On the Complexity of Local Graph Transformations

Christian Scheideler
Paderborn University, Germany
https://cs.uni-paderborn.de/en/ti/
scheideler@upb.de

Alexander Setzer
Paderborn University, Germany
https://cs.uni-paderborn.de/en/ti/
asetzer@mail.upb.de

Abstract
We consider the problem of transforming a given graph $G_s$ into a desired graph $G_t$ by applying a minimum number of primitives from a particular set of local graph transformation primitives. These primitives are local in the sense that each node can apply them based on local knowledge and by affecting only its 1-neighborhood. Although the specific set of primitives we consider makes it possible to transform any (weakly) connected graph into any other (weakly) connected graph consisting of the same nodes, they cannot disconnect the graph or introduce new nodes into the graph, making them ideal in the context of supervised overlay network transformations. We prove that computing a minimum sequence of primitive applications (even centralized) for arbitrary $G_s$ and $G_t$ is NP-hard, which we conjecture to hold for any set of local graph transformation primitives satisfying the aforementioned properties. On the other hand, we show that this problem admits a polynomial time algorithm with a constant approximation ratio.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Theory of computation → Approximation algorithms analysis

Keywords and phrases Graphs transformations, NP-hardness, approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.150

Category Track C: Foundations of Networks and Multi-Agent Systems: Models, Algorithms and Information Management


Funding This work was supported by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901) under Grant No.: GZ SFB 901/02.

1 Introduction

Overlay networks are used in many contexts, including peer-to-peer systems, multipoint VPNs, and wireless ad-hoc networks. In fact, any distributed system on top of a shared communication infrastructure usually has to form an overlay network (i.e., its participating sites have to know each other or at least some server) to allow the exchange of information.

A fundamental task in the context of overlay networks is to maintain or adapt its topology to a desired topology, where the desired topology might either be pre-defined or depend on a certain objective function. The problem of reaching a pre-defined topology has been extensively studied in the context of self-stabilizing overlay networks (e.g., [29, 21, 12, 5, 22, 7]), and the problem of adapting the topology based on a certain objective function has been studied in the context of self-adapting and -optimizing overlay networks (e.g., [33, 14, 2, 19, 11, 3, 10, 8]). Many of these approaches are decentralized, and because of that,
the work (in terms of number of edge changes) they need to adapt to a desired topology might be far away from the minimum possible work to reach that topology. In fact, no non-trivial results on the competitiveness of decentralized overlay network adaptations are known so far other than handling single join or leave operations, and it is questionable whether any good competitive result can be achieved with a decentralized approach. An alternative approach would be that a server is available for controlling the network adaptations, and this has already been considered in the context of so-called supervised overlay networks. In a supervised overlay network there is a dedicated, trusted node called supervisor that controls all network adaptations but otherwise is not involved in the functionality of the overlay network (such as serving search requests), which is handled in a peer-to-peer manner. This has the advantage that even if the supervisor is down, the overlay network is still functional. Solutions for supervised overlay networks have been proposed in [24, 15], for example, and the results in [24] imply that, for specific overlay networks, any set of node arrivals and departures can be handled in a constant competitive fashion (concerning the work needed for adding and removing edges) to get back to a desired topology. But no general result is known so far for supervised overlay networks concerning the competitiveness of converting an initial topology into a desired topology. Also, no result is known so far on how to handle the problem that a supervisor could be faulty or even act maliciously.

A malicious supervisor would pose a significant problem for an overlay network since it could easily launch Sybil attacks (i.e., flooding the overlay network with fake or adversarial nodes) or Eclipse attacks (i.e., isolating nodes from other nodes in the overlay network). We thus ask: Can we limit the power of a supervisor such that it cannot launch an eclipse or sybil attack while still being able to convert the overlay network from any connected topology to any other connected topology?

We answer the question to the affirmative by determining a set of graph transformation commands, also called primitives, that only the supervisor may issue to the nodes. These primitives are powerful enough to transform any (weakly) connected topology into any other (weakly) connected topology but still allow the nodes to locally check that applying them does not disconnect the network or introduce a new node into the network. We additionally aim at minimizing the reconfiguration overhead, i.e., the number of commands to be issued (and, related to this, the number of changes to be made to node neighborhoods) to reach a desired topology. Unfortunately, as we will show, this cannot be done efficiently for the set of primitives we consider unless P \neq NP, and we conjecture that this holds for any set of commands that has the aforementioned property of giving the participants the ability to locally check that they cannot be used for eclipse or sybil attacks. However, we are able to give an O(1)-approximation algorithm for this problem.

1.1 Model and Problem Statement

We model the overlay network as a graph, i.e., nodes represent participants of the network and if there is a directed edge (u, v) in the graph, this means that there is a connection from u to v. Undirected edges \{u, v\} model the two connections from u to v and from v to u. Since there may be multiple connections between the same pair of participants, the graphs we consider in this work are multigraphs, i.e., edges may appear several times in the (multi-)set of edges. For convenience throughout this work we will use the term “graph” instead of multigraph and refer to “edge sets” even though their elements need not be unique.

We consider the following set \(P_d\) of four primitives for the manipulation of directed graphs, first introduced by Koutsopoulos et al. [25] in the context of overlay networks:
Introduction. If a node \( u \) has a reference of two nodes \( v \) and \( w \) with \( v \neq w \), \( u \) introduces \( w \) to \( v \) if \( u \) sends a message to \( v \) containing a reference of \( w \) while keeping the reference.

Delegation. If a node \( u \) has a reference of two nodes \( v \) and \( w \) s.t. \( u, v, w \) are all different, then \( u \) delegates \( w \)’s reference of \( v \) if \( u \) sends a message to \( v \) containing a reference of \( w \) and deletes the reference of \( w \).

Fusion. If a node \( u \) has two references \( v \) and \( w \) with \( v = w \), then \( u \) fuses the two references if it only keeps one of these references.

Reversal. If a node \( u \) has a reference of some other node \( v \), then \( u \) reverses the connection if it sends a reference of itself to \( v \) and deletes its reference of \( v \).

The four primitives are visualized in Figure 1. Note that for the Introduction primitive, it is possible that \( w = u \), i.e., \( u \) introduces itself to \( v \). To simplify the description, we sometimes say that a node \( u \) introduces or delegates an edge \((u, v)\) if \( u \) introduces \( v \) to some other node or delegates \( v \)’s reference to some other node, respectively.

The primitives in \( P_d \) are known to be universal (c.f. [25]), i.e., it is possible to transform any weakly connected graph into any other weakly connected graph by using only the primitives in \( P_d \). Note that for every edge \((u, v)\) used in any of the primitives, either \((u, v)\) still exists after the corresponding primitive is applied, or there is still an (undirected) path from \( u \) to \( v \) in the resulting graph. This directly implies that no application of the primitives can disconnect the graph. We assume that all connections are authorized, meaning that both endpoints are aware of the other endpoint of this connection. Thus, if for an edge \((u, v)\) that is supposed to be transformed into \((v, u)\) by an application of the Reversal Primitive, \( v \) checks that \( u \) actually was the previous endpoint of the former edge then the primitives cannot be used to introduce new nodes into the graph.

For undirected graphs, consider the set \( P_u \) containing only the primitives Introduction, Delegation and Fusion (defined correspondingly). These three primitives, accordingly, are universal on undirected graphs, i.e., any connected undirected graph can be transformed into any other connected undirected graph by applying the primitives in \( P_u \) (c.f. [25]).

We make the following observation:

\textbf{Observation 1.} The Introduction primitive is the only primitive that can increase the number of edges in a graph. The Fusion primitive is the only primitive that can decrease the number of edges in a graph. The Delegation primitive is the only primitive that can remove the last edge between two nodes (i.e., an edge of multiplicity one).

A computation \( C \) is a finite sequence \( G_1 \Rightarrow G_2 \Rightarrow \cdots \Rightarrow G_l \) of either directed or undirected graphs, in which each graph \( G_{i+1} \) is obtained from \( G_i \) by the application of a single primitive from \( P_d \) or \( P_u \), respectively. The graphs \( G_1 \) and \( G_l \) are called the initial and the final graphs of \( C \), respectively. The variable \( l \) is called the length of the computation.
We define the **Undirected Local Graph Transformation Problem** (ULGT) as follows: given two connected undirected graphs \( G_s, G_t \), find a computation of minimum length whose initial graph is \( G_s \) and whose final graph is \( G_t \). The corresponding decision problem \( k\text{-ULGT} \) is defined as follows: given a positive integer \( k \) and two connected undirected graphs \( G_s \) and \( G_t \), decide whether there is a computation with initial graph \( G_s \) and final graph \( G_t \) of length at most \( k \). Accordingly we define the **Directed Local Graph Transformation Problem** (DLGT) and \( k\text{-DLGT} \), which differ from the according problems in that the graphs are directed.

### 1.2 Related Work

Graph transformations have been studied in many different contexts and applications, including but not limited to pattern recognition, compiler construction, computer-aided software engineering, description of biological developments in organisms, and functional programming languages implementation (for a more detailed introduction and literature overview, we refer the reader to [4], [20], or [31, 13]). Simply put, a graph transformation (or graph-rewriting) system consists of a set of rules \( L \rightarrow R \) that may be applied to subgraphs isomorphic to \( L \) of a given graph \( G \) thus replacing \( L \) with \( R \) in \( G \). Since changing the labels assigned to a graph (graph relabeling) is also a kind of graph transformation, basically every distributed algorithm can be understood as a graph transformation system (c.f. [13]).

The type of graph transformations probably closest related to our work is the area of **Topology Control** (TC). In simple terms, the goal of TC is to select a subgraph of a given input graph that fulfills certain properties (such as connectivity) and optimizes some value (such as the maximum degree). This problem has been studied in a variety of settings (for surveys on this topic see, e.g., [27], or [6]) and although the usual approach is decentralized, there are also some centralized algorithms in this area (see, e.g., [30]). However, these works only consider the complexity of computing an optimal topology (instead of the complexity of transforming the graph by a minimum number of rule applications). There is one work by Lin [28] proving the \( \text{NP} \)-hardness of the **Graph Transformation Problem**, in which the goal is to find the minimum integer \( k \) such that an initial graph \( G_s \) can be transformed into a final graph \( G_t \) by adding and removing at most \( k \) edges in \( G_s \). Our work differs from that work in that we do not allow arbitrary edge relocations but restrict them to a set of rules that can be applied locally (and we also provide constant-factor approximation algorithms).

Our approximation algorithms use an approximation algorithm for the Undirected Steiner Forest Problem as a black-box (also known as the Steiner Subgraph Problem with edge sharing, or, in generalizations, the Survivable Network Design Problem or the Generalized Steiner Problem). 2-approximations of this problem were first given by Agrawal, Klein, and Ravi [1], and by Goemans and Williamson [16], and later also by Jain [23]. Gupta and Kumar [18] showed a simple greedy algorithm to have a constant approximation ratio and recently, Groß et al. [17] presented a local-search constant approximation for Steiner Forest.

### 1.3 Our Contribution

The main contributions of this paper are as follows: We prove the Undirected and the Directed Local Graph Transformation Problem to be \( \text{NP} \)-hard in Section 2. Furthermore, in Section 3 we show that they belong to \( \text{APX} \), i.e., there exist constant approximation algorithms for these two problems.
2 NP-hardness results

In this section, we show the NP-hardness of the Undirected Local Graph Transformation Problem by proving the NP-hardness of k-ULGT (see Section 2.1). Since k-DLGT’s NP-hardness is very similar for k-ULGT, we omit its proof and only briefly sketch the differences in the full version of this paper [32].

Throughout this section, for any positive integer i we use the notation [i] to refer to the set \{1, 2, ..., i\}.

2.1 k-ULGT is NP-hard

We prove k-ULGT’s hardness via a reduction from the Boolean satisfiability problem (SAT) which was proven to be NP-hard by Cook [9] and, independently, by Levin [26]. We briefly recap SAT as follows:

\begin{definition}[SAT]
Given a set \(X\) of \(n\) Boolean variables \(x_1, \ldots, x_n\) and a Boolean formula \(\Phi\) over the variables in \(X\) in conjunctive normal form (CNF), decide whether there is a truth assignment \(t: X \rightarrow \{0, 1\}\) that satisfies \(\Phi\).
\end{definition}

To reduce SAT to k-ULGT, we use the following reduction function:

\begin{definition}[Reduction function]
Let \(S = (X, \Phi)\) be a SAT instance, in which \(X = \{x_1, \ldots, x_n\}\) is the set of Boolean variables and \(\Phi = C_1 \land \cdots \land C_m\) for clauses \(C_1, \ldots, C_m\). Then \(f(S) = (G_s, G_t, k)\) in which \(k = 2n + m\) and \(G_s\) and \(G_t\) are undirected graphs defined as follows. Without loss of generality, assume that each literal \(y_i \in \{x_i, \overline{x}_i\}\) occurs only once in each clause. We say \(y_i \in C_j\) if literal \(y_i\) occurs in \(C_j\).

We define the following sets of nodes: \(V_C = \{C_1, \ldots, C_m\}\), and \(V_{X'} = \{x_i, \overline{x}_i, s_i, t_i\}\). Then, the set of nodes of \(G_s\) and \(G_t\) is \(V = \bigcup_{1 \leq i \leq n} V_{X'} \cup V_C \cup \{r\}\). For the set of edges, define \(E_{X'} = \{(s_i, x_i), (\overline{x}_i, s_i), (x_i, t_i), (t_i, \overline{x}_i)\}\), \(E_{C_j} = \{(y_i, C_j) | y_i \in \{x_i, \overline{x}_i\} \land y_i\) occurs in \(C_j\}\), \(E_{sr} = \{(s_i, r) | 1 \leq i \leq n\}\), \(E_{tr} = \{(t_i, r) | 1 \leq i \leq n\}\), \(E_{Cr} = \{(C_j, r) | 1 \leq j \leq m\}\). Both \(G_s\) and \(G_t\) have the edges in \(\bigcup_{1 \leq i \leq n} E_{X'} \cup \bigcup_{1 \leq j \leq m} E_{C_j}\). Additionally, \(G_s\) has the edges in \(E_{sr}\) and \(G_t\) has the edges in \(E_{tr} \cup E_{Cr}\).

Intuitively, each variable \(x_i\) is mapped to a gadget \(X_i\) consisting of the four nodes \(x_i, \overline{x}_i, s_i, t_i\), and \(t_i\). Also each clause \(C_j\) is connected with each literal occurring within it. Lastly, in \(G_s\), each of the \(s_i\) is connected with the node \(r\), whereas in \(G_t\), each of the \(t_i\) and each of the \(C_j\) are connected with \(r\). Figure 2 shows an example of the output of the reduction function for a given formula in CNF.

We now show that every SAT instance \(S\) is satisfiable if and only if \(f(S)\) is a “yes” instance of k-ULGT. We start with the “only if” part for this is the simpler direction:

\begin{lemma}
If a SAT instance \(S\) as in Definition 2 is satisfiable then \(f(S) = (G_s, G_t, k)\) with \(k = 2n + m\) is a k-ULGT instance and there is a computation with initial graph \(G_s\) and final graph \(G_t\) of length at most \(2n + m\).
\end{lemma}

\begin{proof}
Assume there is a satisfying truth assignment \(t: X \rightarrow \{0, 1\}\) of \(S\). For every \(1 \leq i \leq n\) let \(y_i := x_i\) if \(t(x_i) = 1\) or \(y_i := \overline{x}_i\) if \(t(x_i) = 0\). We construct the following computation with initial graph \(G_s\) and final graph \(G_t\):
\begin{enumerate}
\item For every \(1 \leq i \leq n\), \(s_i\) delegates the edge \(\{s_i, r\}\) to \(y_i\).
\item For every \(C_j \in \{C_1, \ldots, C_m\}\) choose one neighbor \(z_j \in \{y_1, \ldots, y_n\}\) (we show below that this exists), and let \(z_j\) introduce \(r\) to \(C_j\).
\item For every \(1 \leq i \leq n\), \(y_i\) delegates the edge \(\{y_i, r\}\) to \(t_i\).
\end{enumerate}
\end{proof}
On the Complexity of Local Graph Transformations

Figure 2 Graph $G_s$ returned by the reduction function for the (example) Boolean formula $(x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \land (\neg x_1 \lor \neg x_3)$. $G_t$ differs from $G_s$ in that the dashed edges do not exist and all grey nodes share an edge with node $r$.

Obviously, the length of this computation is $2n + m$. To prove the missing part, recall that every $C_j$ is satisfied under $t$, i.e., there is at least one literal $z_j$ in $C_j$ that evaluates to true, i.e., there is an $i \in [n]$ such that $z_j = x_i$ if $t(x_i) = 1$, or $z_j = \neg x_i$ if $t(x_i) = 0$. By definition of $y_i$, $z_j = y_i$. Thus because $z_j$ occurs in $C_j$, $y_i$ was a neighbor of $C_j$ during Step 2.

The “if” part is more complex. We begin with the following insight that will prove helpful in the course of this part.

Lemma 4. Suppose the nodes in the initial graph of a computation $C$ can be decomposed into disjoint sets $V_1, \ldots, V_k, P$ such that there is no edge $\{u, v\}$ for some $u \in V_i$, $v \in V_j$, $i, j \in [k], i \neq j$ and throughout $C$ none of the nodes in $P$ applies a primitive. Then there is no edge $\{u, v\}$ for some $u \in V_i$, $v \in V_j$, $i, j \in [k], i \neq j$ in any graph of the computation.

Proof. Assume there is a computation $C$ and sets $V_1, \ldots, V_k, P$ as defined above and assume for contradiction that the claim is not true. We consider the first edge $\{u, v\}$ such that $u \in V_i$, $v \in V_j$, $i, j \in [k], i \neq j$. Clearly, it cannot have been created by the application of a Fusion primitive. Thus it must have been created by an Introduction or Delegation primitive applied by a node $w$ that knew both $u$ and $v$ before the application of this primitive. By definition of $P$, $w \notin P$, i.e., $w \notin V_l$ for some $l \in [k]$. However, by the definition of $\{u, v\}$, $u$ and $v$ must have been from $V_l$ as well, yielding a contradiction.

The next lemma we show represents a main building block of the proof of the “if” part.

Lemma 5. Let $S$ be a SAT instance and let $(G_s, G_t, k) = f(S)$. For every computation $C$ with initial graph $G_s$ and final graph equal to $G_t$ of length at most $2n + m$ it holds: There are $y_1, \ldots, y_n$, $y_i \in \{x_i, \neg x_i\}$ for every $i \in [n]$, such that in $C$ there are no edges other than $E(G_s) \cup E(G_t) \cup \{\{y_i, r\} | i \in [n]\}$ and no edge occurs twice (where $E(G_s)$ and $E(G_t)$ denote the edge set of $G_s$ and $G_t$, respectively).

Due to space constraints, we only sketch the proof here, whereas the full proof can be found in the full version of this paper [32]. The general idea of the proof of Lemma 5 is the following: To obtain the target graph, for each $j \in [m]$ the edge $\{C_j, r\}$ has to be created and for
In the following, we refer to the variables defined in Definition 2. Furthermore, we refer to the algorithm in detail and then summarize its pseudo-code in Algorithm 1. Our algorithm. For an initial graph $G$, we first describe an approximation algorithm for $\text{ULGT}$ (see Section 3.1) and prove it to have a constant approximation ratio (see Section 3.2). Note that a constant approximation factor algorithm for $\text{DLGT}$ can be obtained by a slight adaptation of this algorithm. For a description of this, we refer the reader to the full version [32] due to space constraints.

As an ingredient our algorithm uses a 2-approximation algorithm (see Section 1.2) for the Undirected Steiner Forest Problem (USF) defined as follows: Given a graph $G$ and a set $S$ of pairs of nodes from $G$, find a forest $F$ in $G$ with a minimum number of edges such that the two nodes of each pair in $S$ are connected by a path in $F$.

### 3 Approximation Algorithms

In this section, we first describe an approximation algorithm for ULGT (see Section 3.1) and prove it to have a constant approximation ratio (see Section 3.2). Note that a constant approximation factor algorithm for DLGT can be obtained by a slight adaptation of this algorithm. For a description of this, we refer the reader to the full version [32] due to space constraints.

As an ingredient our algorithm uses a 2-approximation algorithm (see Section 1.2) for the Undirected Steiner Forest Problem (USF) defined as follows: Given a graph $G$ and a set $S$ of pairs of nodes from $G$, find a forest $F$ in $G$ with a minimum number of edges such that the two nodes of each pair in $S$ are connected by a path in $F$.

#### 3.1 Algorithm Description

For an initial graph $G_s = (V, E_s)$ and a final graph $G_t = (V, E_t)$, we define the set of additional edges $E_\Sigma := E_t \setminus E_s$ and the set of excess edges $E_\Omega := E_s \setminus E_t$. We now describe the algorithm in detail and then summarize its pseudo-code in Algorithm 1. Our algorithm consists of two parts, the first of which dealing with establishing all additional edges and the second of which dealing with removing all excess edges.
Algorithm 1 Approximation algorithm for ULGT.

Input: Initial graph $G_s$ and final graph $G_t$.

First part:
1: Compute a 2-approximate solution $F_{ALG,\oplus}$ for the USF with input $G_s$, and the set $E_{\oplus}$ as the set of pairs of nodes.
2: For each tree $T$ in $F_{ALG,\oplus}$, select a root node $r_T$ and connect all nodes in $T$ that are incident to an edge in $E_{\oplus}$ with $r_T$ (details below).
3: For each $\{u, v\} \in E_{\oplus}$, the root of the tree $u$ and $v$ belong to applies the Introduction primitive to create the edge $\{u, v\}$.
4: For each tree $T$ in $F_{ALG,\oplus}$, delegate all superfluous edges (i.e., not belonging to $G_s$ or $E_{\oplus}$) created during Step 2 bottom up in $T$ rooted at $r_T$, starting with the lowest level. At each intermediate node fuse all of these edges before delegating them to the next predecessor.

Second part:
5: Compute a 2-approximate solution $F_{ALG,\ominus}$ for the USF with input $G_t$, and the set $E_{\ominus}$ as the set of pairs of nodes.
6: For each $e \in E_{\ominus}$, let $s(e)$ be an arbitrary of the two endpoints of $e$. For each tree $T$ in $F_{ALG,\ominus}$, select a root node $r_T$ and for each $e \in E_{\ominus}$ whose endpoints belong to $T$, connect $s(e)$ with $r_T$ (similar to Step 2, details below).
7: For each $e \in E_{\ominus}$, $s(e)$ delegates the other endpoint to $r_T$.
8: For each tree $T$ in $F_{ALG,\ominus}$, delegate all superfluous edges bottom-up and fuse multiple edges as in Step 4.

In the first part, using an arbitrary 2-approximation algorithm for the USF as a black box the algorithm computes a 2-approximate solution to the following USF instance: The given graph is $G_s$, and the set of pairs of nodes is $E_{\ominus}$. Note that the result is a forest such that for every edge $\{u, v\} \in E_{\ominus}$, $u$ and $v$ belong to the same tree. For each tree $T$ in this forest the algorithm then selects an arbitrary root $r_T$ and connects all nodes in $T$ that are incident to an edge in $E_{\ominus}$ to $r_T$. The exact details of this will be described when we analyze the length of the resulting computation. In the next step, for every $T$, for every $\{u, v\} \in E_{\ominus}$ such that $u$ and $v$ belong to $T$, $r_T$ introduces $u$ to $v$ to each other, thereby creating the edge $\{u, v\}$. After that, the superfluous edges are deleted in a bottom-up fashion: every node that does not have a descendant with a superfluous edge (in the tree $T$ this node belongs to when viewing this tree as rooted by $r_T$), fuses all superfluous edges and delegates the last such to its predecessor in the tree. Note that all superfluous edges in the same tree $T$ have $r_T$ as one of their endpoints.

The second part of the algorithm is similar to the first, with the following differences: In the fifth step, the USF is approximated for the graph $G_t$ and $E_{\ominus}$ as the set of pairs. Note that the solution is a subgraph of the graph obtained after the first part of the algorithm. In the sixth step, only one of the two endpoints of an edge from $E_{\ominus}$ is selected to become connected with the root of the tree the endpoints belong to. In the seventh step (where in the first part the additional edges are created by the $r_T$ nodes), for each edge $e \in E_{\ominus}$, the endpoint selected in the sixth step delegates this edge to $r_T$ (resulting in the edge $\{r_T, v\}$).
3.2 Analysis

In this section we show that Algorithm 1 is a constant-approximation algorithm for ULGT, which is formalized by the following theorem:

▶ Theorem 7. ULGT ∈ APX.

For convenience we will analyze the two parts of the algorithm individually. Therefore, for a given initial graph $G_s$ and final graph $G_t$, let $ALG_1(G_s,G_t)$ be the length of the computation of the first part of the algorithm for this instance, $ALG_2(G_s,G_t)$ be the length of the computation of the second part, and $ALG(G_s,G_t) := ALG_1(G_s,G_t) + ALG_2(G_s,G_t)$. Furthermore, let $OPT(G_s,G_T)$ be the length of an optimal solution to ULGT for initial graph $G_s$ and final graph $G_t$. We also define the intermediate graph $G' = (V,E_s \cup E_\oplus)$. In the course of the analysis we will establish a relationship between $ALG_1(G_s,G_t)$ and $OPT(G_s,G_t)$ and between $ALG_2(G_s,G_t)$ and $OPT(G',G_t)$. This will aid us in determining the approximation factor of Algorithm 1 due to the following lemma:

▶ Lemma 8. $OPT(G_s,G') + OPT(G',G_t) \leq 2OPT(G_s,G_t) + |E_\oplus|$.

Proof. Let $\mathcal{P}$ denote the problem equal to $k$-ULGT with initial graph $G_s$ and final graph $G_t$ with the additional requirement that the computation must contain $G'$ and let $OPT'(G_s,G_t)$ be the length of an optimal solution to it. Clearly, $OPT(G_s,G') + OPT(G',G_t) \leq OPT'(G_s,G_t)$ (otherwise, split the computation at $G'$ and improve either $OPT(G_s,G')$ by the first part obtained or $OPT(G',G_t)$ by the second part obtained). We now show that $OPT'(G_s,G_t) \leq 2OPT(G_s,G_t) + |E_\oplus|$.

Consider a computation $C$ whose initial graph is $G_s$, whose final graph is $G_t$ and whose length is $OPT(G_s,G_t)$ (note that such a computation is an optimal solution to ULGT). We now transform $C$ into a computation that represents a solution to $\mathcal{P}$. This transformation increases its length by only $OPT(G_s,G_t) + |E_\oplus|$ and thus proves the above claim (recall that any solution to $\mathcal{P}$ has at least the size of an optimal solution to it). First, because the final graph does not contain any edge $\{u,v\} \in E_\oplus$, for every such edge there is one last Delegation in $C$ that removes this edge (recall Observation 1). We replace each of these last delegations by an introduction and obtain a new computation $C'$ of equal length. Note that changing these delegations to introductions does not make the computation infeasible as this only causes the graph to have additional edges. The final graph of $C'$ is $(V,E_s \cup E_\oplus) = (V,E_s \cup E_\oplus) = G'$ (recall that $E_\oplus = (E_s \cup E_\oplus) \setminus E_\oplus$). Next we append $C'$ by $C$ and obtain the computation $C''$ of length $2OPT(G_s,G_t)$. Note that since $C$ transformed $G_s$ to $G_t$, this second half of $C''$, which starts from $G' = (V,E_s \cup E_\oplus)$, has the final graph $G'' = (V,E_t \cup E_\oplus)$, i.e., each edge from $E_\oplus$ appears twice in $G''$. Thus we extend $C''$ by fusing each edge from $E_\oplus$ with its double, resulting in a computation $C'''$ of length $2OPT(G_s,G_t) + |E_\oplus|$. Since $C'''$ represents a solution to $\mathcal{P}$ for initial graph $G_s$ and final graph $G_t$, this completes the proof.

In the rest of the analysis we show that $ALG_1(G_s,G_t) \leq 11OPT(G_s,G')$ (Lemma 9) and that $ALG_2(G_s,G_t) \leq 7OPT(G',G_t)$ (Lemma 10). By Lemma 8 this implies that $ALG(G_s,G_t) \leq 11(2OPT(G_s,G_t) + |E_\oplus|) \leq 33OPT(G_s,G_t)$ (since, clearly, $OPT(G_s,G_t) \geq |E_\oplus|$), which yields the claim of Theorem 7.

We begin with the former claim, which is formalized by the following lemma:

▶ Lemma 9. $ALG_1(G_s,G_t) \leq 11OPT(G_s,G')$.

Proof. Let $F_{OPT,\oplus}$ be an optimal solution for the USF with input $G_s$ and $E_\oplus$ as the set of nodes and recall that $F_{ALG,\oplus}$ is the USF approximation computed in Step 1 of Algorithm 1. Throughout the analysis, $|F_{OPT,\oplus}|$ and $|F_{ALG,\oplus}|$ will denote the number of edges in these
solutions. In the first part of this proof, we show that $ALG_1(G_s, G_t) \leq 4|OPT,\oplus| + 3|E_{\oplus}|$. The second part then consists in proving $OPT(G_s, G') \geq |OPT,\oplus| - |E_{\oplus}|$, which together with the observation that $OPT(G_s, G') \geq |E_{\oplus}|$ yields the claim.

To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves Fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $ST(x)$ consists of $x, w, u, v$ and $u, x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.

Figure 3 Example of a tree $T$ with root $r_T$ for Step 2-4 of Algorithm 1 assuming $\{u, v\} \in E_{\oplus}$. To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves Fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $ST(x)$ consists of $x, w, u, v$ and $u, x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.

To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves Fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $ST(x)$ consists of $x, w, u, v$ and $u, x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.

To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves Fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $ST(x)$ consists of $x, w, u, v$ and $u, x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.

To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves Fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $ST(x)$ consists of $x, w, u, v$ and $u, x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.

To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves Fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $ST(x)$ consists of $x, w, u, v$ and $u, x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.

To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves Fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $ST(x)$ consists of $x, w, u, v$ and $u, x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.
by \((x, w, y, y)\) and note that \(P_{u,v}^1\) exists in \(G_i\). Eventually, we obtain a path \(P_{u,v}^1\) that exists in \(G_s\). For \(i \in \{1, \ldots, L\}\), let \(F^i := \bigcup_{(u,v) \in E_\oplus} E(P_{u,v}^i)\) (where \(E(P)\) is the set of all edges on the path \(P\)) and note that \(F^i\) represents a solution to the USF with input \(G_s\) and \(E_\oplus\) as the set of node pairs. An example is given in Figure 4. For an arbitrary \(i \in \{1, \ldots, L - 1\}\), note that \(|F^i| \leq |F^{i+1}| + 1\): if \(G_{i+1}\) was obtained from \(G_i\) by the application of a Fusion primitive, this inequality trivially holds as none of the above paths changes in this case. Otherwise, \(G_{i+1}\) was obtained from \(G_i\) by an application of an Introduction or Delegation primitive by some node \(w\) causing at most one edge \(\{x, y\}\) to exist in \(G_{i+1}\) that does not exist in \(G_i\). In this case, we further know that \(\{w, x\}\) and \(\{w, y\}\) exist in \(G_i\) and by the definition of the above paths, for every pair \(\{u, v\}\) such that \(P_{u,v}^i\) contains the edge \(\{x, y\}\) the path \(P_{u,v}^i\) contains \((x, w, y)\) as a sub-path instead and for all other pairs \(\{u', v'\}\), \(P_{u',v'}^i = P_{u',v'}^{i+1}\). By the definition of \(F^i\) and \(F^{i+1}\), this implies \(|F^i| \leq |F^{i+1}| + 1\) also in this case. All in all we obtain that \(|F^1| \leq |F^L| + L = |E_\oplus| + L\) because \(F^L = E_\oplus\) (note the definition of \(F^L\)). By the assumption that \(L < |F_{OPT,\oplus}|-|E_\oplus|\), we obtain \(|F^1| < |F_{OPT,\oplus}|\), which represents a contradiction. □

**Lemma 10.** \(ALG_2(G_s, G_t) \leq 7OPT(G', G_t)\).

*Proof.* The general structure of this proof follows the line of the proof of Lemma 9, but differs in the details. Similar to the notation used in that proof, let \(F_{OPT,\oplus}\) be an optimal solution for the USF with input \(G_s\) and \(E_\oplus\) as the set of nodes and recall that \(F_{ALG,\oplus}\) is the USF approximation computed in Step 5 of Algorithm 1. Analogously, \(|F_{OPT,\oplus}|\) and \(|F_{ALG,\oplus}|\) denote the number of edges in these solutions. In the first part of this proof, we show that \(ALG_2(G_s, G_t) \leq 4|F_{OPT,\oplus}| + 3|E_\oplus|\). The second part then consists in proving \(OPT(G', G_t) \geq |F_{OPT,\oplus}|\), which together with the observation that \(OPT(G', G_t) \geq |E_\oplus|\) yields the claim.

To upper bound \(ALG_2(G_s, G_t)\), we analyze the number of primitives applied in each step of the second part of the approximation algorithm. Of course, no primitive is applied in Step 5. The connections required in Step 6 can be created in a similar fashion as in Step 2 (see the proof of Lemma 9: For each tree \(T\), we proceed top-down in the \(T\) rooted at \(r_T\) again. Here, each intermediate node \(u\) checks whether \(u = s(e)\) for some \(e \in E_\oplus\). If so, it introduces \(r_T\) to all relevant children (here a node \(v\) is *relevant* if \(ST(v)\) contains a node \(w\) such that \(w = s(e')\) for some \(e' \in E_\oplus\)). Otherwise, it introduces \(r_T\) to all but one relevant children and delegates
it to the remaining one. In the end, for every edge $e \in E_\Theta$, $s(e)$ has an edge to $r_T$, the number of edges in the graph has increased by at most $|E_\Theta|$, and the process involved at most $|F_{ALG,\Theta}|$ applications of primitives. In Step 7, clearly exactly $|E_\Theta|$ edges have to be delegated. Step 8 is similar to Step 4 and for analogous reasons requires at most $|F_{ALG,\Theta}|$ delegations and at most $2|E_\Theta|$ fusions (recall that up to $|E_\Theta|$ edges were added in Step 6 and the edges delegated in Step 7 have to be removed as well). All in all, Step 6, Step 7 and Step 8 of the algorithm involve at most $|F_{ALG,\Theta}|$, $|E_\Theta|$ and $|F_{ALG,\Theta}| + 2|E_\Theta|$ applications of primitives, respectively, which yields: $ALG_2(G_s, G_t) \leq 2|F_{ALG,\Theta}| + 3|E_\Theta| \leq 4|F_{OPT,\Theta}| + 3|E_\Theta|$ (since $F_{ALG,\Theta}$ is a 2-approximation of $F_{OPT,\Theta}$).

To lower bound the value of $OPT(G', G_t)$, assume for contradiction that there is a computation $C$ with initial graph $G'$ and final graph $G_s$ of length $L < |F_{OPT,\Theta}| - |E_\Theta|$. Let $G_s = G_1 \Rightarrow G_2 \Rightarrow \cdots \Rightarrow G_L$ be the sequence of graphs of this computation. Similar to the proof of Lemma 9, for every $(u, v) \in E_\Theta$, we create a path from $u$ to $v$, but this time we start with $F_{u,v}^1 := (u, v)$ and consider the graphs in increasing order: For $i \in \{2, \ldots, L\}$, if $P_{u,v}^{i-1}$ exists in $G_i$, $P_{u,v}^i := P_{u,v}^{i-1}$. Otherwise since $G_i$ is the result of a single application of a primitive to $G_{i-1}$, there is exactly one edge $\{x, y\}$ in $P_{u,v}^{i-1}$ that exists in $G_{i-1}$ but not in $G_i$, and this edge must have been delegated by either $x$ or $y$ to some node $w$. In the following denote the node that applied the delegation by $z$ and denote by $\bar{z}$ the other node from $\{x, y\}$. In $G_{i-1}$, $z$ must share an edge with $w$ and this edge still exists in $G_i$ (for only one primitive is applied in the transition from $G_{i-1}$ to $G_i$). Since $\{z, \bar{z}\}$ was delegated to $w$, in $G_i$ the edge $\{w, \bar{z}\}$ exists in $G_i$. Thus, let $P_{u,v}^i$ be $P_{u,v}^{i-1}$ with $(x, y)$ replaced by $(x, w, y)$ and observe that $P_{u,v}^i$ exists in $G_i$. Eventually, we obtain a path $P_{u,v}^L$ that exists in $G_t$. Define $F^1_s := \bigcup_{(u,v) \in E_\Theta} E(P_{u,v}_s)$ (where $E(P)$ is the set of all edges on the path $P$) for all $i \in \{1, \ldots, L\}$, and note that $F^L$ represents a solution to the USP with input $G_t$ and $E_\Theta$ as the set of nodes. Furthermore, for an arbitrary $i \in \{1, \ldots, L - 1\}$, note that $|F^{i+1}| \leq |F^i| + 1$ because there is at most one edge $\{x, y\}$ that exists in $G_i$ but not in $G_{i+1}$ and thus causes the replacement of $(x, y)$ by $(x, w, y)$ for some fixed node $w$ for all paths that contain $(x, y)$ as a sub-path. This yields that $|F^L| \leq |F^1| + L = |E_\Theta| + L$ because $F^1 = E_\Theta$ (note the definition of $F^1$). By the assumption that $L < |F_{OPT,\Theta}| - |E_\Theta|$, we obtain $|F^L| < |F_{OPT,\Theta}|$, which represents a contradiction.

\section{Conclusion}

We proposed a set of primitives for topology adaptation that a server may use to adapt the network topology into any desired (weakly) connected state but at the same time cannot use to disconnect the network or to introduce new nodes into the system. So far, we only assumed that the server could act maliciously but that the participants of the network are honest and correct, i.e., they refuse any graph transformation commands beyond the four primitives. What, however, if some participants also behave in a malicious manner? Is it still possible to avoid Eclipse or Sybil attacks? It seems that in this case the only measure that would help is to form quorums of nodes that are sufficiently large so that at least one node in each quorum is honest.

Besides these security-related aspects, our results give rise to additional questions: For example, does the NP-hardness apply to any set of local primitives, or is there a set of local primitives that can transform arbitrary initial graphs much faster into arbitrary final graphs than the set considered in this work? Furthermore, is it possible to obtain decentralized versions of the algorithms presented in Section 3, and, if so, what is their competitiveness when compared to the centralized ones?
References


On the Complexity of Local Graph Transformations


