Congestion-Free Rerouting of Flows on DAGs*

Saeed Akhoondian Amiri† Szymon Dudycz‡
Stefan Schmid§ Sebastian Wiederrecht¶

Changing a given configuration in a graph into another one is known as a reconfiguration problem. Such problems have recently received much interest in the context of algorithmic graph theory. We initiate the theoretical study of the following reconfiguration problem: How to reroute k unsplittable flows of a certain demand in a capacitated network from their current paths to their respective new paths, in a congestion-free manner? This problem finds immediate applications, e.g., in traffic engineering in computer networks. We show that the problem is generally NP-hard already for $k = 2$ flows, which motivates us to study rerouting on a most basic class of flow graphs, namely DAGs. Interestingly, we find that for general k, deciding whether an unsplittable multi-commodity flow rerouting schedule exists, is NP-hard even on DAGs. Both NP-hardness proofs are non-trivial. Our main contribution is a polynomial-time (fixed parameter tractable) algorithm to solve the route update problem for a bounded number of flows on DAGs. At the heart of our algorithm lies a novel decomposition of the flow network that allows us to express and resolve reconfiguration dependencies among flows.

1 Introduction

Reconfiguration problems are combinatorial problems which ask for a transformation of one configuration into another one, subject to some (reconfiguration) rules. Reconfiguration problems are fundamental and have been studied in many contexts, including puzzles and games (such as Rubik's cube) [40], satisfiability [19], independent sets [20], vertex coloring [10], or matroid bases [23], to just name a few.

Reconfiguration problems also naturally arise in the context of networking applications and routing. For example, a fundamental problem in computer networking regards the question of how to reroute traffic from the current path p_1 to a given new path p_2, by changing the forwarding rules at routers (the *vertices*) one-by-one, while maintaining certain properties *during* the reconfiguration (e.g., short path lengths [7]). Route reconfigurations (or *updates*) are frequent

*The research of Saeed Akhoondian Amiri and Sebastian Wiederrecht has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC consolidator grant DISTRUCT, agreement No 648527). Stefan Schmid is supported by the Danish VILLUM foundation project *ReNet*.

†TU Berlin, DE saeed.amiri@tu-berlin.de
‡University of Wroclaw, PL szymon.dudycz@gmail.com
§Aalborg University, DK schmiste@cs.aau.dk
¶TU Berlin, DE sebastian.wiederrecht@tu-berlin.de

in computer networks: paths are changed, e.g., to account for changes in the security policies, in response to new route advertisements, during maintenance (e.g., replacing a router), to support the migration of virtual machines, etc. [16].

This paper initiates the study of a basic *multi-commodity flow rerouting problem*: how to reroute a set of *unsplittable flows* (with certain bandwidth demands) in a capacitated network, from their current paths to their respective new paths *in a congestion-free manner*. The problem finds immediate applications in traffic engineering [4], whose main objective is to avoid network congestion. Interestingly, while congestion-aware routing and traffic engineering problems have been studied intensively in the past [1, 12, 13, 15, 25, 26, 28, 38], surprisingly little is known today about the problem of how to reconfigure resp. *update* the routes of flows. Only recently, due to the advent of Software-Defined Networks (SDNs), the problem has received much attention in the networking community [3, 8, 17, 31].

Figure 1 presents a simple example of the consistent rerouting problem considered in this paper, for just a *single* flow: the flow needs to be rerouted from the solid path to the dashed path, by changing the forwarding links at routers one-by-one. The example illustrates a problem that might arise from updating the vertices in an invalid order: if vertex v_2 is updated first, a forwarding loop is introduced: the transient flow from s to t becomes invalid. Thus, router updates need to be scheduled intelligently over time: A feasible sequence of updates for this example is given in Figure 2. Note that the example is kept simple intentionally: when moving from a single flow to multiple flows, additional challenges are introduced, as the flows may compete for bandwidth and hence interfere. We will later discuss a more detailed example, demonstrating a congestion-free update schedule for multiple flows.

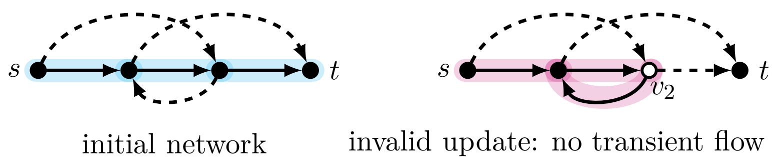

Figure 1: *Example:* We are given an initial network consisting of exactly one active flow F^o (solid edges) and the inactive edges (i.e., inactive forwarding rules) of the new flow F^u to which we want to reroute (dashed edges). Together we call the two flows an (update) pair $P = (F^o, F^u)$. Updating the outgoing edges of a vertex means activating all previously inactive outgoing edges of F^u, and deactivating all other edges of the old flow F^o. Initially, the blue flow is a valid (transient) (s, t)-flow. If the update of vertex v_2 takes effect first, an invalid (not transient) flow is introduced (in pink): traffic is forwarded in a loop, hence (temporarily) invalidating the path from s to t.

Contributions. This paper initiates the algorithmic study of a fundamental unsplittable multicommodity flow rerouting problem. We present a rigorous formal model and show that the problem of rerouting flows in a congestion-free manner is NP-hard already for two flows on general graphs. This motivates us to focus on a most fundamental type of flow graphs, namely the DAG. The main results presented in this paper are the following:

1. Deciding whether a consistent network update schedule exists in general graphs is NP-hard, already for 2 flows.

2. For general k, deciding whether a feasible schedule exists is NP-hard even on loop-free networks (i.e., DAGs).

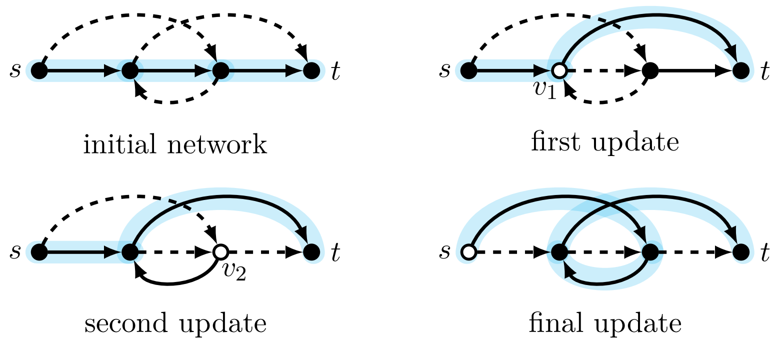

Figure 2: *Example:* We revisit the network of Figure 1 and reroute from F^o to F^u without interrupting the connection between s and t along a unique (transient) path (in blue). To avoid the problem seen in Figure 1, we first update the vertex v_2 in order to establish a shorter connection from s to t. Once this update has been performed, the update of v_2 can be performed without creating a loop. Finally, by updating s, we complete the rerouting.

3. For constant k, we present an elegant linear-time (fixed parameter tractable) algorithm which (deterministically) finds a feasible update schedule on DAGs in time and space $2^{O(k \log k)}O(|G|)$, whenever such a consistent update schedule exists.

Against the backdrop that the problem of *routing* disjoint paths on DAGs is known to be $W[1]$-hard [39] and finding routes *subject to congestion* even harder [1], the finding that the multicommodity flow *rerouting* problem is fixed parameter tractable on DAGs is intriguing.

Technical Novelty. Our algorithm is based on a novel decomposition of the flow graph into so-called *blocks*. This block decomposition allows us to express dependencies between flows. In principle, up to k flows (of unit capacity) can share a physical link of capacity k, and hence, dependencies arise not between pairs but between entire *subsets* of flows along the paths, potentially rendering the problem combinatorial: For every given node, there are up to $k!$ possible flow update orders, leading to a brute force complexity of $O(k!^{|G|})$. However, using a sequence of lemmas, we (1) leverage our block decomposition approach, (2) observe that many of dependencies are redundant, and (3) linearize dependencies, to eventually construct a polynomial-sized graph: this graph has the property that its independent sets characterize dependencies of the block decomposition. We show that this graph is of *bounded path-width*, allowing us to efficiently compute independent sets (using standard dynamic programming), and eventually, construct a feasible update schedule. Overall, this results in an algorithm with linear time complexity in the graph size $|G| = |V(G)| + |E(G)|$.

In addition to our algorithmic contributions, we present rigorous NP-hardness proofs which are based on non-trivial insights into the flow rerouting problem.

2 Model and Definitions

Our problem can be described in terms of edge capacitated directed graphs. In what follows, we will assume basic familiarity with directed graphs and we refer the reader to [5] for more background. We denote a directed edge e with head v and tail u by $e = (u, v)$. For an undirected edge e between vertices u, v, we write $e = \{u, v\}$; u, v are called endpoints of e.

A **flow network** is a directed uncapacitated graph $G = (V, E, s, t, c)$, where s is the *source*, t the *terminal*, V is the set of vertices with $s, t \in V$, $E \subseteq V \times V$ is a set of ordered pairs known as edges, and $c\colon E \to \mathbb{N}$ a capacity function assigning a capacity $c(e)$ to every edge $e \in E$.

Our problem, as described above is a multi-commodity flow problem and thus may have *multiple* source-terminal pairs. To simplify the notation but without loss of generality, in what follows, we define flow networks to have exactly one source and one terminal. In fact, we can model any number of different sources and terminals by adding one super source with edges of unlimited capacity to all original sources, and one super terminal with edges of unlimited capacity leading there from all original terminals.

An (s, t)-*flow* F of capacity $d \in \mathbb{N}$ is a *directed path* from s to t in a flow network such that $d \leq c(e)$ for all $e \in E(F)$. Given a family $\mathcal{F}$ of (s, t)-flows $F_1, \ldots, F_k$ with demands $d_1, \ldots, d_k$ respectively, we call $\mathcal{F}$ a **valid flow set**, or simply **valid**, if $c(e) \geq \sum_{i\colon e \in E(F_i)} d_i$.

Recall that we consider the problem of how to reroute a current (old) flow to a new (update) flow, and hence we will consider such flows in "update pairs":

An **update flow pair** $P = (F^o, F^u)$ consists of two (s, t)-flows F^o, the *old flow*, and F^u, the *update flow*, each of demand d.

A graph $G = (V, E, \mathcal{P}, s, t, c)$, where (V, E, s, t, c) is a flow network, and $\mathcal{P} = \{P_1, \ldots, P_k\}$ with $P_i = (F_i^o, F_i^u)$, a family of update flow pairs of demand d_i, $V = \bigcup_{i \in [k]} V(F_i^o \cup F_i^u)$ and $E = \bigcup_{i \in [k]} E(F_i^o \cup F_i^u)$, is called **update flow network** if the two families $\mathcal{P}^o = \{F_1^o, \ldots, F_k^o\}$ and $\mathcal{P}^u = \{F_1^u, \ldots, F_k^u\}$ are valid. For an illustration, recall the initial network in Figure 2: The old flow is presented as the directed path made of solid edges and the new one is represented by the dashed edges.

Given an update flow network $G = (V, E, \mathcal{P}, s, t, c)$, an **update** is a pair $\mu = (v, P) \in V \times \mathcal{P}$. An update (v, P) with $P = (F^o, F^u)$ is *resolved* by deactivating all outgoing edges of F^o incident to v and activating all of its outgoing edges of F^u. Note that at all times, there is at most one outgoing and at most one incoming edge, for any flow at a given vertex. So the deactivated edges of F^o can no longer be used by the flow pair P (but now the newly activated edges of F^u can).

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

As an illustration, after the second update in Figure 2, one of the original solid edges is still not deactivated. However, already two of the new edges have become solid (i.e., active). So in the picture of the second update, the set $U = \{(v_1, P), (v_2, P)\}$ has been resolved.

We are now able to determine, for a given set of updates, which edges we can and which edges we cannot use for our 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.

The flow pair P is called **transient** for some set of updates $U \subseteq V \times \mathcal{P}$, if $G(P, U)$ contains a unique valid (s, t)-flow $T_{P,U}$.

If there is a family $\mathcal{P} = \{P_1, \ldots P_k\}$ of update flow pairs with demands $d_1, \ldots, d_k$ respectively, we call $\mathcal{P}$ a **transient family** for a set of updates $U \subseteq V \times \mathcal{P}$, if and only if every $P \in \mathcal{P}$ is transient for U. The family of transient flows after all updates in U are resolved is denoted by $\mathcal{T}_{\mathcal{P},\mathcal{U}} = \{T_{P_1,U}, \ldots, T_{P_k,U}\}$.

We again refer to Figure 2. In each of the different states, the transient flow is depicted as the light blue line connecting s to t and covering only solid (i.e., active) edges.

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

Definition 2.1 (Consistency Rule) Let σ be an update sequence. We require that for any $i \in [|V \times \mathcal{P}|]$, there is a family of transient flow pairs $\mathcal{T}_{\mathcal{P},\mathcal{U}_i}$.

To ease the notation, we will denote an update sequence $(\sigma)_{i\in[|V\times\mathcal{P}|]}$ simply by σ and for any update (u, P) we write $\sigma(u, P)$ for the the position i of (u, P) within σ. An update sequence is **valid**, if every set U_i, $i \in [|V \times \mathcal{P}|]$, obeys the consistency rule.

We note that this consistency rule models and consolidates the fundamental properties usually studied in the literature, such as congestion-freedom [8] and loop-freedom [31].

Note that we do not forbid edges $e \in E(F_i^o \cap F_i^u)$ and we never activate or deactivate such an edge. Starting with an initial update flow network, these edges will be active and remain so until all updates are resolved. Hence there are vertices $v \in V$ with either no outgoing edge for a given flow pair F at all; or with an outgoing edge which however is used by both the old and the update flow of F. Such updates do not have any impact on the actual problem since they never affect a transient flow. Hence they can always be scheduled in the first round, and thus w.l.o.g. we ignore them in the following.

Definition 2.2 (*k*-Network Flow Update Problem) Given an update flow network G with k update flow pairs, is there a feasible update sequence σ?

3 NP-Hardness of 2-Flow Update in General Graphs

It is easy to see that for an update flow network with a single flow pair, feasibility is always guaranteed. However, it turns out that for two flows, the problem becomes hard in general.

Theorem 3.1 *Deciding whether a feasible network update schedule exists is NP-hard already for $k = 2$ flows.*

The proof is by reduction from 3-SAT. In what follows let C be any 3-SAT formula with n variables and m clauses. We will denote the variables as $X_1, \ldots, X_n$ and the clauses as $C_1, \ldots, C_m$. The resulting update flow network will be denoted as $G(C)$. Furthermore, we will assume that the variables are ordered by their indices and their appearance in each clause respects this order.

We will create 2 update flow pairs, a blue one $B = (B^o, B^u)$ and a red one $R = (R^o, R^u)$, both of demand 1. The pair B will contain gadgets corresponding to the variables. The order in which the edges of each of those gadgets are updated will correspond to assigning a value to the variable. The pair R on the other hand will contain gadgets representing the clauses: they will have edges that are "blocked" by the variable edges of B. Therefore, we will need to update B to enable the updates of R.

We proceed by giving a precise construction of the update flow network $G(C)$. In the following, the capacities of all edges will be 1. Since we are working with just two flows and each of those flows contains many gadgets, we give the construction of the two update flow pairs in terms of their gadgets.

1. **Clause Gadgets:** For every $i \in [m]$, we introduce eight vertices $u_1^i, u_2^i, \ldots, u_8^i$ corresponding to the clause C_i. The edges (u_j^i, u_{j+1}^i) with $j \in [7]$ are added to R^o while the edges $(u_{j'}^i, u_{j'+5}^i)$ for $j' \in \{1, 2, 3\}$ and $(u_{j'}^i, u_{j'-4}^i)$ for $j' \in \{6, 7\}$ are added to R^u.

2. **Variable Gadgets:** For every $j \in [n]$, we introduce four vertices: $v_1^j, \dots, v_4^j$. Let $P_j = \left\{p_1^j, \dots, p_{k_j}^j\right\}$ denote the set of indices of the clauses containing the literal x_j and $\overline{P}_j = \left\{\overline{p}_1^j, \dots, \overline{p}_{k'_j}^j\right\}$ the set of indices of the clauses containing the literal $\overline{x}_j$. Furthermore, let $\pi(i, j)$ denote the position of x_j in the clause C_i, $i \in P_j$. Similarly, $\overline{\pi}(i', j)$ denotes the position of $\overline{x_j}$ in $C_{i'}$ where $i' \in \overline{P}_j$.

   To B^o we now add the following edges for every $j \in [n]$:

   i) $(u^i_{\pi(i,j)}, u^i_{\pi(i,j)+5})$, for $i \in P_j$ (these edges are shared with R^u),

   ii) $(u^i_{\pi(i,j)+5}, u^i_{\pi(i+1,j)})$, for $i \in P_j, i \neq p^j_{k_j}$,

   iii) $(v^j_1, u^{p^j_1}_{\pi(p^j_1,j)})$ and $(u^{p^j_{k_j}}_{\pi(p^j_{k_j},j)+5}, v^j_2)$,

   iv) $(u^i_{\overline{\pi}(i,j)}, u^i_{\overline{\pi}(i,j)+5})$, for $i \in \overline{P}_j$,

   v) $(u^{\overline{p}^j_i}_{\overline{\pi}(\overline{p}^j_i,j)+5}, u^{\overline{p}^j_{i+1}}_{\overline{\pi}(\overline{p}^j_{i+1},j)})$, for $i \in [\left|\overline{P}_j\right| - 1]$,

   vi) $(v^j_3, u^{\overline{p}^j_1}_{\overline{\pi}(p^j_1,j)})$ and $(u^{\overline{p}^j_{\bar{k}_j}}_{\overline{\pi}(\overline{p}^j_{\bar{k}_j},j)+5}, v^j_4)$, and

   vii) (v^j_2, v^j_3).

   On the other hand, B^u will contain the edges (v^j_1, v^j_3), (v^j_3, v^j_2) and (v^j_2, v^j_4).

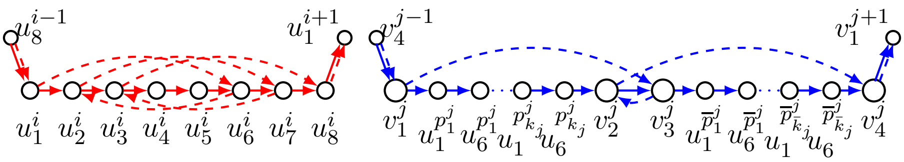

Figure 3: *Examples:* A clause gadget is shown in red, the R^o edges are depicted as a solid line, and the dashed lines belong to R^u. The variable gadget is shown in blue. Again, solid lines indicate the old flow and dashed lines the update flow.

3. **Blocking Edges:** The goal is to block the updates (v^j_3, B) for every $j \in [n]$ until all clauses are satisfied. To do this, we introduce 4 additional vertices w_1, w_2, z_1 and z_2. Then for R^o, we introduce the following edges:

   i) (v^j_3, v^j_2) for $j \in [n]$,

   ii) (v^j_2, v^{j+1}_3) for $j \in [n-1]$, and

   iii) (z_1, v^j_3) and (v^n_2, z_2),

   while R^u contains the edges (z_1), (w_1, w_2) and (w_2, z_2).

   In a similar fashion, B^o contains the edge (w_1, w_2). For B^u, we introduce the following edges:

   i) (u^i_4, u^i_5) for $i \in [m]$,

   ii) (u^i_5, u^{i+1}_4) for $i \in [m-1]$, and

   iii) (w_1, u^1_4) and (u^m_5, w_2).

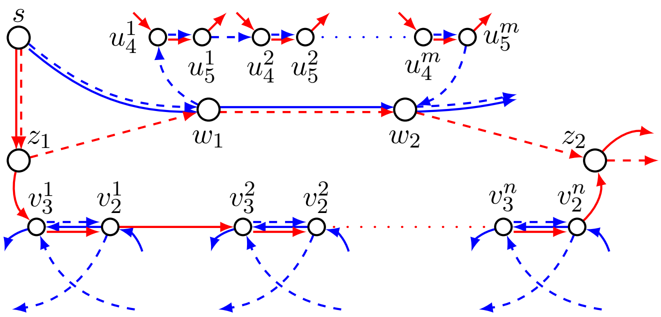

Figure 4: The gadget for blocking the update (v_3^j, B) for all $j \in [n]$. Again dashed edges correspond to the update flows and solid ones to the old flows.

4. **Source and Terminal.** Finally, to complete the graph, we introduce a source s and a terminal t.

   For both, R^o and R^u we introduce the following edges:

   i) (s, z_1) and (z_2, u_1^1),

   ii) (u_8^i, u_1^{i+1}) for $i \in [m-1]$, and

   iii) (u_8^m, t).

   And for B^o and B^u we complete the flows with the following edges:

   i) (s, w_1) and (w_2, v_1^1),

   ii) (v_4^j, v_1^{j+1}) for $j \in [n-1]$, and

   iii) (v_4^n, t).

Lemma 3.2 *Given any valid update sequence σ for the above constructed update flow network $G(C)$, the following conditions hold for every $r < \sigma(w_1, B)$.*

1. $r < \sigma(z_1, R)$

2. *For any $j \in [n]$, v_1^j is a vertex of the transient network flow T_{B,U_r} and $r < \sigma(v_3^j, B)$.*

3. *Let $j \in [n]$ and P_j and $\overline{P}_j$ be the index sets of the clauses containing the corresponding literals x_j and $\overline{x}_j$. Then T_{B,U_r} contains all edges of the form $(u^i_{\pi(i,j)}, u^i_{\pi(i,j)+5})$ for $i \in P_i$, or all the edges $(u^i_{\overline{\pi}(i,j)}, u^i_{\overline{\pi}(i,j)+5})$ for $i \in \overline{P}_j$ (or both).*

4. *The vertex z_1 and the u_1^i, for all $i \in [m]$, are contained in T_{R,U_r}.*

Proof. 1. Suppose $\sigma(z_1, R) \le r$, then there is a step $r' \ge r$ such that (w_1, B) is not in $U_{r'}$, but (z_1, R) is. If $\sigma(w_1, R) \le r'$, $T_{R,U_{r'}}$ and $T_{B,U_{r'}}$ pass through (w_1, w_2) violating the capacity of 1, otherwise there is no path $T_{R,U}$ in $G(R, U)$.

2. The first assertion is trivially true, since the edges (w_2, v_1^1) and (v_4^j, v_1^{j+1}) for $j \in [n-1]$ belong to both B^o and B^u, hence T_{B,U_r} has to always contain these edges. From Property 1 we know, that T_{R,U_r} has to contain the z_1-z_2-subpath of R^o and thus T_{R,U_r} fills the capacity of the edges (v_3^j, v_2^j) for all $j \in [n]$: hence resolving (v_3^j, B) is impossible for all $j \in [n]$.

3. Let $j \in [n]$. By Property 2, v_1^j is contained in T_{R,U_r}, but $\sigma(v_3^j, B) > r$. Hence, if $\sigma(v_1^j, B) \le r$, then T_{B,U_r} traverses directly from v_1^j to v_3^j and then follows along B^o to v_4. Otherwise it follows along B^o from v_1^j to v_3^j. In both cases we are done.

4. This is again trivially true, since the edges (s, z_1) and (u_8^i, u_1^{i+1}) for $i \in [m-1]$ are contained in both R^o and R^u: thus they always have to be part of T_{R,U_r}. □

Proof. (Proof of Theorem 3.1) Now we are ready to finish the proof of Theorem 3.1. First we will show that if C is satisfiable, then there is a feasible order of updates for $G(C)$. Let σ be an assignment satisfying C. Then the update order for $G(C)$ is as follows. For each item i we define r_i^f to be the position of the first update defined by i and r_i^l to be the position of its last update:

1. For each $j \in [n]$, if $\sigma(X_j) = 1$ then update v_1^j. Otherwise update v_2^j.
2. For each $i \in [m]$, at least one of edges $(u_1^i, u_6^i), (u_2^i, u_7^i), (u_3^i, u_8^i)$ is no longer used by $T_{B,U_{r_2^f - 1}}$. Therefore the corresponding update of R can be resolved (this follows from σ being a satisfying assignment).
3. For each $i \in [m]$, (u_4^i, u_5^i) is no longer used by $T_{R,U_{r_3^f - 1}}$. Therefore we can resolve to blue updates along the w_1-w_2-subpath of B^u excluding (w_1, B).
4. Resolve (w_1, B).
5. Resolve (w_1, R) and (w_2, R). (Note that now all conflicts between B and R have been resolved and we can finish the updates. We will now leave the state described in Lemma 3.2.)
6. Resolve (z_1, R).
7. For each $j \in [n]$, v_k^j has already been updated for exactly one $k \in \{1, 2\}$. If $k = 1$, resolve all updates of B along the $u_1^{p_1^j}$-$u_6^{p_{k_j}^j}$-subpath of B^o together with (v_2^j, B). Otherwise resolve (v_3^j, B) together with all updates of B along the $u_1^{\overline{p}_1^j}$-$u_6^{\overline{p}_{k_j}^j}$-subpath of B^o.
8. Resolve the remaining updates of B.
9. Resolve all updates of R along the v_3^1-v_2^n-subpath of R^o and for each $i \in [m]$ resolve (u_1^i, R), (u_2^i, R) and (u_3^i, R).
10. Resolve the remaining updates of R.

Now let us assume that there is a feasible update sequence σ for $G(C)$. We will show that C is satisfiable by constructing an assignment σ.

Let us consider the steps $r < \min\{\sigma(w_1, R), \sigma(w_1, B)\}$. Then we will use Condition 3 of Lemma 3.2 to assign values to variables in the following way. Let $j \in [n]$, if T_{B,U_r} does not use the edges $(u_{\pi(h,j)}^h, u_{\pi(h,j)+5}^h)$ for all $h \in P_j$ (or equivalently if v_1^j is updated) we set $\sigma(X_j) := 1$. Otherwise we set $\sigma(x) := 0$.

Now we will show that assignment σ satisfies C. First let us notice that because we can resolve (w_1, B), none of edges (u_4^i, u_5^i), for any $i \in [m]$, can be used by $T_{B,U_{\sigma(w_1,B)}}$ in $\sigma(w_1, B)$.

Hence, from Condition 4 of Lemma 3.2, we know that all vertices u_1^i, for any $i \in [n]$, and the vertex z_1, are contained in $T_{R,U_{\sigma(w_1,B)}}$.

Let us consider any clause C_i, $i \in [m]$. The transient network flow T_{R,U_r} cannot go from u_1^i to u_1^{i+1} along R^o: this would mean that edge (u_4^i, u_5^i) cannot be used by T_{B,U_r}. Therefore, for at least one of the edges (u_1^i, u_6^i), (u_2^i, u_7^i) and (u_3^i, u_8^i), the corresponding blue update has already been resolved. This implies that there is some variable X_j, $j \in [n]$, that appears in C_i, such that, in the gadget for X_j, T_{B,U_r} skips u_h^i, for some $h \in \{1, 2, 3\}$. This vertex is between v_1^j and v_2^j, if C_i contains literal x_j. In that case, we set $\sigma(X_j) := 1$, so C_i is satisfied. Otherwise C_i contains literal $\bar{x}_j$ and we assign $\sigma(X_j) := 0$, so C_i is also satisfied. □

4 Rerouting flows in DAGs

In this section we consider the flow rerouting problem when the underlying flow graph is acyclic. In the remainder of this work we will always consider our update flow network to be acyclic. This leads to an important substructure in the flow pairs: the blocks. These blocks will play a major role in both the hardness proof and the algorithm.

Let $G = (V, E, \mathcal{P}, s, t, c)$ be an acyclic update flow network, i.e., we assume that the graph (V, E) is acyclic. Let $\prec$ be a topological order on the vertices $V = \{v_1, \ldots, v_n\}$. Let $P_i = (F_i^o, F_i^u)$ be an update flow pair of demand d and let $v_1^i, \ldots, v_{\ell_i^o}^i$ be the induced topological order on the vertices of F_i^o; analogously, let $u_1^i, \ldots, v_{\ell_i^u}^i$ be the order on F_i^u. Furthermore, let $V(F_i^o) \cap V(F_i^u) = \left\{z_1^i, \ldots, z_{k_i}^i\right\}$ be ordered by $\prec$ as well.

The subgraph of $F_i^o \cup F_i^u$ induced by the set $\left\{v \in V(F_i^o \cup F_i^u) \mid z_j^i \prec v \prec z_{j+1}^i\right\}$, $j \in [k_i - 1]$, is called the jth *block* of the update flow pair F_i, or simply the jth i*-block*. We will denote this block by b_j^i.

For a block b, we define $\mathscr{S}(b)$ to be the *start of the block*, i.e., the smallest vertex w.r.t. $\prec$; similarly, $\mathscr{E}(b)$ is the *end of the block*: the largest vertex w.r.t. $\prec$.

Let $G = (V, E, \mathcal{P}, s, t, c)$ be an update flow network with $\mathcal{P} = \{P_1, \ldots, P_k\}$ and let $\mathcal{B}$ be the set of its blocks. We define a binary relation $<$ between two blocks as follows. For two blocks $b_1, b_2 \in \mathcal{B}$, where b_1 is an i-block and b_2 a j-block, $i, j \in [k]$, we say $b_1 < b_2$ (b_1 *is smaller than* b_2) if one of the following holds.

i $\mathscr{S}(b_1) \prec \mathscr{S}(b_2)$,

ii if $\mathscr{S}(b_1) = \mathscr{S}(b_2)$ then $b_1 < b_2$, if $\mathscr{E}(b_1) \prec \mathscr{E}(b_2)$,

iii if $\mathscr{S}(b_1) = \mathscr{S}(b_2)$ and $\mathscr{E}(b_1) = \mathscr{E}(b_2)$ then $b_1 < b_2$, if $i < j$.

Let b be an i-block and P_i the corresponding update flow pair. For a feasible update sequence σ, we will denote the round $\sigma(\mathscr{S}(b), P_i)$ by $\sigma(b)$. We say that i-block b is *updated*, if all edges in $b \cap F_i^u$ are active and all edges in $b \cap F_i^o \setminus F_i^u$ are inactive. We will make use of a basic, but important observation on the structure of blocks and how they can be updated. This structure is the fundamental idea of the algorithm in the next section since it allows us to consider the update of blocks as a whole instead of updating it vertex by vertex.

Lemma 4.1 *Let b be a block of the flow pair $P = (F^u, F^o)$. Then in a feasible update sequence σ, all vertices (resp. their outgoing edges belonging to P) in $F^u \cap b - \mathscr{S}(b)$ are updated strictly before $\mathscr{S}(b)$. Moreover, all vertices in $b - F^u$ are updated strictly after $\mathscr{S}(b)$ is updated.*

Proof. By F_b^u and F_b^o we denote $F^u \cap b$ and $F^o \cap b$ respectively. For the sake of contradiction, let $U = \{v \in V(G) \mid v \in F_b^u - F_b^o - \mathscr{S}(b), \sigma(v, P) > \sigma(\mathscr{S}(b), P)\}$. Moreover, let v be the vertex of U which is updated the latest and $\sigma(v, P) = \max_{u \in U} \sigma(u, P)$. By our condition, the update of v enables a transient flow along edges in F_b^u. Hence, there now exists an (s,t)-flow through b using only update edges.

No vertex in $F_1 := F_b^o - (F_b^u - \mathscr{S}(b))$ could have been updated before, or simultaneously with v: otherwise, between the time u has been updated and before the update of v, there would not exist a transient flow. But once we update v in round r, there is a transient flow T_{P,U_r} which traverses the vertices in $F_b^o - F_b^u$, and another transient flow T_{P,U_r} traverses $v \notin F_1$: a contradiction. Note that $F_1 \neq \emptyset$. The other direction is obvious: updating any vertex in $(F_c^o \cap b) - F_c^u$ inhibits any transient flow. ☐

Lemma 4.2 *Let G be an update flow network and σ a valid update sequence for G. Then there exists a feasible update sequence σ' which updates every block in consecutive rounds.*

Proof. Let σ be a feasible update sequence with a minimum number of blocks that are not updated in consecutive rounds. Furthermore let b be such a block for the flow pair $P = (F^o, F^u)$. Let r be the step in which $\mathscr{S}(b)$ is updated. Then by Lemma 4.1, all other vertices of $F_c^u \cap b$ have been updated in the previous rounds. Moreover, since they do not carry any flow during these rounds, the edges can all be updated in the steps immediately preceding r in any order. By our assumption, we can update $\mathscr{S}(b)$ in round r, and hence now this is still possible.

As $\mathscr{S}(b)$ is updated in step r, the edges of $F_c^o \cap b$ are not used by $T_{P,U_{r+1}}$ and thus we can deactivate all remaining such edges in the steps starting with $r+1$. This is a contradiction to the choice of σ, and hence there is always a feasible sequence σ' satisfying the requirements of the lemma. ☐

Note that G is acyclic and every flow pair in G forms a single block. Let σ be a feasible update sequence of G. We suppose in σ, every block is updated in consecutive rounds (Lemma 4.2). For a single flow F, we write $\sigma(F)$ for the round where the last edge of F was updated.

4.1 Linear Time Algorithm for Constant Number of Flows on DAGs

In the next section we will see that for an arbitrary number of flows, the congestion-free flow reconfiguration problem is hard, even on DAGs. In this section we show that if the number of flows is a constant k, then a solution can be computed in linear time. More precisely, we describe an algorithm to solve the network update problem on DAGs in time $2^{O(k \log k)} O(|G|)$, for arbitrary k. In the remainder of this section, we assume that every block has at least 3 vertices (otherwise, postponing such block updates will not affect the solution).

We say a block b_1 *touches* a block b_2 (denoted by $b_1 \succ b_2$) if there is a vertex $v \in b_1$ such that $\mathscr{S}(b_2) \prec v \prec \mathscr{E}(b_2)$, or there is a vertex $u \in b_2$ such that $\mathscr{S}(b_1) \prec v \prec \mathscr{E}(b_1)$. If b_1 does not touch b_2, we write $b_1 \not\succ b_2$. Clearly, the relation is symmetric, i.e., if $b_1 \succ b_2$ then $b_2 \succ b_1$.

For some intuition, consider a drawing of G which orders vertices w.r.t. $\prec$ in a line. Project every edge on that line as well. Then two blocks touch each other if they have a common segment on that projection.

Algorithm and Proof Sketch

Before delving into details, we provide the main ideas behind our algorithm. We can think about the update problem on DAGs as follows. Our goal is to compute a feasible update order for the (out-)edges of the graph. There are at most k flows to be updated for each edge, resulting in $k!$ possible orders and hence a brute force complexity of $O(k!^{|G|})$ for the entire problem. We can reduce this complexity by considering blocks instead of edges.

The update of a given i-block b_i might depend on the update of a j-block sharing at least one edge of b_i. These dependencies can be represented as a directed graph. If this graph does not have any directed cycles, it is rather easy to find a feasible update sequence, by iteratively updating sink vertices.

There are several issues here: First of all these dependencies are not straight-forward to define. As we will see later, they may lead to representation graphs of exponential size. In order to control the size we might have to relax our definition of dependency, but this might lead to a not necessarily acyclic graph which will then need further refinement. This refinement is realized by finding a suitable subgraph, which alone is a hard problem in general. To overcome the above problems, we proceed as follows.

Let TouchSeq(b) contain all feasible update sequences for the blocks that touch b: still a (too) large number, but let us consider them for now. For two distinct blocks b, b', we say that two sequences $s \in \text{TouchSeq}(b), s' \in \text{TouchSeq}(b')$ are *consistent*, if the order of any common pair of blocks is the same in both s, s'. It is clear that if for some block b, $\text{TouchSeq}(b) = \emptyset$, there is no feasible update sequence for G: b cannot be updated.

We now consider a graph H whose vertices correspond to elements of TouchSeq(b), for all $b \in \mathcal{B}$. Connect all pairs of vertices originating from the same TouchSeq(b). Connect all pairs of vertices if they correspond to inconsistent elements of different TouchSeq(b). If (and only if) we find an independent set of size $|\mathcal{B}|$ in the resulting graph, the update orders corresponding to those vertices are mutually consistent: we can update the entire network according to those orders. In other words, the update problem can be reduced to finding an independent set in the graph H.

However, there are two main issues with this approach. First, H can be very large. A single TouchSeq(b) can have exponentially many elements. Accordingly, we observe that we can assume a slightly different perspective on our problem: we linearize the lists TouchSeq(b) and define them sequentially, bounding their size by a function of k (the number of flows). The second issue is that finding a maximum independent set in H is hard. The problem is equivalent to finding a clique in the complement of H, a $|\mathcal{B}|$-partite graph where every partition has bounded cardinality. We can prove that for an n-partite graph where every partition has bounded cardinality, finding an n-clique is NP-complete. So, in order to solve the problem, we either should reduce the number of partitions in H (but we cannot) or modify H to some other graph, further reducing the complexity of the problem. We do the latter by trimming H and removing some extra edges, turning the graph into a very simple one: a graph of *bounded path width*. Then, by standard dynamic programming, we find the independent set of size $|\mathcal{B}|$ in the trimmed version of H: this independent set matches the independent set I of size $|\mathcal{B}|$ in H (if it exists). At the end, reconstructing a correct update order sequence from I needs some effort. As we have reduced the size of TouchSeq(b) and while not all possible update orders of all blocks occur, we show that they suffice to cover all possible feasible solutions. We provide a way to construct a valid update order accordingly. With these intuitions in mind, we now present a rigorous analysis. Let $\pi_{S_1} = (a_1, \ldots, a_{\ell_1})$ and $\pi_{S_2} = (a'_1, \ldots, a'_{\ell_2})$ be permutations of sets S_1 and S_2. We define the *core* of π_{S_1} and π_{S_2} as $core(\pi_{S_1}, \pi_{S_2}) := S_1 \cap S_2$. We say that two permutations π_1 and π_2 are *consistent*, $\pi_1 \approx \pi_2$, if there is a permutation π of symbols of $core(\pi_1, \pi_2)$ such that π is a subsequence of both π_1 and π_2.

The **Dependency Graph** is a labelled graph defined recursively as follows. The dependency graph of a single permutation $\pi = (a_1, \ldots, a_\ell)$, denoted by G_π, is a directed path $v_1, \ldots, v_\ell$, and the label of the vertex $v_i \in V(G_\pi)$ is the element a with $\pi(a) = i$. We denote by $\text{Labels}(G_\pi)$

the set of all labels of G_π.

Let G_Π be a dependency graph of the set of permutations Π and $G_{\Pi'}$ the dependency graph of the set Π'. Then, their union (by identifying the same vertices) forms the dependency graph $G_{\Pi\cup\Pi'}$ of the set $\Pi\cup\Pi'$. Note that such a dependency graph is not necessarily acyclic.

We call a permutation π of blocks of a subset $\mathcal{B}'\subseteq\mathcal{B}$ *congestion free*, if the following holds: it is possible to update the blocks in π in the graph $G_\mathcal{B}$ (the graph on the union of blocks in $\mathcal{B}$), in order of their appearance in π, without violating any edge capacities in $G_\mathcal{B}$. Note that we do not respect all conditions of our *Consistency Rule* (definition 2.1) here.

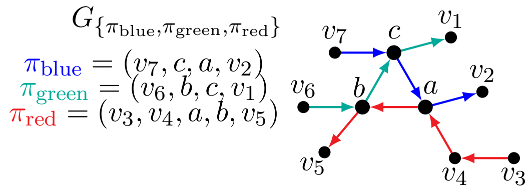

Figure 5: *Example:* The dependency graph of three pairwise consistent permutations π_{blue}, π_{green} and π_{red}. Each pair of those permutation has exactly one vertex in common and with this the cycle (a, b, c) is created. With such cycles being possible a dependency graph does not necessarily contain sink vertices. To get rid of them, we certainly need some more refinements.

In the approach we are taking, one of the main advantages we have is the nice properties of blocks when it comes to updating. The following algorithm formalizes the procedure already described in Lemma 4.2. The correctness follows directly from said lemma. Let $P=(F^o,F^u)$ be a given flow pair.

Algorithm 1. **Update a Free Block** b

1. Resolve (v,P) for all $v\in F^u\cap b-\mathscr{S}(b)$.
2. Resolve $(\mathscr{S}(b),P)$.
3. Resolve (v,P) for all $v\in(b-F^u)$.
4. For any edge in $E(b\cap F^u)$ check whether d_{F^u} together with the other loads on e exceed $c(e)$. If so output: *Fail.*

Lemma 4.3 *Let π be a permutation of the set $\mathcal{B}_1\subseteq\mathcal{B}$. Whether π is congestion free can be determined in time $O(k\cdot|G|)$.*

Proof. In the order of π, perform Algorithm 1. If it fails, i.e., if it violates congestion freedom for some edges, π is not a congestion free permutation. The running time of Algorithm 1 is in $O(|b|)$ for a block b, hence the overall running time is bounded above by:

$$\sum_{b\in\mathcal{B}_1}|b| = \sum_{i=1}^{k}\sum_{\substack{b\in\mathcal{B}_1\\ b\text{ is an }i\text{-block}}}|b|\le k\cdot|G|\,.$$

□

The smaller relation defines a total order on all blocks in G. Let $\mathcal{B}=\{b_1,\ldots,b_{|\mathcal{B}|}\}$ and suppose the order is $b_1<\ldots<b_{|\mathcal{B}|}$.

We define an auxiliary graph H which will help us find a suitable dependency graph for our network. We first provide some high-level definitions relevant to the construction of the graph H only. Exact definitions will follow in the construction of H, and will be used throughout the rest of this section.

Recall that $\mathcal{B}$ is the set of all blocks in G. We define another set of blocks $\mathcal{B}'$ which is initialized as $\mathcal{B}$; the construction of H is iterative, and in each iteration, we eliminate a block from $\mathcal{B}'$. At the end of the construction of H, $\mathcal{B}'$ is empty. For every block $b \in \mathcal{B}'$, we also define the set TouchingBlocks(b) of blocks which touch the block b. Another set which is defined for every block b is the set PermutList(b); this set actually corresponds to a set of vertices, each of which corresponds to a valid congestion free permutation of blocks in TouchingBlocks(b). Clearly if TouchingBlocks(b) does not contain any congestion-free permutation, then PermutList(b) is an empty set. As we already mentioned, every vertex $v \in \text{PermutList}(b)$ comes with a **label** which corresponds to some congestion-free permutation of elements of TouchingBlocks(b). We denote that permutation with Label (v).

Construction of H: We recursively construct a labelled graph H from the blocks of G as follows.

i Set $H := \emptyset$, $\mathcal{B}' := \mathcal{B}$, PermutList $:= \emptyset$.

ii For $i := 1, \ldots, |\mathcal{B}|$ do

1 Let $b := b_{|\mathcal{B}|-i+1}$.
2 Let TouchingBlocks$(b) := \{b'_1, \ldots, b'_t\}$ be the set of blocks in $\mathcal{B}'$ touched by b.
3 Let $\pi := \{\pi_1, \ldots, \pi_\ell\}$ be the set of congestion free permutations of TouchingBlocks(b).
4 Set PermutList$(b) := \emptyset$.
5 For $i \in [\ell]$ create a vertex v_{π_i} with Label$(v_{\pi_i}) = \pi_i$ and set PermutList$(b) :=$ PermutList$(b) \cup v_{\pi_i}$.
6 Set $H := H \cup \text{PermutList}(b)$.
7 Add edges between all pairs of vertices in $H[\text{PermutList}(b)]$.
8 Add an edge between every pair of vertices $v \in H[\text{PermutList}(b)]$ and $u \in V(H) - \text{PermutList}(b)$ if the labels of v and u are inconsistent.
9 Set $\mathcal{B}' := \mathcal{B}' - b$.

Lemma 4.4 *For Item (ii) of the construction of H, $t \leq k$ holds.*

Proof. Suppose for the sake of contradiction that t is bigger than k. So there are j-blocks b, b' (where $b_{|\mathcal{B}|-i+1}$ corresponds to a flow pair different from j) that touch $b_{|\mathcal{B}|-i+1}$. But then one of $\mathscr{S}(b)$ or $\mathscr{S}(b')$ is strictly larger than $\mathscr{S}(b_{|\mathcal{B}|-i+1})$. This contradicts our choice of $b_{|\mathcal{B}|-i+1}$ in that we deleted larger blocks from $\mathcal{B}'$ in Item (ii9). □

Lemma 4.5 (Touching Lemma) *Let $b_{j_1}, b_{j_2}, b_{j_3}$ be three blocks (w.r.t. $<$) where $j_1 < j_2 < j_3$. Let b_z be another block such that $z \notin \{j_1, j_2, j_3\}$. If in the process of constructing H, b_z is in the touch list of both b_{j_1} and b_{j_3}, then it is also in the touch list of b_{j_2}.*

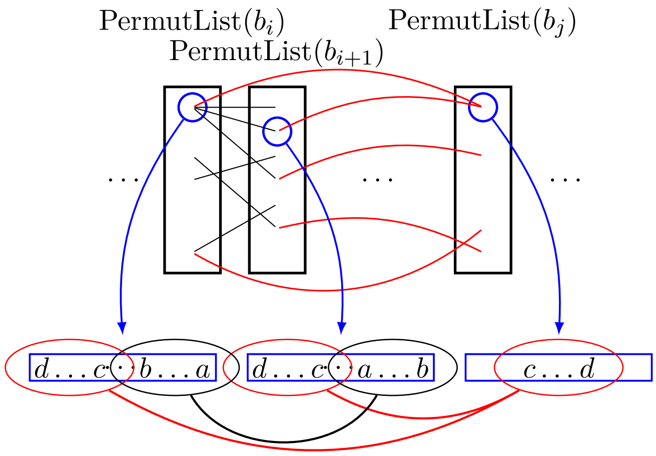

Figure 6: *Example:* The graph H consists of vertex sets PermutList(b_i), $i \in [|\mathcal{B}|]$, where each such partition contains all congestion free sequences of the at most k iteratively chosen touching blocks. In the whole graph, we then create edges between the vertices of two such partitions if and only if the corresponding sequences are inconsistent with each other, as seen in the three highlighted sequences. Later we will distinguish between such edges connecting vertices of neighbouring partitions (w.r.t. the topological order of their corresponding blocks), PermutList(b_i) and PermutList(b_{i+1}), and partitions that are further away, PermutList(b_i) and PermutList(b_j). Edges of the latter type, depicted as red in the figure, are called long edges and will be deleted in the trimming process of H.

Proof. Let us suppose that $\mathscr{S}(b_{j_1}) \neq \mathscr{S}(b_{j_2}) \neq \mathscr{S}(b_{j_3})$. We know that $\mathscr{S}(b_z) \prec \mathscr{S}(b_{j_1})$ as otherwise, in the process of creating H, we eliminate b_z before we process b_{j_1}: it would hence not appear in the touch list of b_{j_1}. As $b_z \succ b_{j_3}$, there is a vertex $v \in b_z$ where $\mathscr{S}(b_{j_3}) \prec v$. But by our choice of elimination order: $\mathscr{S}(b_{j_2}) \prec \mathscr{S}(b_{j_3}) \prec v \prec \mathscr{E}(b_z)$, and on the other hand: $\mathscr{S}(b_z) \prec \mathscr{S}(b_{j_1}) \prec \mathscr{S}(b_{j_2})$. Thus, $\mathscr{S}(b_z) \prec \mathscr{S}(b_{j_2}) \prec \mathscr{E}(b_z)$, and therefore b_z touches b_{j_2}. If some of the start vertices are the same, a similar case distinction applies. □

For an illustration of the property described in the Linear Time Algorithm for Constant Number of Flows on see Figure 7: it refers to the dependency graph of Figure 5. This example also points out the problem with directed cycles in the dependency graph and the property of the Linear Time Algorithm for Constant Number of Flows on DAGs.

We prove some lemmas in regard to the dependency graph of elements of H, to establish the base of the inductive proof for Lemma 4.9.

We begin with a simple observation on the fact that a permutation π induces a total order on the elements of S.

Observation 4.6 *Let π be a permutation of a set S. Then the dependency graph G_π does not contain a cycle.*

Lemma 4.7 *Let π_1, π_2 be permutations of sets S_1, S_2 such that π_1, π_2 are consistent. Then the dependency graph $G_{\pi_1 \cup \pi_2}$ is acyclic.*

Proof. For the sake of contradiction suppose there is a cycle C in $G_{\pi_1 \cup \pi_2}$. By Observation 4.6 this cycle must contain vertices corresponding to elements of both S_1 and S_2. Let a be the

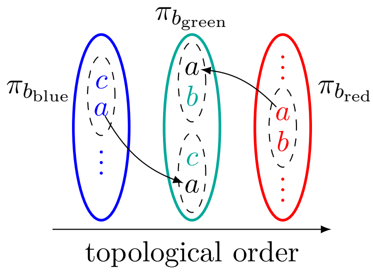

Figure 7: *Example:* Select one of the permutations of length at most k from every PermutList(b). These permutations obey the Linear Time Algorithm for Constant Number of Flows on DAGs. Taking the three permutations from the example in Figure 5, we can see that the Linear Time Algorithm for Constant Number of Flows on DAGs forces a to be in the green permutation as well. Assuming consistency, this would mean a to come *before* b and *after* c. Hence $a <_{\pi_{\text{green}}} b$ and $b <_{\pi_{\text{green}}} a$, a contradiction. So if our permutations are derived from H and are consistent, we will show that cycles cannot occur in their dependency graph.

least element of S_1 with respect to π_1 such that $v_a \in V(C)$. As C is a cycle there is a vertex v_b with $b \in S_1 \cup S_2$ such that the edge (v_b, v_a) is an edge of C. By our choice of a, b is not contained in S_1. Hence, since the edge (v_b, v_a) exists, $a \in S_1 \cap S_2$. Similarly we can consider the least element $c \in S_2$ in C and its predecessor $d \in S_1 \setminus S_2$ along the cycle. Again the edge (v_d, v_c) exists and thus $c \in S_1 \cap S_2$. Now we have $d < a$ in π_2, but $a < d$ in π_1 contradicting the consistency of π_1 and π_2. □

In the next lemma, we need a closure of the dependency graph of permutations which we define as follows.

Definition 4.8 (Permutation Graph Closure) The *Permutation Graph Closure*, or simply *closure*, of a permutation π is the graph G^+_π obtained from taking the transitive closure of G_π, i.e. its vertices and labels are the same as G_π and there is an edge (u, v) in G^+_π if there is a path starting at u and ending at v in G_π. Similarly the *Permutation Graph Closure* of a set of permutations $\Pi = \{\pi_1, \dots, \pi_n\}$ is the graph obtained by taking the union of $G^+_{\pi_i}$'s (for $i \in [n]$) by identifying vertices of the same label.

In the above definition note that if Π is a set of permutations then $G_\Pi \subseteq G^+_\Pi$.

The following lemma generalizes Lemma 4.7 and Observation 4.6 and uses them as the base of its inductive proof.

Lemma 4.9 *Let $I = \{v_{\pi_1}, \dots, v_{\pi_\ell}\}$ be an independent set in H. Then the dependency graph G_Π, for $\Pi = \{\pi_1, \dots, \pi_\ell\}$, is acyclic.*

Proof. Instead of working on G_Π, we can work on its closure G^+_Π as defined above. First we observe that every edge in G_Π also appears in G^+_Π, so if there is a cycle in G_Π, the same cycle exists in G^+_Π.

We prove that there is no cycle in G^+_Π. By Lemma 4.7 and Observation 4.6 there is no cycle of length at most 2 in G^+_Π; otherwise there is a cycle in G_Π which consumes at most two consistent permutations.

For the sake of contradiction, suppose G^+_Π has a cycle and let $C = (a_1, \ldots, a_n) \subseteq G^+_\Pi$ be the shortest cycle in G^+_Π. By Lemma 4.7 and Observation 4.6 we know that $n \geq 3$.

In the following, because we work on a cycle C, whenever we write any index i we consider it w.r.t. its cyclic order on C, in fact $i \mod |C|+1$. So for example, $i = 0$ and $i = n$ are identified as the same indices; similarly for $i = n+1, i = 1$, etc.

Recall the construction of the dependency graph where every vertex $v \in C$ corresponds to some block b_v. In the remainder of this proof we do not distinguish between the vertex v and the block b_v.

Let π_v be the label of a given vertex $v \in I$. For each edge $e = (a_i, a_{i+1}) \in C$, there is a permutation π_{v_i} such that (a_i, a_{i+1}) is a subsequence of π_{v_i} and additionally the vertex v_i is in the set I. So there is a block b^i such that π_{v_i} is a permutation of the set $\text{TouchingBlocks}(b^i)$.

The edge $e = (a_i, a_{i+1})$ is said to *represent* b^i, and we call it the representative of π_{v_i}. For each i we fix one block b^i which is represented by the edge (a_i, a_{i+1}) (note that one edge can represent many blocks, but here we fix one of them). We define the set of those blocks as $B^I = \{b^1, \ldots, b^\ell\}$ and state the following claim.

Claim 1. For every two distinct vertices $a_i, a_j \in C$, either there is no block $b \in B^I$ such that $a_i, a_j \in \text{TouchingBlocks}(b)$ or if $a_i, a_j \in \text{TouchingBlocks}(b)$ then (a_i, a_j) or (a_j, a_i) is an edge in C. Additionally $\left|B^I\right| = |C|$.

Proof. Suppose there is a block $b \in B^I$ such that $a_i, a_j \in \text{TouchingBlocks}(b)$. Then in $E(G^+_\Pi)$ there is an edge $e_1 = (a_i, a_j)$ or $e_2 = (a_j, a_i)$. If either of e_1, e_2 is an edge in C then we are done. Otherwise if $e_1 \in E(G^+_\Pi)$ then the cycle on the vertices $a_1, \ldots, a_i, a_j, \ldots, a_n$ is shorter than C and if $e_2 \in E(G^+_\Pi)$ then the cycle on the vertices $a_i, \ldots, a_j$ is shorter than C. Both cases contradict the assumption that C is the shortest cycle in G^+_Π. For the second part of the claim it is clear that $\left|B^I\right| \leq |C|$, on the other hand if both endpoints of an edge $e = (a_i, a_{i+1}) \in C$ appear in $\text{TouchingBlocks}(b)$ and $\text{TouchingBlocks}(b')$ for two different blocks $b, b' \in B^I$ then, by our choice of the elements of B^I, at least one of them (say b) has a representative $e' \neq e$. But, then there is a vertex $a_j \in V(e')$ such that $a_j \neq a_i, a_j \neq a_{i+1}$. But by the first part this cannot happen, so we have $|C| \leq \left|B^I\right|$ and the second part of the claim follows. ⊣

By the above claim we have $\ell = n$. W.l.o.g. suppose $b^1 < b^2 < \ldots < b^n$. There is an $i \in [n]$ such that (a_{i-1}, a_i) represents b^1, we fix this i.

Claim 2. If (a_{i-1}, a_i) represents b^1 then (a_{i-2}, a_{i-1}) represents b^2.

Proof. By Claim 1 there is a block b^t represented by (a_{i-2}, a_{i-1}). We also have $b^1 < b^2 \leq b^t$ hence by the Linear Time Algorithm for Constant Number of Flows on DAGs, a_{i-1} appears in $\text{TouchingBlocks}(b^2)$. But then by Claim 1 either a_{i+1} is in $\text{TouchingBlocks}(b^2)$ or $a_{i-2} \in \text{TouchingBlocks}(b^2)$, by the former case we have $b^1 = b^2$ which is a contradiction to the assumption that $b^1 < b^2$. In the latter case we have $t = 2$ which proves the claim. ⊣

Similarly we can prove the endpoints of the edges, that have a_i as their head, are in b^2.

Claim 3. If (a_{i-1}, a_i) represents b^1 then (a_i, a_{i+1}) represents b^2.

Proof. By Claim 1 there is a block b^t such that (a_i, a_{i+1}) represents b^t. We also have $b^1 < b^2 \leq b^t$ thus by the Linear Time Algorithm for Constant Number of Flows on DAGs, a_i appears in

TouchingBlocks(b^2). But, then by Claim 1 either a_{i-1} or a_{i+1} is in TouchingBlocks(b^2). In the former case we have $b^1 = b^2$ which is a contradiction to the assumption that $b^1 < b^2$. In the latter case we have $t = 2$ which proves the claim. $\dashv$

By Claims 2 and 3 we have that both (a_{i-2}, a_{i-1}) and (a_i, a_{i+1}) represent b^2 hence by Claim 1 they are the same edge. Thus there is a cycle on the vertices a_{i-1}, a_i in G^+_Π and this gives a cycle in G_Π on at most 2 consistent permutations which is a contradiction according to Lemma 4.7.□

The following lemma establishes the link between independent sets in H and feasible update sequences of the corresponding update flow network G.

Lemma 4.10 *There is a feasible sequence of updates for an update network G on k flow pairs, if and only if there is an independent set of size $|\mathcal{B}|$ in H. Additionally if the independent set $I \subseteq V(H)$ of size $|\mathcal{B}|$ together with its vertex labels are given, then there is an algorithm which can compute a feasible sequence of updates for G in $O(k \cdot |G|)$.*

Proof. First we prove that if there is a sequence of feasible updates σ, then there is an independent set of size $|\mathcal{B}|$ in H. Suppose σ is a feasible sequence of updates of blocks. For a block b, recall that TouchingBlocks$(b) = \{b'_1, \ldots, b'_\ell\}$ is the set of remaining (not yet processed) blocks that touch b. Let π_b be the reverse order of updates of blocks in TouchingBlocks(b) w.r.t. σ. In fact, if σ updates b'_1 first, then b'_2, then $b'_3, \ldots, b'_\ell$, then $\pi_b = b'_\ell \ldots b'_1$.

For every two blocks $b, b' \in I$, we have $\pi_b \approx \pi_{b'}$. From every set of vertices PermutList(b), for $b \in B$, let v^b_i be a vertex such that Label(v^b_i) is a subsequence of π_b. Recall that, the labels of vertices in PermutList(b) are all possible congestion free permutations of blocks that touch b in the remaining set of blocks $\mathcal{B}'$ during the construction of H. So the vertex v^b_i exists. Put v^b_i in I. The labels of every pair of vertices in I are consistent, as their super-sequences were consistent, so I is an independent set and furthermore $|I| = |\mathcal{B}|$.

For the other direction, suppose there is an independent set of vertices I of size $|\mathcal{B}|$ in H. It is clear that for every block $b \in \mathcal{B}$, there is exactly one vertex $v_b \in I \cap \text{PermutList}(b)$.

Let us define the dependency graph of the set of labels (permutations) $\Pi = \{\text{Label}(v_b) \mid b \in \mathcal{B}, v_b \in I\}$ as the dependency graph $D := G_\Pi$. I is an independent set and thus every pair of labels of vertices in I are consistent, hence by Lemma 4.9 we know that D is a DAG, and thus it has a sink vertex.

We update blocks which correspond to sink vertices of D in parallel by applying Algorithm 1 and we remove those vertices from D after they are updated. Then we proceed recursively, until there is no vertex in D. We claim that this gives a feasible sequence of updates for all blocks.

Suppose there is a sink vertex whose corresponding block b cannot be updated. There are two reasons preventing us from updating a block by ignoring the Consistency Rule:

1. Its update stops the flow between some source and terminal. So afterwards there is no transient flow on the active edges.

2. There is an edge $e \in E(b)$ which cannot be activated because this would imply routing along it and produce congestion.

The first will never be the case by definition of Algorithm 1. So suppose there is such an edge e. Edge e cannot be updated because some other blocks are incident to e and currently route flows: updating b would violate a capacity constraint. There may be some blocks which are incident to e but are not updated yet. These blocks would not effect the rest of our reasoning and we restrict ourselves to those blocks which have been updated already by our algorithm.

Otherwise, if there is no such block, the label corresponding to b is an invalid congestion free label. We will denote the set of the blocks preventing the update of e by $\mathcal{B}_e$.

Suppose the blocks in $\mathcal{B}_e$ are updated in the order $b'_1, b'_2, \ldots, b'_\ell$ by the above algorithm. Among $b, b'_1, \ldots, b'_\ell$, there is a block b' which is the largest one (w.r.t. $<$). In the construction of H, we know that $\text{PermutList}(b') \neq \emptyset$, as otherwise I was not of size $|\mathcal{B}|$. Suppose $v \in \text{PermutList}(b') \cap I$. In the iteration where we create $\text{PermutList}(b')$, b' touches all blocks in $\{b, b'_1, \ldots, b'_\ell\}$, hence, in the $\text{Label}(v)$, we have a subsequence $b''_1, \ldots, b''_{\ell+1}$ such that $b''_i \in \{b, b'_1, \ldots, b'_\ell\}$.

We claim that the permutations $\pi_1 = b''_1, \ldots, b''_{\ell+1}$ and $\pi_2 = b'_1, \ldots, b'_\ell, b$ are exactly the same, which would contradict our assumption that e cannot be updated: π_1 is a subsequence of the congestion free permutation $\text{Label}(v)$. Suppose $\pi_1 \neq \pi_2$, then there are two blocks b'''_1, b'''_2 with $\pi_1(b'''_1) < \pi_1(b'''_2)$ and $\pi_2(b'''_2) < \pi_2(b'''_1)$, then $\pi_1 \not\approx \pi_2$. Since both, b'''_2 and b'''_1, will appear in $\text{Label}(v)$, there is a directed path from b'''_2 to b'''_1 in D. Then our algorithm cannot choose b'''_2 as a sink vertex before updating b'''_1: a contradiction.

At the end recall that we used Algorithm 1 as a subroutine and this guarantees the existence of transient flow if we do not violate the congestion of edges, i.e. the algorithm does not return *Fail* at any point. Hence, the sequence of updates we provided by deleting the sink vertices, is a valid sequence of updates if I is an independent set of size $|\mathcal{B}|$.

On the other hand, in the construction of H, all congestion free routings are already given and the runtime of Algorithm 1 is linear in the size of the dependency graph: If I is given, the number of blocks is at most k times larger than the original graph or $|G_\Pi| = O(k \cdot |G|)$; therefore, we can compute the corresponding update sequence in $O(k\,|G|)$ as claimed. □

With Lemma 4.10, the update problem boils down to finding an independent set of size $|\mathcal{B}|$ in H. However, this reduction does not suffice yet to solve our problem in polynomial time, as we will show next.

Finding an independent set of size $|\mathcal{B}|$ in H is equivalent to finding a clique of size $|\mathcal{B}|$ in its complement. The complement of H is a $|\mathcal{B}|$-partite graph where every partition has cardinality $\leq k!$. In general, it is computationally hard to find such a clique. This is shown in the following lemma. Note that the lemma is not required for the analysis of our algorithm, but constitutes an independent result and serves to round off the discussion.

Lemma 4.11 *Finding an m-clique in an m-partite graph, where every partition has cardinality at most 3, is NP-hard.*

PROOF. We provide a polynomial time reduction from 3-SAT. Let $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ be an instance of 3-SAT with n variables $X_1, \ldots, X_n$. We denote positive appearances of X_i as a literal x_i and negative appearance as a literal $\bar{x}_i$ for $i \in [m]$. So we have at most $2n$ different literals $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$. Create an m-partite graph G as follows. Set G to be an empty graph. Let $C_i = \{l_{i_1}, l_{i_2}, l_{i_3}\}$ be a clause for $i \in [m]$, then add vertices $v^i_{l_{i_1}}, v^i_{l_{i_2}}, v^i_{l_{i_3}}$ to G as partition p_i. Note that $l_{i_1} = x_t$ or $l_{i_1} = \bar{x}_t$ for some $t \in [n]$. Add an edge between each pair of vertices v^i_x, u^j_y for $i, j \in [m], i \neq j$ if $x = x_t$ for some $t \in [n]$ and $y \neq \bar{x}_t$ or if $x = \bar{x}_t$ and $y \neq x_t$. It is clear that G now is an m-partite graph with exactly 3 vertices in each partition.

Claim 4. There is a satisfying assignment σ for C if, and only if, there is an m-clique in G. □

Proof. Define a vertex set $K = \emptyset$. Let σ be a satisfying assignment. Then from each clause C_i for $i \in [m]$, there is a literal l_{i_j} which is set to true in σ. We take all vertices of G of the form $v^i_{l_j}$ and add it to K. The subgraph $G[K]$ forms a clique of size m. On the other hand suppose we have an m-clique K_m as a subgraph of G. Then, clearly from each partition p_i, there exists exactly one vertex $v^i_{l_j}$ which is in K_m. We set the literal l_j to true. This gives a valid satisfying assignment for C. ⊣

Now we trim H to avoid the above problem. Again we will use the special properties of the touching relation of blocks. We say that some edge $e \in E(H)$ is *long*, if one end of e is in $\text{PermutList}(b_i)$, and the other in block type $\text{PermutList}(b_j)$ where $j > i + 1$. The *length* of e is $j - i$. *Delete* all long edges from H to obtain the graph R_H. In other words we can construct R_H directly, similar to H, without adding long edges. In the following we first prove that in linear time we can construct the graph R_H. Second we show that if there is an independent set I of size exactly $|\mathcal{B}|$ in R_H then I is also an independent set of H.

Lemma 4.12 *There is an algorithm which computes R_H in time $O((k \cdot k!)^2 \, |G|)$.*

Proof. The algorithm is similar to the construction of H. For completeness we repeat it here and then we prove it takes time proportional to $(k \cdot k!)^2 \, |G|$.

Algorithm 2. **Construction of R_H**

Input: Update Flow Network G

i Set $H := \emptyset$, $\mathcal{B}' := \mathcal{B}$, $\text{PermutList} := \emptyset$.

ii For $i := 1, \ldots, |\mathcal{B}|$ do

1 Let $b := b_{|\mathcal{B}|-i+1}$.

2 Let $\text{TouchingBlocks}(b) := \{b'_1, \ldots, b'_t\}$ be the set of blocks in $\mathcal{B}'$ which touch b.

3 Let $\pi := \{\pi_1, \ldots, \pi_\ell\}$ be the set of congestion free permutations of $\text{TouchingBlocks}(b)$, compute π by the algorithm provided in 4.3.

4 Set $\text{PermutList}(b) := \emptyset$.

5 For $i \in [\ell]$ create a vertex v_{π_i} with $\text{Label}(v_{\pi_i}) = \pi_i$ and set $\text{PermutList}(b) := \text{PermutList}(b) \cup v_{\pi_i}$.

6 Set $H := H \cup \text{PermutList}(b)$.

7 Add edges between all pairs of vertices in $H[\text{PermutList}(b)]$.

8 Add an edge between every pair of vertices $v \in H[\text{PermutList}(b)]$ and $u \in \text{PermutList}(b_{|\mathcal{B}|-i+2})$ if the labels of v and u are inconsistent and if $b_{|\mathcal{B}|-i+2}$ exists.

9 Set $\mathcal{B}' := \mathcal{B}' - b$.

The only difference between the above algorithm and the construction of H is line ii8, where we add at most $O(k!^2)$ edges to the graph. As there are at most $|\mathcal{B}|$ steps in the algorithm, this shows that the size of R_H is at most $O(|\mathcal{B}| \cdot k!^2)$. Moreover, as there are at most $O(k\,|E(G)|)$ blocks in G, the total size of R_H w.r.t. G is at most $O(k \cdot k!^2 \cdot |G|)$. The computations in all other lines except for line ii3 are linear in k, hence we only show that the total amount of computations in line ii3 is in $O(k! \cdot |G|)$. We know that every edge appears in at most k blocks, hence the algorithm provided in Lemma 4.3, for each edge, runs at most k times and as per individual round of that algorithm, takes $O(k \cdot |G|)$. Since there are $k!$ possible permutations for each block, this yields a running time of $O(k^2 \cdot k! \cdot |G|)$. So all in all, the construction of R_H takes at most $O((k \cdot k!)^2 \, |G|)$ operations. □

In the above lemma note that we can run the algorithm in parallel. Hence using parallelization, the algorithm could be sped up in practice.

Lemma 4.13 *H has an independent set I of size $|\mathcal{B}|$ if, and only if, I is also an independent set of size $|\mathcal{B}|$ in R_H.*

PROOF. One direction is clear: if I is an independent set of size $|\mathcal{B}|$ in H, then it is an independent set of size $|\mathcal{B}|$ in R_H. On the other hand, suppose I is an independent set of size $|\mathcal{B}|$ in R_H. Then for the sake of contradiction, suppose there are vertices $u, v \in I$ and an edge $e = \{u, v\} \in E(H)$, where e has the shortest length among all possible long edges in $H[I]$. Let us assume that $u \in \text{PermutList}(b_i), v \in \text{PermutList}(b_j)$ where $j > i + 1$. Suppose from each $\text{PermutList}(b_\ell)$ for $i \leq \ell \leq j$, we have $v_{b_\ell} \in I$, where $v_{b_i} = u, v_{b_j} = v$. Clearly as I is of size $|\mathcal{B}|$ there should be exactly one vertex from each $\text{PermutList}(b_\ell)$. We know $core(\text{Label}(u), \text{Label}(v)) \neq \emptyset$ as otherwise the edge $e = \{u, v\}$ was not in $E(H)$. On the other hand, as e is the smallest long edge which connects vertices of I, then there is no long edge between v_{b_i} and $v_{b_{j-1}}$ in H. That means $\text{Label}(v_{b_i}) \approx \text{Label}(v_{b_{j-1}})$ but then as $\text{Label}(v_{b_i}) \not\approx \text{Label}(v_{b_j})$ and by Linear Time Algorithm for Constant Number of Flows on DAGs we know that $core(\text{Label}(u), \text{Label}(v)) \subseteq \text{Label}(v_{b_{j-1}})$, so $\text{Label}(v_{b_j}) \not\approx \text{Label}(v_{b_{j-1}})$. Therefore, there is an edge between v_{b_j} and $v_{b_{j-1}}$: a contradiction, by our choice of I in R_h. □

R_H is a much simpler graph compared to H, which helps us find a large independent set of size $|\mathcal{B}|$ (if exists). We have the following lemma.

Lemma 4.14 *There is an algorithm that finds an independent set I of size exactly $|\mathcal{B}|$ in R_H if such an independent set exists; otherwise it outputs that there is no such an independent set. The running time of this algorithm is $O(|R_H|)$.*

PROOF. We find an independent set of size $|\mathcal{B}|$ (or we output there is no such set) by dynamic programming. For this purpose we define a function $f : [|\mathcal{B}|] \times V(R_H) \to 2^{V(R_H)}$ which is presented in detail in the algorithm below. Before providing said algorithm we explain it in plain text. It is a straightforward dynamic program: start from the left most groups of vertices in R_H (one extreme side of R_H). Consider every vertex as part of the independent set and build the independent set bottom up on those groups. We omit the proof of correctness and the exact calculation of the running time as it is clear from the algorithm.

Algorithm 3. **Finding an Independent Set of Size $|\mathcal{B}|$ in R_H**

Input: R_H

a) Set $f(i, v) := \emptyset$ for all $i \in [|\mathcal{B}|], v \in V(R_H)$.
b) Set $f(1, v) := v$ for all $v \in \text{PermutList}(b_1)$.
c) For $2 \leq i \leq [|\mathcal{B}|]$ do
   i. For all $v \in \text{PermutList}(b_i)$
      A. If there is a vertex $u \in \text{PermutList}(b_{i-1})$ and $|f(i-1, u)| = i - 1$ and $\{u, v\} \notin E(R_H)$ then $f(i, v) := f(i-1, u) \cup \{v\}$,
      B. otherwise set $f(i, v) := \emptyset$
d) If $\exists v \in \text{PermutList}(b_{|\mathcal{B}|})$ where $|f(|\mathcal{B}|, v)| = |\mathcal{B}|$ then output $f(|\mathcal{B}|, v)$,
e) otherwise output there is no such independent set. □

Our main theorem is now a corollary of the previous lemmas and algorithms.

Theorem 4.15 *There is a linear time FPT algorithm for the network update problem on an acyclic update flow network G with k flows (the parameter), which finds a feasible update sequence, if it exists; otherwise it outputs that there is no feasible solution for the given instance. The algorithm runs in time* $O(2^{O(k \log k)} |G|)$.

PROOF. First construct R_H using Algorithm 2, then find the independent set I of size $|\mathcal{B}|$ in R_H using Algorithm 3. If there is no such independent set I, then we output that there is no feasible update solution for the given network; this is a consequence of Lemmas 4.10 and 4.13. On the other hand, if there is such an independent set I, then one can construct the corresponding dependency graph and update all blocks, using the algorithm provided in the proof of Lemma 4.10. The dominant runtime term in the above algorithms is $O(k^2 \cdot k!^2 \cdot |G|)$ (from Lemma 4.14), which proves the claim of the theorem. □

4.2 Updating k-Flows in DAGs is NP-complete

In this section we show that, if the number of flows, k, is part of the input, the problem remains hard even on DAGs. In fact, we prove the following theorem.

Theorem 4.16 *Finding a feasible update sequence for k-flows is NP-complete, even if the update graph G is acyclic.*

To prove the theorem, we provide a polynomial time reduction from the 3-SAT problem. Let $C = C_1 \wedge \ldots \wedge C_m$ be an instance of 3-SAT with n variables $X_1, \ldots, X_n$, where each variable X_i appears positive (x_i) or negative ($\bar{x}_i$) in some clause C_j. We construct an acyclic network update graph G such that there is a feasible sequence of updates σ for G, if and only if C is satisfiable by some variable assignment σ. By Lemma 4.2, we know that if G has a feasible update sequence, then there is a feasible update sequence which updates each block in consecutive rounds.

In the following, we denote the first vertex of a directed path p with $head(p)$ and the end vertex with $tail(p)$. Furthermore, we number the vertices of a path p with numbers $1, \ldots, |V(p)|$, according to their order of appearance in p ($head(p)$ is number 1). We will write $p(i)$ to denote the i'th vertex in p.

We now describe how to construct the initial update flow network G.

1. G has a start vertex s and a terminal vertex t.
2. We define n variable selector flow pairs $S_1, \ldots, S_n$, where each $S_i = (S_i^o, S_i^u)$ is of demand 1, as follows:
   a) **Variable Selector Old Flows** are n s,t-flows $S_1^o, \ldots, S_n^o$ defined as follows: Each one consists of a directed path of length 3, where every edge in path S_i^o (for $i \in [n]$) has capacity 1, except for the edge $(S_i^o(2), S_i^o(3))$, which has capacity 2.
   b) **Variable Selector Update Flows** are n s,t-flows $S_1^u, \ldots, S_n^u$ defined as follows: Each consists of a directed path of length 5, where the edge's capacity of path S_i^u is set as follows. $(S_i^u(2), S_i^u(3))$ has capacity 2, $(S_i^u(4), S_i^u(5))$ has capacity m, and the rest of its edges has capacity 1.
3. We define m clause flow pairs $C_1, \ldots, C_n$, where each $C_i = (C_i^o, C_i^u)$ is of demand 1, as follows.
   a) **Clauses Old Flows** are m s,t-flows $C_1^o, \ldots, C_m^o$, each of length 5, where for $i, j \in [m]$, $C_i^o(3) = C_j^o(3)$ and $C_i^o(4) = C_j^o(4)$. Otherwise they are disjoint from the above

defined. The edge $(C_i^o(3), C_i^o(4))$ (for $i \in [m]$) has capacity m, all other edges in C_i^o have capacity 1.

b) **Clauses Update Flows** are m s,t-flows $C_1^u, \ldots, C_m^u$, each of length 3. Every edge in those paths has capacity 3.

4. We define a Clause Validator flow pair $V = (V^o, V^u)$ of demand m, as follows.

   a) **Clause Validator Old Flow** is an s,t-flow V^o whose path consists of edges $(s, S_1^u(4)), S_i^u(4), S_i^u(5)), (S_i^u(5), S_{i+1}^u(4)), (S_n^u(4), S_n^u(5)), (S_n^u(5), t)$ for $i \in [n-1]$. Note that, the edge $(S_i^u(4), S_i^u(5))$ (for $i \in [n]$) also belongs to S_i^u. All edges of V have capacity m.

   b) **Clause Validator Update Flow** is an s,t-flow V^u whose path has length 3, such that $V^u(2) = C_1^o(3), V^u(3) = C_1^o(4)$. All new edges of V^u have capacity m.

5. We define $2n$ literal flow pairs $L_1, \ldots, L_{2n}$. Each $L_i = (L_i^o, L_i^u)$ of demand 1 is defined as follows:

   a) **Literal's Old Flows** are $2n$ s,t-flows $L_1^o, \ldots, L_n^o$ and $\bar{L}_1^o, \ldots, \bar{L}_n^o$. Suppose x_i appears in clauses $C_{i_1}, \ldots, C_{i_\ell}$, then the path L_i^o is a path of length $2\ell + 5$, where $L_i^o(2j+1) = C_{i_j}^u(2), L_i^o(2j+2) = C_{i_j}^u(3)$ for $j \in [\ell]$ and furthermore $L_i^o(2\ell+3) = S_i^u(2), L_i^o(2\ell+4) = S_i^u(3)$. On the other hand, if $\bar{x}_i$ appears in clauses $C_{i_1}, \ldots, C_{i_{\ell'}}$, then $\bar{L}_i^o$ is a path of length $2\ell' + 5$ where $\bar{L}_i^o(2j+3) = C_{i_j}^u(, \bar{L}_i^o(2j+4) = C_{i_j}^u(3)$ for $j \in [\ell']$, and furthermore $\bar{L}_i^o(2\ell'+3) = S_i^u(2), \bar{L}_i^o(2\ell'+4) = S_i^u(3)$. All new edges in L_i^o (resp. $\bar{L}_i^o$) have capacity 3. Note that some L_i^os may share common edges.

   b) **Literal's Update Flows** are $2n$ s,t-flows $L_1^u, \ldots, L_n^u$ and $\bar{L}_1^u, \ldots, \bar{L}_n^u$. For $i \in [n]$, L_i^u and $\bar{L}_i^u$ are paths of length 5 such that $L_i^u(2) = \bar{L}_i^u(2) = S_i^o(2)$ and $L_i^u(3) = \bar{L}_i^u(3) = S_i^o(3)$. All new edges in those paths have capacity 3.

Lemma 4.17 *For σ and G, we have the following observations.*

i) We either have $\sigma(L_i^o) < \sigma(S_i^o) < \sigma(\bar{L}_i^o)$, or $\sigma(\bar{L}_i^o) < \sigma(S_i^o) < \sigma(L_i^o)$, for all $i \in [n]$.

ii) $\sigma(C_i^o) < \sigma(V^o)$ for all $i \in [m]$.

iii) $\sigma(S_i^o) < \sigma(V^o)$ for all $i \in [n]$.

iv) For every $i \in [m]$ there is some $j \in [n]$ such that $\sigma(C_i^o) < \sigma(L_j^o)$ or $\sigma(C_i^o) < \sigma(\bar{L}_j^o)$.

v) We either have $\sigma(L_j^o) < \sigma(C_i^o) < \sigma(\bar{L}_j^o)$, or $\sigma(\bar{L}_j^o) < \sigma(C_i^o) < \sigma(L_j^o)$, for all $i \in [m]$ and all $j \in [n]$.

Proof.

i) As the capacity of the edge $e = (S_i^o(2), S_i^o(3))$ is 2, and both $L_i^u, \bar{L}_i^u$ use that edge, before updating both of them, S_i^o (resp. S_i^u) should be updated. On the other hand, the edge $e' = (S_i^u(2), S_i^u(3))$ has capacity 2 and it is in both L_i^o and $\bar{L}_i^o$. So to update S_i^o, e' for one of the $L_i^o, \bar{L}_i^o$ should be updated.

ii) The edge $(V^u(2), V^u(3))$ of V^u also belongs to all C_i^o (for $i \in [m]$) and its capacity is m. Moreover, the demand of (V^o, V^u) is m, so V^o cannot be updated unless C_i^o has been updated for all $i \in [m]$.

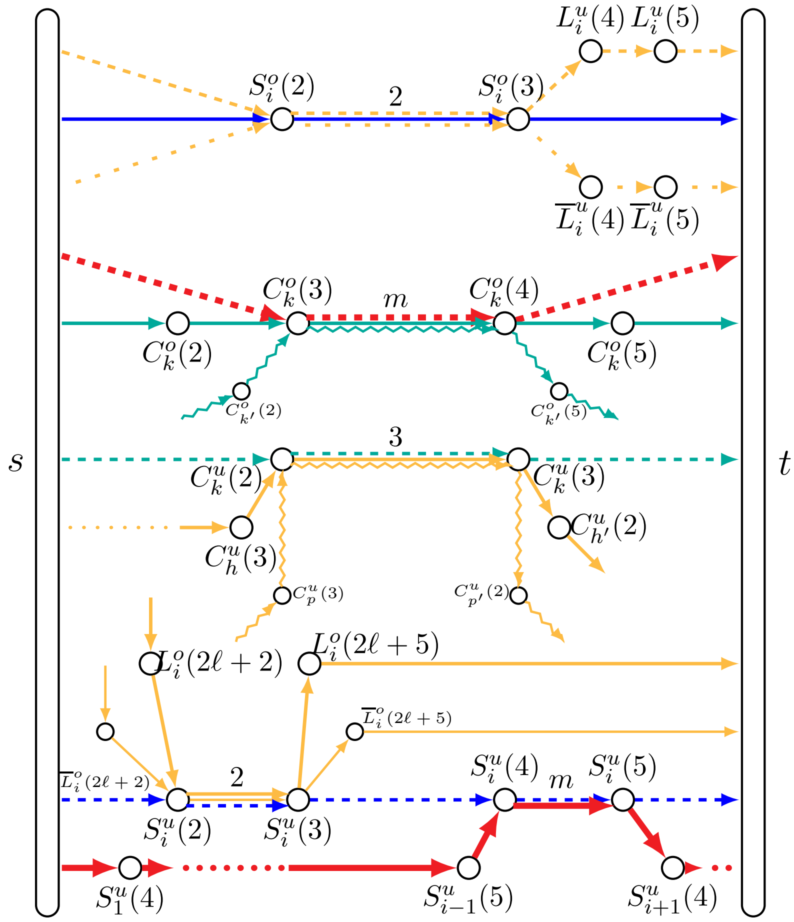

Figure 8: *Gadget Construction for Hardness in DAGs:* There are 4 types of flows: Clause flows, Literal flows, Clause Validator flow and Literal Selector flows. The edge $(S_i^o(2), S_i^o(3))$ cannot route 3 different flows S_i^o, L_i^u,$\bar{L}_i^u$ at the same time. On the other hand the edge $(S_i^u(2), S_i^u(3))$ cannot route the flow S_i^u before updating either L_i^o or $\bar{L}_i^o$, hence by the above observation, exactly one of the L_i or $\bar{L}_i$'s will be updated strictly before S_i and the other will be updated strictly after S_i was updated. Only after all Clause flows are updated, the edge $(C_k^o(3), C_k^o(4))$ can route the flow V (Clause Validator flow). A Clause flow C_k can be updated only if at least one of the Literal flows which goes along $(C_k^u(2), C_k^u(3))$ is updated. So in each clause, there should be a valid literal. On the other hand the Clause validator flow can be updated only if all Clause Selector flows are updated, this is guaranteed by the edge $(S_i^u(4), S_i^u(5))$. Hence, before updating all clauses, we are allowed to update at most one of the L_i or $\bar{L}_i$'s, and this corresponds to a valid satisfying assignment.

iii) Every S_i^u ($i \in [n]$) requires the edge $(S_i^u(4), S_i^u(5))$, which is also used by V^o, until after step $\sigma(V^o)$.

iv) This is a consequence of Observation iii and Observation ii.

v) This is a consequence of Observation iv and Observation i. □

PROOF. (PROOF OF THEOREM 4.16) Given a sequence of updates, we can check if it is feasible or not. The length of the update sequence is at most k times the size of the graph, hence, the problem clearly is in NP.

To show that the problem is complete for NP, we use a reduction from 3-SAT. Let C be as defined earlier in this section, and in polynomial time we can construct G.

By the construction of G, if there is a satisfying assignment σ for C, we obtain a sequence σ to update the flows in G as follows. First, if in σ we have $X_i = 1$ for some $i \in [n]$, update the literal flow L_i^o; otherwise update the literal flow $\bar{L}_i^o$. Afterwards, since σ satisfies C, for every clause C_i there is some literal flow L_j or $\bar{L}_j$, which is already updated. Hence, for all $i \in [m]$ the edge $(C_i^u(3), C_i^u(4))$ incurs a load of 2 while its capacity is 3. Therefore, we can update all of the clause flows and afterwards the clause validator flow V^o. Next, we can update the clause selector flows and at the end, we update the remaining half of the literal flows. These groups of updates can all be done consecutively.

On the other hand, if there is a valid update sequence σ for flows in G, by Lemma 4.17 observation v, there are exactly n literal flows that have to be updated, before we can update C_i^o. To be more precise, for every $j \in [n]$, either L_j^o, or $\bar{L}_j^o$ has to be updated, but never both. If L_j^o is one of those first n literal flows to be updated for some $j \in [n]$, we set $X_j := 1$; otherwise $\bar{L}_j^o$ is to be updated and we set $X_j := 0$. Since these choices are guaranteed to be unique for every $j \in [n]$, this gives us an assignment σ. After these n literal flows are updated, we are able to update the clause flows, since σ is a valid update sequence. This means in particular, that for every clause C_i, $i \in [m]$, there is at least one literal which is set to true. Hence σ satisfies C and therefore solving the network update problem on DAGs, is as hard as solving the 3-SAT problem. □

5 Related Work

To the best of our knowledge, our model is novel in the context of reconfiguration theory [40]. The reconfiguration model closest to ours is by Bonsma [7] who studied how to perform rerouting such that transient paths are always *shortest*. However, the corresponding techniques and results are not applicable in our model where we consider flows of certain *demands*, and where different flows may *interfere* due to capacity constraints in the underlying network.

The problem of how to update routes of flows has been studied intensively by the networking community recently [9, 24, 29, 33, 36], in particular in the context of software-defined networks and motivated by the unpredictable router update times [24, 27]. For an overview, we refer the reader to a recent survey by Foerster et al. [16]. In a seminal work by Reitblatt et al. [36], a strong *per-packet consistency* notion has been studied, which is well-aligned with the strong consistency properties usually provided in traditional networks [11]. Mahajan and Wattenhofer [33] 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, and their study was recently refined in [3, 17, 18], where the authors also establish hardness results, as well as in [14, 30, 31, 32], which respectively, focus on the problem of minimizing the number of scheduling rounds [31], initiate the study of multiple policies [14], and introduce additional transient routing constraints related to waypointing [30, 32]. 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. [29] on congestion-free rerouting has already been extended in several papers, using static [8, 21, 37, 43], dynamic [42], or time-based [34, 35] approaches. Vissicchio et al. presented FLIP [41], which combines per-packet consistent updates with order-based rule replacements, in order to reduce memory overhead: additional rules are used only when necessary. Moreover, Hua et al. [22] recently initiated the study of adversarial settings, and presented FOUM, a flow-ordered update mechanism that is robust to packet-tampering and packet dropping attacks.

However, to the best of our knowledge, bandwidth capacity constraints have so far only been considered in strong, per-packet consistent settings, and for splittable flows. We in this paper argue that this is both impractical (splittable flows introduce a wide range of problems and overheads) as well as too restrictive (per-packet consistent updates require traffic marking and render many problem instances infeasible).

6 Conclusion

This paper initiated the study of a natural and fundamental reconfiguration problem: the congestion-free rerouting of unsplittable flows. Interestingly, we find that while *computing* disjoint paths on DAGs is $W[1]$-hard [39] and finding routes under congestion even harder [1], *reconfiguring* multicommodity flows is fixed parameter tractable on DAGs. However, we also show that the problem is NP-hard for an arbitrary number of flows.

In future work, it will be interesting to chart a more comprehensive landscape of the computational complexity for the network update problem. In particular, it would be interesting to know whether the complexity can be reduced further, e.g., to $2^{O(k)}O(|G|)$. More generally, it will be interesting to study other flow graph families, especially more sparse graphs or graphs of bounded DAG width [2, 6].

Acknowledgements. We would like to thank Stephan Kreutzer, Arne Ludwig and Roman Rabinovich for discussions on this problem.

References

[1] Saeed Akhoondian Amiri, Stephan Kreutzer, Dániel Marx, and Roman Rabinovich. Routing with congestion in acyclic digraphs. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS*, pages 7:1–7:11, 2016.

[2] Saeed Akhoondian Amiri, Stephan Kreutzer, and Roman Rabinovich. Dag-width is pspace-complete. *Theor. Comput. Sci.*, 655:78–89, 2016.

[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] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. Rsvp-te: Extensions to rsvp for lsp tunnels. In *RFC 3209*, 2001.

[5] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs - theory, algorithms and applications*. Springer, 2002.

[6] Dietmar Berwanger, Anuj Dawar, Paul Hunter, Stephan Kreutzer, and Jan Obdrzálek. The dag-width of directed graphs. *J. Comb. Theory, Ser. B*, 102(4):900–923, 2012.

[7] Paul Bonsma. The complexity of rerouting shortest paths. *Theoretical computer science*, 510:1–12, 2013.

[8] 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.

[9] 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.

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

[11] 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.

[12] Chandra Chekuri, Alina Ene, and Marcin Pilipczuk. Constant congestion routing of symmetric demands in planar directed graphs. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP*, 2016.

[13] Chandra Chekuri, Sreeram Kannan, Adnan Raja, and Pramod Viswanath. Multicommodity flows and cuts in polymatroidal networks. *SIAM J. Comput.*, 44(4):912–943, 2015.

[14] 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.

[15] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5(4):691–703, 1976.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Stephan Kreutzer. An excluded half-integral grid theorem for digraphs and the directed disjoint paths problem. In *Proc. Symposium on Theory of Computing (STOC)*, pages 70–78, 2014.

[26] Jon M. Kleinberg. Decision algorithms for unsplittable flow and the half-disjoint paths problem. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 530–539, 1998.

[27] 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.

[28] Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.

[29] 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.

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

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

[32] 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.

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

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

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

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

[37] 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.

[38] Martin Skutella. Approximating the single source unsplittable min-cost flow problem. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2000.

[39] Aleksandrs Slivkins. Parameterized tractability of edge-disjoint paths on directed acyclic graphs. *SIAM Journal on Discrete Mathematics*, 24(1):146–157, 2010.

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

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

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

[43] 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.