Congestion-Free Rerouting of Flows on DAGs

Saeed Akhoondian Amiri
Max-Planck Institute of Informatics, Germany
samiri@mpi-inf.mpg.de

Szymon Dudycz
University of Wroclaw, Poland
szymon.dudycz@gmail.com

Stefan Schmid
University of Vienna, Austria
stefan_schmid@univie.ac.at

Sebastian Wiederrecht
TU Berlin, Germany
sebastian.wiederrecht@tu-berlin.de

Abstract

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. 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.

2012 ACM Subject Classification Networks → Network algorithms, Theory of computation → Network flows

Keywords and phrases Unsplittable Flows, Reconfiguration, DAGs, FPT, NP-Hardness

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.143

Funding The research of Saeed Amiri and Sebastian Wiederrecht was partly supported by the ERC consolidator grant DISTRUCT, agreement No 648527. Stefan Schmid was supported by the Danish VILLUM foundation project *ReNet*.

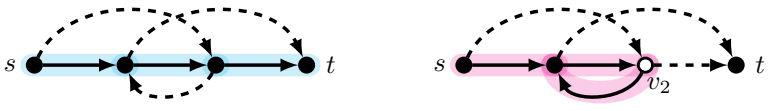
Acknowledgements We would like to thank Stephan Kreutzer, Arne Ludwig and Roman Rabinovich for discussions on this problem.

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) [24], satisfiability [15], independent sets [16], vertex coloring [9], or matroid bases [17], to just name a few.

45th International Colloquium on Automata, Languages, and Programming (ICALP 2018).
Editors: Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella;
Article No. 143; pp. 143:1–143:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



initial network invalid update: no transient flow (loop!)

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.

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 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. [13].

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, 10, 11, 12, 18, 19, 20, 22], 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, 14, 21].

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.

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 constant k, we present a linear-time (fixed parameter tractable) algorithm which 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.

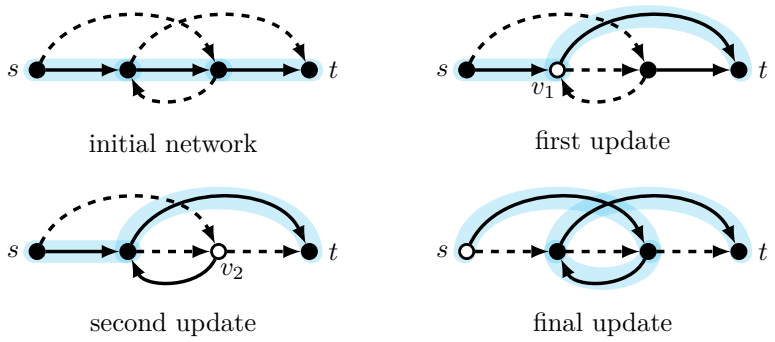


Figure 2 *Example:* We revisit the network of Figure 1 and reroute from F^o to F^u without interrupting the connection between s and t along a unique (transient) path (in blue). To avoid the problem seen in Figure 1, we first update the vertex v_1 in order to establish a shorter connection from s to 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 general k, deciding whether a feasible schedule exists is NP-hard even on loop-free networks (i.e., DAGs).

Against the backdrop that the problem of *routing* disjoint paths on DAGs is known to be $W[1]$-hard [23] and computing routes *subject to congestion* even harder [1], our 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, and we represent dependencies between blocks by a (directed) dependency graph D. The structure of D is sophisticated, hence to analyze it, we first construct a helper graph H. In our first main technical lemma, we show that if there is an independent set I in H, then the dependency graph that corresponds to the vertices of I is a DAG (Lemma 11). So we may concentrate on a subgraph of D with a simpler structure, which we use to prove the next main technical lemma: there is a congestion-free rerouting if and only if the maximum independent set in H is large enough (Lemma 14). We are left with the challenge that finding a maximum independent set is a hard problem, even in our very restricted graph classes. We hence carefully modify H to obtain a much simpler graph of *bounded pathwidth*, without losing any critical properties. Thanks to these lemmas, the proof of the main theorem will follow.

In addition to our algorithmic contributions, we present NP-hardness proofs. These hardness proofs are based on novel and non-trivial insights into the flow rerouting problem, which might be helpful for similar problems in the future.

2 Model and Definitions

The 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 capacitated 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 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 [21].

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

The proof, briefly sketched in the following, is by reduction from 3-SAT. Let C be any 3-SAT formula with n variables and m clauses. Denote the variables by $X_1, \ldots, X_n$ and the clauses by $C_1, \ldots, C_m$. The resulting update flow network is denoted by $G(C)$. Assume that the variables are ordered by their indices, and their appearance in each clause respects this order.

We 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 contains 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 contains gadgets representing the clauses: they have edges that are "blocked" by the variable edges of B. Therefore, we will need to update B to enable the updates of R.

4 Rerouting Flows in DAGs

We now consider the flow rerouting problem when the underlying flow graph is acyclic. In particular, we identify an important substructure arising for flow-pairs in acyclic graphs, which we call *blocks*. These blocks will play a major role in both the hardness proof and the algorithm presented in this section.

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 a DAG. 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 $V(F_i^o) \cap V(F_i^u) = \{z_1^i, \ldots, z_{k_i}^i\}$ be ordered by $\prec$ as well.

The subgraph of $F_i^o \cup F_i^u$ induced by the set $\{v \in V(F_i^o \cup F_i^u) \mid z_j^i \prec v \prec z_{j+1}^i\}$, $j \in [k_i - 1]$, 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 an 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 key to our flow reconfiguration algorithm (presented below), as it allows us to consider the update of blocks as a whole, rather than vertex-by-vertex.

▶ **Lemma 4.** *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.*

▶ **Lemma 5.** *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.*

Recall 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 5). For a single flow F, we write $\sigma(F)$ for the round where the last edge of F was updated.

4.1 Updating k-Flows in DAGs is NP-complete

We first show that if k is part of the input, the congestion-free flow reconfiguration problem is even hard on the DAG. Hence the algorithm presented in the following is essentially tight. To prove the theorem, we use a polynomial time reduction from the 3-SAT problem.

▶ **Theorem 6.** *Finding a feasible update sequence for k-flows is NP-complete, even if the update graph G is acyclic.*

4.2 Linear Time Algorithm for Constant Number of Flows on DAGs

By Theorem 6 we cannot hope to find a polynomial time algorithm that finds a feasible update sequence. However, if the problem is parameterized by the number k of flows, a rerouting sequence can be computed in FPT-linear time if the update graph is acyclic. In this subsection 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 $\mathscr{S}(b_1) \prec u \prec \mathscr{E}(b_1)$. 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.

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'. 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.

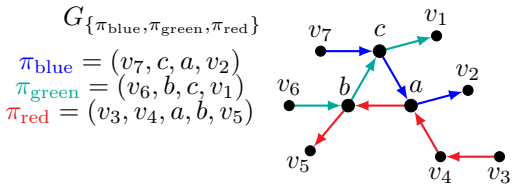


Figure 3 *Example:* The dependency graph of three pairwise consistent permutations π_{blue}, π_{green} and π_{red}. Each pair of those permutation has exactly one vertex in common and with this the cycle (a, b, c) is created. With such cycles being possible, a dependency graph does not necessarily contain sink vertices. To get rid of them, we certainly need some more refinements.

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 (see Figure 3).

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 1) here.

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 5. 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 currently active on e exceed $c(e)$. If so output: *Fail*.

▶ **Lemma 7.** *Let π be a permutation of the set $\mathcal{B}_1 \subseteq \mathcal{B}$. Whether π is congestion free can be determined in time $O(k \cdot |G|)$.*

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 $\mathcal{B}'$ and initialize it 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 $\text{TouchingBlocks}(b)$ of blocks which touch the block b, note that this

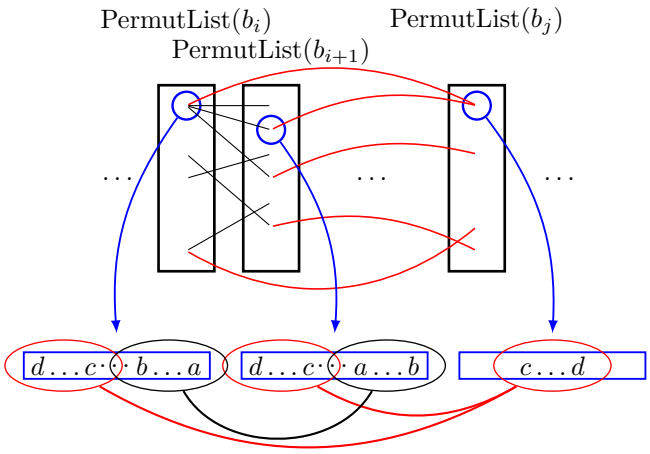


Figure 4 *Example:* The graph H consists of vertex sets PermutList(b_i), $i \in [|\mathcal{B}|]$, where each such partition contains all congestion free sequences of the at most k iteratively chosen touching blocks. In the whole graph, we then create edges between the vertices of two such partitions 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.

set is dynamically defined: it depends on $\mathcal{B}'$. 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 $v \in$ PermutList(b) comes with a **label** which corresponds to some congestion-free permutation of elements of TouchingBlocks(b). We denote that permutation by Label (v).

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

i Set $H \coloneqq \emptyset$, $\mathcal{B}' \coloneqq \mathcal{B}$, PermutList $\coloneqq \emptyset$.
ii For $i \coloneqq 1, \dots, |\mathcal{B}|$ do

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

We have the following lemmas based on our construction.

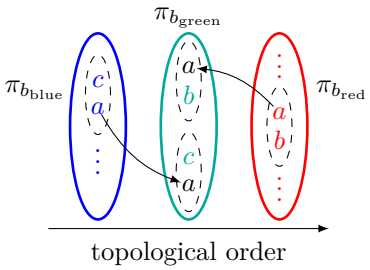


Figure 5 *Example:* Select one of the permutations of length at most k from every PermutList(b). These permutations obey the Touching Lemma. Taking the three permutations from the example in Figure 3, we can see that the Touching Lemma forces a to be in the green permutation as well. Assuming consistency, 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.

▶ **Lemma 8.** *For Item (ii2) of the construction of H, $t \leq k$ holds.*

▶ **Lemma 9** (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}.*

For an illustration of the property described in the Touching Lemma, see Figure 5: it refers to the dependency graph of Figure 3. This example also points out the problem with directed cycles in the dependency graph and the property of the Touching Lemma, which is crucial for Observation 10 and Lemma 11.

We prove a series of lemmas in regard to the dependency graph of elements of H, to establish the base of the inductive proof for Lemma 13.

▶ **Observation 10.** *Let π be a permutation of a set S. Then the dependency graph G_π does not contain a cycle.*

▶ **Lemma 11.** *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.*

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

▶ **Definition 12** (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 $\Pi = \{\pi_1, \ldots, \pi_n\}$ 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 11 and Observation 10 and uses them as the base of its inductive proof.

▶ **Lemma 13.** *Let $I = \{v_{\pi_1}, \ldots, v_{\pi_\ell}\}$ be an independent set in H. Then the dependency graph G_Π, for $\Pi = \{\pi_1, \ldots, \pi_\ell\}$, 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 11 and Observation 10 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 a shortest cycle in G_Π^+. By Lemma 11 and Observation 10 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 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 $a_i, a_j \in$ TouchingBlocks(b) or if $a_i, a_j \in$ TouchingBlocks(b) then (a_i, a_j) or (a_j, a_i) is an edge in C. Additionally $\left|B^I\right| = |C|$.

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.

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.

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 11. ◀

The following lemma is the key to establish a link between independent sets in H and feasible update sequences of the corresponding update flow network G.

▶ **Lemma 14.** *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 $O(k \cdot |G|)$.*

With Lemma 14, the update problem boils down to finding an independent set of size $|\mathcal{B}|$ in H.

Finding an independent set of size $|\mathcal{B}|$ in H is a hard problem already on very restricted class families. Hence, we trim H to avoid the above problem. We will use the special properties of the touching relation of blocks. We say that an edge $e \in E(H)$ is *long*, if one end of e is in PermutList(b_i), and the other in 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. We prove the following lemmas.

▶ **Lemma 15.** *There is an algorithm which computes R_H in time $O((k \cdot k!)^2 |G|)$.*

▶ **Lemma 16.** *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.*

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 17.** *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 $O(|R_H|)$.*

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

▶ **Theorem 18.** *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 $O(2^{O(k \log k)} |G|)$.*

5 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 [23] and finding routes under congestion as well [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]. Finally, besides feasibility, it remains to study algorithms to efficiently compute *short* schedules.

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

10 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.

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

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

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

14 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.

15 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.

16 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.

17 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.

18 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.

19 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.

20 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.

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

22 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.

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

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