers and is oriented upwards.

Correctness: Let C be a size-h independent set of $\hat{G}$. Following blocks can be updated sequentially: in layers L_2 and L_3 that correspond to C, then all counting layers greedily, and finally the rest. Conversely, if no independent set of $\hat{G}$ has size at least h, then block $v_{0,h} \in \mathcal{L}_{\log \hat{n}}$ cannot be updated by Claim 2.2. $\square$

IV. Fixed-Parameter Tractable Algorithm

In this section, we present a fixed-parameter tractable algorithm for k-Network Flow Update solves the problem in polynomial time parameterized by the number of flows and the treewidth.

Given flow networks, we first show how to construct an ED-graph for any number of flows. Then, we prove that the treewidth of the ED-graph is not too large compared to the treewidth of G. Finally, we show a fixed-parameter tractable algorithm that decides the feasibility of the problem.

Tree Decomposition and Treewidth: We recall the notion of treewidth, see [8, Sec. 12.4, p. 355] for an overview. Given a graph $D = (V, E)$a *tree decomposition* $T = (V_T, E_T)$ of D is a tree, where we refer to nodes of the tree as bags. Every bag B represent a set of vertices of D and T satisfies the following: 1) $\bigcup_{B \in V_T} B = V$; 2) with $E_B = \{(u, v) \in E \mid u, v \in B\}$, we have $\bigcup_{B \in V_T} E_B = E$; and 3) for all $X, Y, Z \in V_T$, if Y is on the unique path from X to Z, then $X \cap Z \subseteq Y$. The *width* of a tree decomposition is the maximal size of a bag minus one, i.e., $\max_{B \in V_T} |B| - 1$. For a graph D, *treewidth* of D is the minimum width over all tree decompositions of D and we denote it $tw(D)$.

ED-graph: Given $(G, c, s, t, \mathcal{P})$, we construct the *extended dependency graph (ED-graph)* $D = (V_D, E_D)$. First, we decompose the flows to blocks as in Section III. Blocks of flows are vertices in D. Two vertices v and u from V_D are connected by an edge if the respective blocks share an edge in G. Here, in ED-graph, we do not need special types of edges, one α-edge in d-graph creates one edge in ED-graph, one β or γ-edge in d-graph creates three edges in ED-graph, one edge between every two vertices participating in the β or γ-edge.

Lemma 3. *Given $(G, c, s, t, \mathcal{P})$ with k flow pairs and its ED-graph D, then $tw(D) \leq k \cdot tw(G)$.*

Proof. We modify G to G', such that: (i) $tw(G') \leq k \cdot tw(G)$; and (ii) graph D is a minor of G'. Since the minor operations do not increase the treewidth, this means $tw(D) \leq tw(G')$. Graph G' is the Cartesian product of G and K_k, then $tw(G') \leq k \cdot tw(G)$. Observe that D is minor of G' where every copy of vertex from G represents j-th flow. ☐

For a rooted tree decomposition, we construct a dependency oracle that given a bag and ordering returns true if and only if the given ordering satisfies all capacity conditions. Note that every condition on capacity needs to be stored only once and in the highest (closest to the root) bag that contains all variables in that condition. As usual, the algorithm caches the results of the same calls and except for true or false it can return the ordering. We can find a constant approximation of the treewidth of a graph G with n vertices in time
$\mathcal{O}(2^{tw(G)n})$ [21].

Theorem 4. *Given the update flow network $(G, c, s, t, \mathcal{P})$ where G has n vertices and $tw(G) = tw$ and update pairs $(P_1, \ldots, P_k)$ where for every flow the union of old and new flow is acyclic, Algorithm 1 decides the feasibility in time $2^{\mathcal{O}(tw \cdot k \log(tw \cdot k))} n$.*

Proof. Note that every edge (there are only $n \cdot k$ of them) introduces only one condition. From Lemma 3, we have that the treewidth of the extended dependency graph of G is at most $t \cdot k$ and the number of conditions is kn.

Algorithm 1 FPT Algorithm

Input: Dependency graph D, rooted tree decomposition T of D, dependency oracle f

```
1: procedure SOLVE($T$,$f$, $B$, $o$)
2:     for all permutations $o_1$ of $B$ extending $o$ do
3:         if $f(B, o_1)$ then
4:             $r \leftarrow$ true
5:             for all children $B'$ of $B$ do
6:                 $r = r$ and SOLVE($T, f, B', o_1$)
7:             if $r$ then return true
8:     return false
```

Now, we show the correctness and time complexity of Algorithm 1. The time complexity $2^{\mathcal{O}(tw \log tw)}(n + m)$ where n is the number of vertices of ED-graph and m the number of conditions, implies the theorem.

Correctness: We argue the correctness from the bottom up. Suppose that the call SOLVE(T, f, B, o) on leaf bag B returns true. All conditions associated with the leaf B are fulfilled (from Line 3).

Moreover, if there are vertices $X \subseteq B$ that need to satisfy some condition c not associated with B, then let B' be a highest predecessor of B such that $X \subseteq B'$. From the construction of dependency oracle, condition c is associated with B'. Since the calls only extend the ordering, we know that if SOLVE(T, f, B, o) is called, then in the call of SOLVE(T, f, B', o'), on line 2 the algorithm already has the same ordering of vertices in X as in o. Therefore the condition is verified.

If the bag B is not a leaf in the call of SOLVE(T, f, B, o), from induction, we know that all the subtrees under B can be extended such that all conditions are fulfilled. Again, any condition on vertices $X \subseteq B$ is verified either in B directly or in some predecessor.

Time complexity: For given T and f, there are $\mathcal{O}(tw! \cdot n)$ possible calls SOLVE$(T, F, \cdot, \cdot)$ and every condition is associate with one bag. Since $tw! \in 2^{\mathcal{O}(tw \log tw)}$, we have the time complexity $2^{\mathcal{O}(tw \log tw)}(n + m)$. ☐

Acknowledgments

Research supported by the Austrian Science Fund (FWF), project I 5025-N (DELTA), 2020-2024. Esra Ceylan's research was supported by FFG, FEMtech Praktika für Studentinnen. Jakub Svoboda and Krishnendu Chatterjee were supported by the European Research Council (ERC) CoG 863818 (ForM-SMArt).

References

[1] Saeed Akhoondian Amiri, Szymon Dudycz, Stefan Schmid, and Sebastian Wiederrecht. Congestion-free rerouting of flows on dags. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[2] Saeed Akhoondian Amiri, Szymon Dudycz, Mahmoud Parham, Stefan Schmid, and Sebastian Wiederrecht. On polynomial-time congestion-free software-defined network updates. In *2019 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2019.

[3] Saeed Akhoondian Amiri, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Transiently consistent sdn updates: Being greedy is hard. In *23rd International Colloquium on Structural Information and Communication Complexity, SIROCCO*, 2016.

[4] Sebastian Brandt, Klaus-Tycho Förster, and Roger Wattenhofer. On Consistent Migration of Flows in SDNs. In *Proc. 36th IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[5] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. 34th IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[6] Luis Cereceda, Jan Van Den Heuvel, and Matthew Johnson. Finding paths between 3-colorings. *Journal of graph theory*, 67(1):69–82, 2011.

[7] Pavol Cerný, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. Optimal consistent network updates in polynomial time. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 114–128, 2016.

[8] Reinhard Diestel. *Graph Theory*. Springer Berlin Heidelberg, New York, NY, 2017.

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

[10] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. Survey of consistent network updates. In *ArXiv Technical Report*, 2016.

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

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

[13] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Mathematical Sciences Series. W. H. Freeman and Company, 1979.

[14] Parikshit Gopalan, Phokion G Kolaitis, Elitza Maneva, and Christos H Papadimitriou. The connectivity of boolean satisfiability: computational and structural dichotomies. *SIAM Journal on Computing*, 38(6):2330–2355, 2009.

[15] Robert A Hearn and Erik D Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, 2005.

[16] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26, 2013.

[17] Jingyu Hua, Xin Ge, and Sheng Zhong. FOUM: A Flow-Ordered Consistent Update Mechanism for Software-Defined Networking in Adversarial Settings. In *Proc. IEEE INFOCOM*, 2016.

[18] Takehiro Ito, Erik Demaine, Nicholas Harvey, Christos Papadimitriou, Martha Sideri, Ryuhei Uehara, and Yushi Uno. On the complexity of reconfiguration problems. *Algorithms and Computation*, pages 28–39, 2008.

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

[20] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. Adaptable and data-driven softwarized networks: Review, opportunities, and challenges. *Proceedings of the IEEE*, 107(4):711–731, 2019.

[21] Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 184–192. IEEE, 2021.

[22] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about sdn flow tables. In *Proc. Passive and Active Measurement (PAM)*, pages 347–359. Springer, 2015.

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

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

[25] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM PODC*, 2015.

[26] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2014.

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

[28] Tal Mizrahi and Yoram Moses. Time-based updates in software defined networks. In *Proc. ACM HotSDN*, pages 163–164, 2013.

[29] Tal Mizrahi, Ori Rottenstreich, and Yoram Moses. Timeflip: Scheduling network updates with timestamp-based tcam ranges. In *Proc. IEEE INFOCOM*, pages 2551–2559, 2015.

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

[31] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. ACM HotNets*, 2011.

[32] Jan van den Heuvel. The complexity of change. *Surveys in combinatorics*, 409(2013):127–160, 2013.

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

[34] X. Jin et al. Dynamic scheduling of network updates. In *Proc. ACM SIGCOMM*, 2014.

[35] Jiaqi Zheng, Hong Xu, Guihai Chen, and Haipeng Dai. Minimizing transient congestion during network update in data centers. In *Proc. 23rd IEEE International Conference on Network Protocols (ICNP)*, 2015.