Approximation and Hardness of Token Swapping

Tillmann Miltzow*[1], Lothar Narins[2], Yoshio Okamoto†[3], Günter Rote[4], Antonis Thomas[5], and Takeaki Uno[6]

1 **Freie Universität Berlin, Berlin, Germany**
2 **Freie Universität Berlin, Berlin, Germany**
3 **University of Electro-Communications, Tokyo, Japan**
4 **Freie Universität Berlin, Berlin, Germany**
5 **Department of Computer Science, ETH Zürich, Zürich, Switzerland**
6 **National Institute of Informatics, Tokyo, Japan**

Abstract

Given a graph $G = (V, E)$ with $V = \{1, \ldots, n\}$, we place on every vertex a token $T_1, \ldots, T_n$. A swap is an exchange of tokens on adjacent vertices. We consider the algorithmic question of finding a shortest sequence of swaps such that token T_i is on vertex i. We are able to achieve essentially matching upper and lower bounds, for exact algorithms and approximation algorithms. For exact algorithms, we rule out any $2^{o(n)}$ algorithm under the ETH. This is matched with a simple $2^{O(n \log n)}$ algorithm based on a breadth-first search in an auxiliary graph. We show one general 4-approximation and show APX-hardness. Thus, there is a small constant $\delta > 1$ such that every polynomial time approximation algorithm has approximation factor at least δ.

Our results also hold for a generalized version, where tokens and vertices are colored. In this generalized version each token must go to a vertex with the same color.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases token swapping, minimum generator sequence, graph theory, NP-hardness, approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.66

1 Introduction

In the theory of computation, we regularly encounter the following type of problem: Given two configurations, we wish to transform one to the other. In these problems we also need to fix a family of operations that we are allowed to perform. Then, we need to solve two problems: (1) Determine if one can be transformed to the other; (2) If so, find a shortest sequence of such operations. Motivations come from the better understanding of solution spaces, which is beneficial for design of local-search algorithms, enumeration, and probabilistic analysis. See [9] for example. The study of *combinatorial reconfigurations* is a young growing field [2].

Among problem variants in combinatorial reconfiguration, we study the *token swapping problem* on a graph. The problem is defined as follows. We are given an undirected connected graph with n vertices $v_1, \ldots, v_n$. Each vertex v_i holds exactly one token $T_{\pi(i)}$, where π is a permutation of $\{1, \ldots, n\}$. In one step, we are allowed to swap tokens on a pair of adjacent

* Partially supported by the ERC grant PARAMTIGHT: "Parameterized complexity and the search for tight complexity results", no. 280152.
† Partially supported by MEXT/JSPS KAKENHI Grant Numbers 24106005, 24700008, 24220003 and 15K00009, and JST, CREST, Foundation of Innovative Algorithms for Big Data.

vertices, that is, if v_i and v_j are adjacent, v_i holds the token T_k, and v_j holds the token T_ℓ, then the swap between v_i and v_j results in the configuration where v_i holds T_ℓ, v_j holds T_k, and all the other tokens stay in place. Our objective is to determine the minimum number of swaps so that every vertex v_i holds the token T_i. It is known (and not difficult to observe) that such a sequence of swaps always exists [22].

The problem was introduced by Yamanaka et al. [22] in its full generality, but special cases had been studied before. When the graph is a path, the problem is equivalent to counting the number of adjacent swaps in bubble sort, and it is folklore that this is exactly the number of inversions of π (see Knuth [16, Section 5.2.2]). When the graph is complete, it was already known by Cayley [4] that the minimum number is equal to n minus the number of cycles in π (see also [14]). Note that the number of inversions and the number of cycles can be computed in $O(n \log n)$ time. Thus, the minimum number of swaps can be computed in $O(n \log n)$ time for paths and complete graphs. Jerrum [14] gave an $O(n^2)$-time algorithm to solve the problem for cycles. When the graph is a star, a result by Pak [19] implies an $O(n \log n)$-time algorithm. Exact polynomial-time algorithms are also known for complete bipartite graphs [22] and complete split graphs [24]. Polynomial-time approximation algorithms are known for trees with factor two [22] and for squares of paths with factor two [12]. Since Yamanaka et al. [22], it has remained open whether the problem is polynomial-time solvable or NP-complete, even for general graphs, and whether there exists a constant-factor polynomial-time approximation algorithm for general graphs.

Our results. In this paper, we resolve the open problems above from Yamanaka et al. [22]. Namely, we prove that the token swapping problem is NP-complete, and give a polynomial-time approximation algorithm with factor 4 for general graphs and factor 2 for trees. For the NP-completeness, we consider a more general problem, called the *colored token swapping problem*. In the colored token swapping problem, each token has a color, each vertex of the graph has a color, and we are asked to move the tokens to the vertices of the same color. This can also be seen as a graph-theoretic generalization of bubble sort on a multiset. Yamanaka et al. [23] proved that the colored token swapping problem is NP-complete. We strengthen their result in the sense that our hardness results hold for instances with special structure. Furthermore we design a gadget called an *even permutation* network (not to be confused with a sorting network). This can be regarded as a special class of instances of the token swapping problem. It allows us to achieve *any* even permutation of the tokens between dedicated input and output vertices. Using even permutation networks allows us to reduce further from the colored token swapping problem to the more specific uncolored version. We believe that this gadget is of general interest and we hope that similar gadgets will prove useful in other situations too. A close look at the hardness proof gives APX-hardness. Therefore we have that the constant approximation, we provide, is essentially tight (up to the constant). In addition, the proof rules out any $2^{o(n)}$-time algorithm under the Exponential Time Hypothesis, where n is the number of vertices of the input graph. To complement this, we provide a simple $2^{O(n \log n)}$-time exact algorithm. Therefore, our algorithmic results are almost *tight*. Even though our exact algorithm is completely straightforward, our reductions show that we cannot hope for much better results. The approximation algorithm we suggest makes deep use of the structure of the problem and is nice to present. Our reductions are quite technical and, so, we try to modularize them as much as possible. In the process, we show hardness for a number of intermediate problems. Due to space limitations, some of the proofs are missing. A self-contained full version of this paper can be found online [17].

Organization. In Section 2, we provide a simple exact algorithm. Section 3 describes a 4-approximation algorithm of the token swapping problem. In Section 4, we show a general technique to attain approximation algorithms for the colored token swapping problem from the uncolored version. In this way, we attain a 4-approximation for the colored token swapping problem. In Section 5, we define and construct even permutation networks, which are interesting in their own right. The hardness results are modularized into four sections. In Section 6, we show a reduction from 3SAT to a problem of finding disjoint paths in a structured graph. (A precise definition can be found in the section.) Section 7 shows how to reduce further to the colored token swapping problem. Section 8 is committed to the reduction from the colored to the ordinary token swapping problem. For this purpose, we attach an even permutation network to each color class. Section 9 finally puts the three previous reductions together in order to attain the main results.

Related concepts. The famous 15-puzzle is similar to the token swapping problem, and indeed a graph-theoretic generalization of the 15-puzzle was studied by Wilson [21]. The 15-puzzle can be modeled as a token-swapping problem by regarding the empty square of the 4×4 grid as a distinguished token, but there remains an important difference from our problem: in the 15-puzzle, two adjacent tokens can be swapped only if one of them is the distinguished token. For the 15-puzzle and its generalization, the reachability question is not trivial, but by the result of Wilson [21] we can determine if the tokens can be moved to the right vertices in polynomial time. However, it is NP-hard to find the minimum number of swaps even for grids [20].

The token swapping problem can be seen as an instantiation of the minimum generator sequence problem of permutation groups. There, a permutation group is given by a set of generators $\pi_1, \ldots, \pi_k$, and we want to find a shortest sequence of generators whose composition is equal to a given target permutation τ. Indeed, this is the problem that Jerrum [14] studied, where he gave an $O(n^2)$-time algorithm for the token swapping problem on cycles. He proved that the minimum generator sequence problem is PSPACE-complete [14]. To formulate the token swapping problem as the minimum generator sequence problem of permutation groups, for a given connected graph G, we consider the symmetric group on $\{1, \ldots, n\}$ (the set of all permutations on $\{1, \ldots, n\}$) that is given by the set of transpositions determined by the edges of G.

In the literature, we also find the problem of *token sliding*, but this is different from the token swapping problem. In the token sliding problem on a graph G, we are given two independent sets I_1 and I_2 of G of the same size. We place one token on each vertex of I_1, and we perfom a sequence of the following sliding operations: We may move a token on a vertex v to another vertex u if v and u are adjacent, u has no token, and after the movement the set of vertices with tokens forms an independent set of G. The goal is to determine if a sequence of sliding operations can move the tokens on I_1 to I_2. The problem was introduced by Hearn and Demaine [11], and they proved that the problem is PSPACE-complete even for planar graphs of maximum degree three. Subsequent research showed that the token sliding problem is PSPACE-complete for perfect graphs [15] and graphs of bounded treewidth [18]. Polynomial-time algorithms are known for cographs [15], claw-free graphs [3], trees [6] and bipartite permutation graphs [8].

Several other models of swapping have been studied in the literature, e.g. [10, 7, 5].

2 Simple Exact Algorithms

We start presenting our results with the simplest one. There is an exact, exponential time algorithm which, after the hardness results that we obtain in Section 9, will prove to be almost tight to the lower bounds.

▶ **Theorem 1.** *Let I be an instance of the token swapping problem on a graph $G = (V, E)$ with n vertices and m edges. There exists a simple algorithm for finding an optimal sequence in time $O(n!\, m) = 2^{O(n \log n)}$.*

Proof. The algorithm is breadth first search in the configuration graph. The vertices $\mathcal{V}$ of the configuration graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consist of all $n! = 2^{O(n \log n)}$ possible configurations of tokens on vertices V of G. Two configurations A and B are adjacent if there is a single swap that transforms A to B. Each configuration has $m \leq \binom{n}{2} = O(n^2)$ adjacent configurations. Thus the total number of edges is $n!\ m/2 \leq O(n^2) \cdot 2^{O(n \log n)} = 2^{O(n \log n)}$. It is easy to see that any shortest path in $\mathcal{G}$ from the start to the target configuration gives a shortest sequence of swaps. Breadth first search finds this shortest path. The running time of breadth first search is $O(|\mathcal{V}| + |\mathcal{E}|) = 2^{O(n \log n)}$. ◀

3 A 4-Approximation Algorithm

In this section, we describe an approximation algorithm for the token swapping problem, which has approximation factor 4 on general graphs and 2 on trees. Firstly, we state the following simple lemma.

▶ **Lemma 2.** *Let $d(T_i)$ be the distance of token T_i to the target vertex i. Let L be the sum of distances of all tokens to their target vertices:*

$$L := \sum_{i=1}^{n} d(T_i).$$

Then any solution needs at least $L/2$ swaps.

Proof. Every swap reduces L by at most 2. ◀

We are now ready to describe our algorithm. It has two atomic operations. The first one is called an *unhappy swap*: This is an edge swap where one of the tokens swapped is already on its target and the other token reduces its distance to its target vertex (by one).

The second operation is called a *happy swap chain*. Consider a path of $\ell + 1$ distinct vertices $v_1, \ldots, v_{\ell+1}$. We swap the tokens over edge (v_1, v_2), then (v_2, v_3), etc., performing ℓ swaps in total. The result is that the token that was on vertex v_1 is now on vertex $v_{\ell+1}$ and all other tokens have moved from v_j to v_{j-1}. If every swapped token reduces its distance by at least 1, we call this a happy swap chain of length ℓ. A single swap that is part of a happy swap chain is called a *happy swap*. When our algorithm applies a happy swap chain, there will be an edge between $v_{\ell+1}$ and v_1, closing a cycle. In this case, the happy swap chain performs a cyclic shift. Figure 1 illustrates this definition with an example. Note that a happy swap chain might consist of a single swap. We leave it to the reader to find such an example.

▶ **Lemma 3.** *Let G an undirected graph $G = (V, E)$ with a token placement where not every token is at its target vertex. Then, there is a happy swap chain or an unhappy swap.*

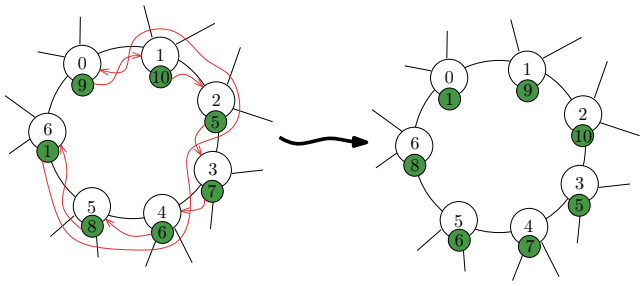

Figure 1 Before and after a happy swap chain. The swap sequence is, in this order, $6-5$, $5-4$, $4-3$, $3-2$, $2-1$, $1-0$. Token 1 swaps with every other token, moving counter-clockwise; every other token moves one step clockwise.

Proof. Given a token placement on a graph $G = (V, E)$ we define the auxiliary directed graph F on V as follows. For each undirected edge $e = \{v, w\}$ of G we define a directed edge from v to w if the token on v reduces its distance to its target vertex by swapping along e. Note that for a pair of vertices both directed edges might be part of the graph F. We can perform a happy swap chain whenever we find a directed cycle in F. The outdegree of a vertex v in F is 0 if and only if the token on v has target vertex v. Assume that not every token is in its target position. Choose any vertex v that does not hold the right token and perform a depth first search from v in F. This search ends with either revisiting a vertex and we get a directed cycle or we encounter a vertex with outdegree 0 and we get an unhappy swap. ◀

The above lemma gives rise to our algorithm: Search for a happy swap chain or unhappy swap; when one is found it is performed, until none remains. If there is no such swap, the final placement of every token is reached. This algorithm is polynomial time (follows from the proof of Lemma 3). Moreover, it correctly swaps the tokens to their target position with at most $2L$ swaps.

▶ **Lemma 4.** *Let T_i be a token on vertex i. If T_i participates in an unhappy swap, then the next swap involving T_i will be a happy swap.*

Proof. Refer to Figure 2. Let the vertices i, j be the ones participating in the unhappy swap and let e be the edge that connects them. On vertex j is token T_i that got unhappily removed from its target vertex and on vertex j is token S' whose target is neither i nor j. Based on Lemma 3, our algorithm performs either unhappy swaps or happy swap chains. Note that, currently, edge e cannot participate in an unhappy swap. This is because none of its endpoints holds the right token. Moreover, token T_i cannot participate in an unhappy swap that does not involve edge e, as that would not decrease its distance. Therefore, there has to be a happy swap chain that involves token T_i. ◀

▶ **Theorem 5.** *Any sequence of happy swap chains and unhappy swaps is at most 4 times as long as an optimal sequence of swaps on general graphs and 2 times as long on trees.*

Proof. Let L be the sum of all distances of tokens to its target vertex. We know that the optimal solution needs at least $L/2$ swaps as every swap reduces L by at most 2 (Lemma 2). We will show that our algorithm needs at most $2L$ swaps, and this implies the claim.

A happy swap chain of length ℓ reduces the sum of total distances by $\ell + 1$ and involves ℓ happy swaps. Thus, #(happy swaps) $< L$. By Lemma 4, #(unhappy swaps) $\leq$

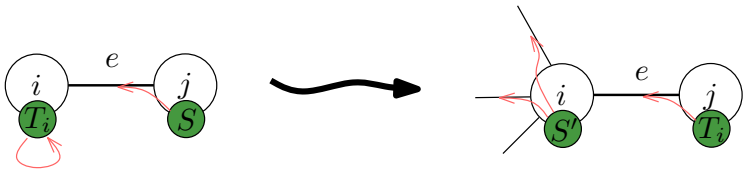

Figure 2 After a Token T_i makes an unhappy swap along edge e, T_i wants to go back to vertex i and is not willing to go to any other vertex. Also whatever token S' will be on vertex i, it has no desire to stay there. This implies that the next swap involving T_i will be part of a happy swap chain.

#(happy swaps), this implies: #(swaps) = #(unhappy swaps) + #(happy swaps) ≤ 2 #(happy swaps) < $2L$. This algorithm is a 2-approximation algorithm on trees as the longest possible cycle in F (as in Lemma 3) has length 2 and thus every happy swap chain consists of only one happy swap that reduces L by *two*. This implies #(happy swaps) = $L/2$. ◀

4 The Colored Version of the Problem

In this section, we consider the *colored* token swapping problem as defined in the introduction. We will see that all approximation bounds from the previous sections carry over to this problem. Our approach to the problem is easy:

1. We first decide which token goes to which target vertex.
2. We then apply one of the algorithms from the previous sections for n distinct tokens.

The details can be found in the full version.

▶ **Theorem 6.** *There is a 4-approximation of the colored token swapping problem on general graphs and a 2-approximation on trees.*

5 Even Permutation Networks

Before we dive in the details of the reductions for token swapping (Sections 6–8), we introduce our even permutation network PN, the gadget that we advertised in the introduction. We consider it stand-alone and we present it independently of the other parts of the reduction.

The vertices of PN are partitioned into the sets V_{in}, V_{out}, and V_{PN}, where $|V_{\text{in}}| = |V_{\text{out}}|$, together with a bijection $\varphi\colon V_{\text{in}} \to V_{\text{out}}$ and a bijection $\psi\colon V_{\text{out}} \cup V_{\text{PN}} \to V_{\text{in}} \cup V_{\text{PN}}$, defining for each token on $V_{\text{PN}} \cup V_{\text{out}}$ a fixed target vertex in $V_{\text{in}} \cup V_{\text{PN}}$. We place on each $v \in V_{\text{out}} \cup V_{\text{PN}}$ the token $T_{\psi(v)}$. (The goal of token T_v is to be on vertex v.)

The target vertices of V_{in} lie in V_{out}, and they are specified by an additional variable permutation π of V_{in}. The instance $I(\pi)$ corresponding to π is obtained by placing on each $v \in V_{\text{in}}$ the token $T_{\varphi(\pi(v))}$.

This *even* permutation network has the property that the optimal number of swaps to solve $I(\pi)$ is the same for every even permutation π, see Figure 3. We will refer to the composition $\varphi \circ \pi$ as an *assignment*. Recall that a permutation is even if the number of inversions of the permutation is even. Realizing *all* permutations at the same cost is impossible, since every swap changes the parity, and hence all permutations reachable by a given number of swaps have the same parity.

We will later use even permutation networks to reduce from the *colored* token swapping problem on a network H to the token swapping problem, see Lemma 10. We will attach an even permutation network PN to each layer (color class) of H by identifying that layer with the input vertices V_{in} of PN. This permutation network will bring the colored tokens of H,

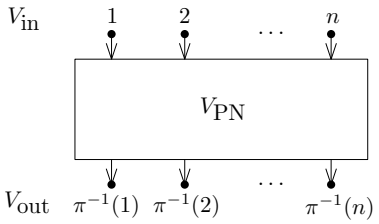

Figure 3 The interface of a permutation network PN. The permutation φ is not indicated; it maps each vertex of V_{in} to the vertex of V_{out} of that is vertically below.

which have arrived in V_{in} in a first phase, to their individual final destinations in V_{out}. To ensure that even permutations suffice, we use the simple trick of doubling the whole input graph before attaching the permutation networks: the product of two permutations of the same parity is always even.

▶ **Lemma 7.** *For every n, there is a permutation network* PN *with n input vertices V_{in}, n output vertices V_{out}, and $O(n^3)$ additional vertices V_{PN}, which has the following properties, for some value T:*

- *For every target assignment π between the inputs V_{in} and the outputs V_{out} that is an even permutation, the shortest swapping sequence has length T.*
- *For any other target assignment, the shortest realizing sequence has length at least $T + 1$.*
- *The same statement holds for any extension of the network PN, which is the union of PN with another graph H that shares only the vertices V_{in} with PN, and in which the starting positions of the tokens assigned to V_{out} may be anywhere in H.*

The last clause concerns not only all assignments between V_{in} and V_{out} that are odd permutations, but also all other conceivable situations where the tokens destined for V_{out} do not end up in V_{in}, but somewhere else in the graph H. The lemma confirms that such non-optimal solutions for H cannot be combined with solutions for PN to yield better swapping sequences than the ones for which the network was designed.

Proof. (Sketch. For the complete proof with diagrams of the gadgets, see the full version.)

The even permutation network will be built up hierarchically from small gadgets. Each gadget is built in a layered manner, subject to the following rules.

1. There is a strict layer structure: The vertices are partitioned into layers $V_1, \ldots, V_t$ of the same size.
2. Each gadget has its own input layer V_{in} at the top and its output layer V_{out} at the bottom, just as the overall network PN.
3. Edges may run between two vertices of the same layer (*horizontal edges*), or between adjacent layers (*downward edges*).
4. Every vertex has at most one neighbor in the successor layer and at most one neighbor in the predecessor layer.

The goal is to bring the input tokens from the input layer V_{in} to the output layer V_{out}. By Rule 3, the cheapest conceivable way to achieve this is by using only the downward edges, and then every such edge is used precisely once. Our first gadget is the so-called *swapping gadget*, which can either realize the identity permutation, or swap a specified input vertex with one of two others at a cost of one additional swap. Our second gadget, the *shift gadget*, consists of two swapping gadgets chained together in such a way that both the identity permutation and a cyclic shift of three input vertices can be realized in the same minimal number of swaps, but every other permutation takes more swaps. Both of these constructions

involve some auxiliary input tokens that have set destinations within the gadget. We then chain these shifting gadgets together in a cascading fashion, so that any even permutation of the input vertices can be realized as a composition of the cyclic shifts of three vertices. In this way, all the even permutations can be achieved with the same amount of swaps, and all other permutations are more costly. ◀

6 Reduction to a Disjoint Paths Problem

For the lower bounds of the Token Swapping problem, we study some auxiliary problems. The first problem is a special multi-commodity flow problem.

Disjoint Paths on a Directed Acyclic Graph (DP)
Input: A directed acyclic graph $G = (V, E)$ and a bijection $\varphi\colon V^- \to V^+$ between the sources V^- (vertices without incoming arcs) and the sinks V^+ (vertices without outgoing arcs), with the following properties:

1. The vertices can be partitioned into layers $V_1, V_2, \ldots, V_t$ such that, for every vertex in some layer V_j, all incoming arcs (if any) come from the same layer V_i, with $i < j$. Note that i need not be the same for every vertex in V_j.
2. Every layer contains at most 10 vertices.
3. For every $v \in V^-$, there is a path from v to $\varphi(v)$ in G. Let $n(v)$ denote the number of vertices on the shortest path from v to $\varphi(v)$.
4. The total number of vertices is $|V| = \sum_{v \in V^-} n(v)$.

Question: Is there a set of vertex-disjoint directed paths $P_1, \ldots, P_k$ with $k = |V^-| = |V^+|$, such that P_i starts at some vertex $v \in V^-$ and ends at $\varphi(v)$?

By Property 4, the k paths must completely cover the vertices of the graph. The graphs that we will construct in our reduction have in fact the stronger property that *any* directed path from $v \in V^-$ to $\varphi(v)$ contains the same number $n(v)$ of vertices.

In our construction, we will label the source and the sink that should be connected by a path by the same symbol X, and "the path X" refers to this path. In the drawings, the arcs will be directed from top to bottom.

▶ **Lemma 8.** *There is a linear-size reduction from 3SAT to the Disjoint Paths Problem on a Directed Acyclic Graph (DP).*

Proof. Let $x_1, \ldots, x_n$ be the variables and $C_1, \ldots, C_m$ the clauses of the 3SAT formula. Each variable x_i is modeled by a variable path, which has a choice between two tracks. The track is determined by the choice of the first vertex on the path after x_i: either x_i^T or x_i^F . This choice models the truth assignment. There is also a path for each clause. In addition, there will be supplementary paths that fill the unused variable tracks. Figure 4 shows an example of a variable x_i that appears in three clauses C_u, C_v, and C_w. Consequently, the two tracks x_i^T and x_i^F, which run in parallel, pass through three *clause* gadgets, which are shown schematically as gray boxes in Figure 4 and which are drawn in greater detail in Figure 5. The bold path in Figure 4 corresponds to assigning the value *false* to x_i: the path follows the track x_i^F. For a variable that appears in ℓ clauses, there are $\ell + 1$ supplementary paths. In Figure 4, they are labeled $s_1, \ldots, s_4$. The path s_j covers the unused track (x_i^T in the example) between the $(j-1)$-st and the j-th clause in which the variable x_i is involved.

Initially, the path x_i can choose between the tracks x_i^T and x_i^F; the other track will be covered by a path starting at s_1. This choice is made possible by a *crossing* gadget. Each variable has two crossing gadgets attached, one at the beginning of the variable path and

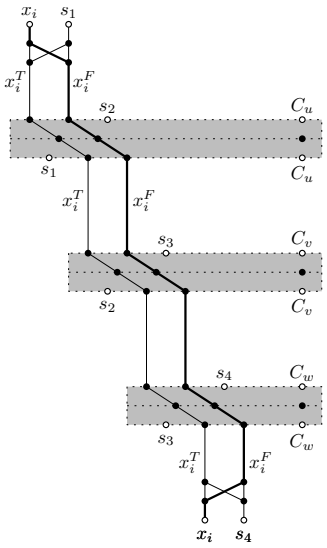

Figure 4 Schematic representation of a variable x_i. Sources and sinks are marked by white vertices, and their labels indicated the one-to-one correspondence φ between sources and sinks.

one at the end. Those gadgets consist of 6 vertices: x_i, s_1, x_i^T, x_i^F and two auxiliary vertices that allow the variable to change tracks. In Figure 4, the crossing gadgets appear at the top and the bottom. Note, that a variable can only change tracks in the crossing gadgets; the last supplementary path $s_{\ell+1}$ allows the path x_i to reach its target sink.

Figure 5 shows a clause gadget C_z in greater detail. It consists of three successive layers and connects the three variables that occur in the clause. The clause itself is represented by a clause path that spans only these three layers. A supplementary path starts at the first layer and one ends at the third layer of each clause gadget. Each layer of the clause gadget has at most 10 vertices: three for each variable, two that are on the tracks x_i^T, x_i^F, one for the supplementary path of the variable. There is at most three variables per clause. Finally, there is a vertex that corresponds to the clause itself.

Let C_z be the current clause which is the jth clause in which x_i appears, and it does so as a positive literal (as in Figure 5a). Each track, say x_i^T, connects to the corresponding vertex in the middle and bottom layer of the clause gadget and to s_j (the end of the supplementary path). The track of the literal that does not appear in this clause (x_i^F in this case) is also connected to the vertex of the clause on the middle layer (z in Figure 5). Moreover, the middle layer vertex of this track is connected to the top and bottom layer vertices that correspond to the clause. The supplementary path s_{j+1} starts in this clause gadget and goes through the middle layer and then connects to the vertices of both tracks on the bottom layer. Figure 5 depicts all these connections.

We can make the following observations about the interaction between the variable x_i and the clause C_z. We, further, illustrate those in Figure 5.

1. A variable path (shown in black) that enters on the track x_i^T or x_i^F must leave the gadget on the same track. Equivalently, a path cannot change track except in the crossing gadgets.
2. The clause path (in red) can make a detour through vertex v (w.r.t. Figure 5) only if the variable x_i makes the clause true, according to the track chosen by the variable path. In this case, the supplementary path s_j covers the intermediate vertex z of the clause.
3. If variable x_i makes the clause true, the clause path may also choose a different detour, in case more variables make the clause true.

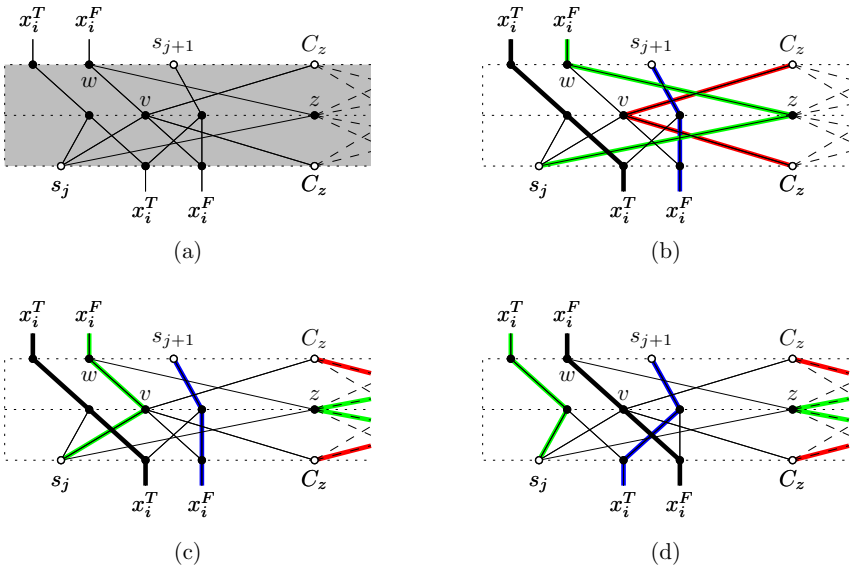

Figure 5 (a) Gadget for a clause C_z containing a variable x_i as a positive literal. The clause involves two other variables, whose connections are indicated by dashed lines. (b–d) The possibilities of paths passing through the gadget: (b) $x_i = \mathit{true}$, the clause is fulfilled, and the clause path makes its detour via the vertex v. (c) $x_i = \mathit{true}$, the clause is fulfilled, and the clause path makes its detour via another vertex. (d) $x_i = \mathit{false}$, this variable does not contribute to fulfilling the clause, and the clause path *has to* make its detour via another vertex. For a negative literal, the detour vertex v and the upper neighbor w of z would be placed on the other track, x_i^T.

4. The supplementary path s_j (shown in green) can reach its sink vertex.
5. The supplementary path s_{j+1} (in blue) can reach the track the track x_i^T or x_i^F which is not used by the variable path.

Since each clause path *must* make a detour into one of the variables, it follows from Property 2 that a set of disjoint source-sink paths exists if and only if all clauses are satisfiable.

The special properties of the graph that are required for the Disjoint Paths Problem on a Directed Acyclic Graph (DP) can be checked easily. Whenever a vertex has one or more incoming arcs, they come from the previous layer of the same clause gadget. We can assign three distinct layers to each clause gadget and to each crossing gadget. Thereby, we ensure that every layer contains at most 10 vertices. It follows from the construction that the graph has just enough vertices that the shortest source-sink paths can be disjointly packed, but we can also check this explicitly. (See the full version for an explicit calculation.) ◀

7 Reduction to Colored Token Swapping

We have shown in Lemma 8 how to reduce 3SAT to the Disjoint Paths Problem on a Directed Acyclic Graph (DP) in linear time. Now, we show how to reduce from this problem to the colored token swapping problem.

▶ **Lemma 9.** *There exists a linear reduction from DP to the colored token swapping problem.*

Proof. Let G be a directed graph and φ be a bijection, as in the definition of DP. We place k tokens $t_1, \ldots, t_k$ of distinct colors on the vertices in V^-, see Figure 6. Their target positions are in V^+ as determined by the assignment φ. We define a color for each layer $V_1, \ldots, V_{t-1}$. Each vertex in layer V_i which is not a sink is colored by the corresponding color.

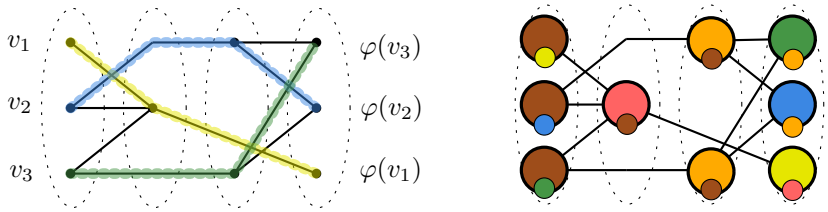

Figure 6 left: an instance of the disjoint paths problem with $t = 4$ and $k = 3$, together with a solution; right: an equivalent instance of the colored token swapping problem.

Recall that for each vertex v all ingoing edges come from the same layer, which we denote by L_v. On each vertex v which is not a source, we place a token with the color of layer L_v. We call these tokens *filler tokens*.

We set the threshold T to $|V(G)| - k$. This equals the number of filler tokens.

We have to show that there are k paths with the properties above if and only if there is a sequence of at most T swaps that brings every token to its target position (see the full version). ◀

We conclude that the Colored Token Swapping Problem is NP-hard. This has already been proved by Yamanaka et al. [23], even when there are only three colors.

The reduction in Lemma 9 produces instances of the colored token swapping problem with additional properties, which are directly derived from the properties of DP. A precise definition of the *structured token swapping problem* can be found in the full version.

8 Reduction to the Token Swapping Problem

In this section we describe the final reduction which results in an instance of the token swapping problem. To achieve this we make use of the even permutation network gadget from Section 5. For adhering to space constraints we omit from here an illustration of the reduction, the the full version for details.

▶ **Lemma 10.** *There exists a linear reduction from the structured colored token swapping problem to the token swapping problem.*

Proof. Let I be an instance of the structured colored token swapping instance. We denote the graph by G, the layers by $V_1, \ldots, V_t$, the sources by V^-, the sinks by V^+ and the threshold for the number of swaps by k.

We construct an instance J of the token swapping problem. The graph $\overline{G}$ consists of two copies of G. For each set $V_j \setminus V^+$, we add *one* even permutation network to the union of both copies. In other words, the two copies of $V_j \setminus V^+$ serve as inputs of the permutation network. We denote the output vertices of the permutation network attached to the copies of $V_j \setminus V^+$ by V'_j. The filler tokens that were are destined for $V_j \setminus V^+$ in I have V'_j as their new final destination in J, see the full version for illustrations and details. It is not important how we assign each token to a target vertex, as long as this assignment is consistent between the two copies of G (that is, for the bijection φ_j between the input and output vertices of the permutation network, we have that the tokens in the first copy are sent to the image under φ_j of the vertices in the first copy, while the tokens in the second are sent to the corresponding vertices in the image of the second copy).

Further each token gets a unique label. The threshold $\overline{k}$ for the number of swaps is defined by $2k$ plus the number of swaps needed for the permutation networks.

We show at first that this reduction is linear. For this it is sufficient to observe that the size of each layer is constant. And thus also the permutation network attached to these layers have constant size each.

Now we show correctness. Let I be an instance of the structured colored token swapping problem and J the constructed instance as described above.

[$\Rightarrow$] Let S be a valid sequence of swaps that brings every token on G to a correct target position within k swaps. We need to show that there is a sequence of $\overline{k}$ swaps that brings each token in $\overline{G}$ to its unique target position. We perform S on each copy of G. Thereafter every token is swapped through the permutation network to its target position. Note that the permutation between the input and output is an *even* permutation. This is because we doubled the graph G. Therefore, the number of swaps in the permutation network is constant regardless of the permutation of the filler tokens in layer V_i, by Lemma 7. This also implies that the total number of swaps is $\overline{k}$.

[$\Leftarrow$] Assume that there is a sequence S' of $\overline{k}$ swaps that brings each token of J to its target position. This implies that each filler token went through the permutation network to its correct target vertex. In order to do that each filler token must have gone to some input vertex of its corresponding permutation network. As the number of swaps inside each permutation network is independent of the permutation of the tokens on the input vertices, there remain exactly $2k$ swaps to put all tokens in each copy of G at its right place. This implies that each token in G can be swapped to its correct position in I in k swaps as claimed. ◀

9 Hardness of the Token Swapping Problem

In this section we put together all the reductions. They imply the following theorem.

▶ **Theorem 11.** *The token swapping problem has the following properties:*

1. *It is NP-complete.*
2. *It cannot be solved in time $2^{o(n)}$ unless ETH fails, where n is the number of vertices.*
3. *It is APX-hard.*

These properties also hold, when we restrict ourselves to instances of bounded degree.

Proof. For the NP-completeness, we reduce from 3SAT. For the lower bound under ETH, we need to use the Sparsification Lemma, see [13], and reduce from 3SAT instances where the number of clauses is linear in the number of variables. This prevents a potential quadratic blow up of the construction. For the inapproximability result, we reduce from 5-OCCURRENCE-MAX-3SAT. (In this variant of 3SAT each variable is allowed to have at most 5 *occurrences*.) This gives us some additional structure that we use for the argument later on. In all three cases the reduction is exactly the same.

Let f be a 3SAT instance. We denote by $K(f)$ the instance of the token swapping problem after applying the reduction of Lemma 8, Lemma 9 and Lemma 10 in this order. (It is easy to see that the graph of $K(f)$ has bounded degree.)

As all three reductions are correct, we can immediately conclude that the problem is NP-hard. NP-membership follows easily from the fact that a valid sequence of swaps is at most quadratic in the size of the input and can be checked in polynomial time.

The Exponential Time Hypothesis (ETH) asserts that 3SAT cannot be solved in $2^{o(n')}$, where n' is the number of variables. The Sparsification Lemma implies that 3SAT cannot be solved in $2^{o(m')}$, where m' is the number of clauses. As the reductions are linear the number of vertices in $K(f)$ is linear in the number of clauses of f. Thus a subexponential-time

algorithm for the token swapping problem implies a subexponential-time algorithm for 3SAT and contradicts ETH.

To show APX-hardness, we do the same reductions as before, but we reduce from 5-OCCURRENCE-MAX-3SAT. Thus we can assume that each variable in f appears in at most 5 clauses. This variant of 3SAT is also APX-hard, see [1]. Assume a constant fraction of the clauses of f are not satisfiable. We have to show that we need an additional constant fraction on the total number of swaps. For this, we assume that the reader is familiar with the proofs of Lemma 8, 9 and 10. It follows from these proofs that there is a constant sized gadget in $K(f)$ for each clause of f. Also there are certain tokens that represent variables and the paths they take correspond to a variable assignment. We denote with x, y, z the variables of some clause C, and we denote with T_x, T_y, T_z the tokens corresponding to these variables and G_C the gadget corresponding to C. We need two crucial observations. In the case where the paths taken by the tokens T_x, T_y and T_z do not correspond to an assignment that makes C true, at least one more swap is needed. This swap can be attributed to the clause gadget G_C. In case that a token T_x changes its track, which corresponds to another assignment of the variable, then at least one more swap needs to be performed that can be attributed to all its clauses with value 1/5. These two observations follow from the proofs of the above lemmas, as otherwise it would be possible to "cheat" at each clause gadget and the above lemmas would be incorrect. The observations also imply the claim. Let f be a 3SAT formula with a constant fraction of the clauses not satisfiable. Assume at first that the swaps are "honest" in the sense that the variable tokens T_x does not change its track and corresponds consistently with the same assignment. In this case, by the first observation, we need at least one extra swap per clause. And thus a constant fraction of extra swaps, compared to the total number of swaps. In a dishonest sequence of swaps, changing the track of some variable token T_x fixes at most five clauses. This implies at least one extra swap for every five unsatisfied clauses, which is a constant fraction of all the swaps as the total number of swaps is linear in the number of clauses of f. This finishes the APX-hardness proof. ◀

Acknowledgment. This project was initiated on the GWOP 2014 workshop in Val Sinestra, Switzerland. We want to thank the organizers (GREMO) for inviting us to the very enjoyable workshop.

References

1 Sanjeev Arora and Carsten Lund. Hardness of approximations. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 399–446. PWS Publishing, 1996.

2 Paul Bonsma, Takehiro Ito, Marcin Kamiński, and Naomi Nishimura. Invitation to combinatorial reconfiguration. Presentation at the First International Workshop on Combinatorial Reconfiguration (CoRe 2015), `http://www.ecei.tohoku.ac.jp/alg/core2015/page/template.pdf`, February 2015.

3 Paul Bonsma, Marcin Kamiński, and Marcin Wrochna. Reconfiguring independent sets in claw-free graphs. In R. Ravi and Inge Li Gørtz, editors, *Algorithm Theory – SWAT 2014*, volume 8503 of *Lecture Notes in Computer Science*, pages 86–97. Springer International Publishing, 2014. `doi:10.1007/978-3-319-08404-6_8`.

4 Arthur Cayley. Note on the theory of permutations. *Philosophical Magazine Series 3*, 34:527–529, 1849. `doi:10.1080/14786444908646287`.

5 Gruia Călinescu, Adrian Dumitrescu, and János Pach. Reconfigurations in graphs and grids. *SIAM Journal on Discrete Mathematics*, 22(1):124–138, 2008. `doi:10.1137/060652063`.

6 Erik D. Demaine, Martin L. Demaine, Eli Fox-Epstein, Duc A. Hoang, Takehiro Ito, Hirotaka Ono, Yota Otachi, Ryuhei Uehara, and Takeshi Yamada. Linear-time algorithm for sliding tokens on trees. *Theoretical Computer Science*, 600:132–142, 2015. doi:10.1016/j.tcs.2015.07.037.

7 Ruy Fabila-Monroy, David Flores-Peñaloza, Clemens Huemer, Ferran Hurtado, Jorge Urrutia, and David R. Wood. Token graphs. *Graphs and Combinatorics*, 28:365–380, 2012. doi:10.1007/s00373-011-1055-9.

8 Eli Fox-Epstein, Duc A. Hoang, Yota Otachi, and Ryuhei Uehara. Sliding token on bipartite permutation graphs. In Khaled Elbassioni and Kazuhisa Makino, editors, *Algorithms and Computation*, volume 9472 of *Lecture Notes in Computer Science*, pages 237–247. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48971-0_21.

9 Parikshit Gopalan, Phokion G. Kolaitis, Elitza N. Maneva, and Christos H. Papadimitriou. The connectivity of Boolean satisfiability: Computational and structural dichotomies. *SIAM J. Comput.*, 38(6):2330–2355, 2009. doi:10.1137/07070440X.

10 Daniel Graf. How to sort by walking on a tree. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms – ESA 2015*, volume 9294 of *Lecture Notes in Computer Science*, pages 643–655. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48350-3_54.

11 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. doi:10.1016/j.tcs.2005.05.008.

12 Lenwood S. Heath and John Paul C. Vergara. Sorting by short swaps. *Journal of Computational Biology*, 10(5):775–789, 2003. doi:10.1089/106652703322539097.

13 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

14 Mark R. Jerrum. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36:265–289, 1985. doi:10.1016/0304-3975(85)90047-7.

15 Marcin Kamiński, Paul Medvedev, and Martin Milanič. Complexity of independent set reconfigurability problems. *Theoretical Computer Science*, 439:9–15, 2012. doi:10.1016/j.tcs.2012.03.004.

16 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

17 Tillmann Miltzow, Lothar Narins, Yoshio Okamoto, Günter Rote, Antonis Thomas, and Takeaki Uno. Approximation and Hardness of Token Swapping. Preprint, February 2016. arXiv:1602.05150.

18 Amer E. Mouawad, Naomi Nishimura, Venkatesh Raman, and Marcin Wrochna. Reconfiguration over tree decompositions. In Marek Cygan and Pinar Heggernes, editors, *Parameterized and Exact Computation*, volume 8894 of *Lecture Notes in Computer Science*, pages 246–257. Springer International Publishing, 2014. doi:10.1007/978-3-319-13524-3_21.

19 Igor Pak. Reduced decompositions of permutations in terms of star transpositions, generalized Catalan numbers and k-ARY trees. *Discrete Mathematics*, 204(1-3):329–335, 1999. Selected papers in honor of Henry W. Gould. doi:10.1016/S0012-365X(98)00377-X.

20 Daniel Ratner and Manfred Warmuth. The (n^2-1)-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990. doi:10.1016/S0747-7171(08)80001-6.

21 Richard M. Wilson. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*, 16(1):86–96, 1974. doi:10.1016/0095-8956(74)90098-7.

22 Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. Swapping labeled tokens on graphs. *Theoretical Computer Science*, 586:81–94, 2015. Special issue for the conference *Fun with Algorithms* 2014. doi:10.1016/j.tcs.2015.01.052.

23 Katsuhisa Yamanaka, Takashi Horiyama, David Kirkpatrick, Yota Otachi, Toshiki Saitoh, Ryuhei Uehara, and Yushi Uno. Swapping colored tokens on graphs. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures*, volume 9214 of *Lecture Notes in Computer Science*, pages 619–628. Springer International Publishing, 2015. doi:10.1007/978-3-319-21840-3_51.

24 Gaku Yasui, Kouta Abe, Katsuhisa Yamanaka, and Takashi Hirayama. Swapping labeled tokens on complete split graphs. *SIG Technical Reports*, 2015-AL-153(14):1–4, 2015.