


PLS-Completeness of String Permutations

Dominik Scheder 

TU Chemnitz, Germany

Johannes Tantow 

TU Chemnitz, Germany

Abstract

Bitstrings can be permuted via permutations and compared via the lexicographic order. In this paper we study the complexity of finding a minimum of a bitstring via given permutations. As finding a global optimum is known to be NP-complete [1], we study the local optima via the class PLS[8] and show hardness for PLS. Additionally, we show that even for one permutation the global optimization problem is NP-complete and give a formula that has these permutation as its symmetries. This answers an open question inspired from Kołodziejczyk and Thapen [9] and stated at the *SAT and interactions* seminar in Dagstuhl [14].

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases PLS, total search problems, local search, permutation groups, symmetry

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.56

Related Version *Full Version:* <https://arxiv.org/abs/2505.02622>

Acknowledgements We want to thank Neil Thapen for introducing the problem to us.

1 Introduction

Given a string $\mathbf{x} \in \{0, 1\}^n$ and a couple of permutations in S_n , we can apply a permutation to \mathbf{x} and obtain a new bit string. What is the lexicographically smallest string we can obtain this way? This problem is known [1] to be NP-hard. What about finding a local minimum, i.e., arriving at a bit string that cannot be further improved by a single application of a permutation? In this paper, we show that this problem is PLS-complete.

To be more formal, we are given a string $\mathbf{x} \in \{0, 1\}^n$ and permutations π_1, \dots, π_m on the set $[n]$. When we view \mathbf{x} as a function $[n] \rightarrow \{0, 1\}$, the notation $\mathbf{x} \circ \pi$ makes sense and is the string obtained from \mathbf{x} by permuting the coordinates according to π . By $\langle \pi_1, \dots, \pi_m \rangle$ we denote the subgroup of $S_{[n]}$ generated by the π_i . The problem k -PERMUTATION GLOBAL ORBIT MINIMUM asks for the $\pi \in \langle \pi_1, \dots, \pi_k \rangle$ such that $\mathbf{x} \circ \pi$ is lexicographically minimal. Babai and Luks [1] showed that this is NP-hard even for $k = 2$. In fact, we will see that it is NP-hard even for $k = 1$, i.e., a single permutation.

k -PERMUTATION LOCAL ORBIT MINIMUM asks for a local minimum. That is, an element $\pi \in \langle \pi_1, \dots, \pi_k \rangle$ such that

$$\mathbf{x} \circ \pi \preceq_{\text{lex}} \mathbf{x} \circ \pi \circ \pi_i \quad \text{for all } 1 \leq i \leq k,$$

i.e., a single application of a permutation π_i cannot further improve the string $\mathbf{x} \circ \pi$.

A local optimum always exists and hence this is an instance of a total search problem. Total search problems where solutions are recognizable in polynomial time form the class TFNP. Total search problems that can be stated as finding a local optimum with respect to a certain *cost function* and a *neighborhood relation* constitute the subclass PLS (polynomial local search). Known hard problems for PLS include finding a pure Nash-equilibrium in a congestion game[4] or finding a locally optimal max cut (LOCALMAXCUT) [13]. Almost all known PLS-complete problems require quite involved cost functions. Our problem



© Dominik Scheder and Johannes Tantow;
licensed under Creative Commons License CC-BY 4.0

33rd Annual European Symposium on Algorithms (ESA 2025).

Editors: Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman; Article No. 56; pp. 56:1–56:14

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

k -PERMUTATION LOCAL ORBIT MINIMUM has the benefit of using the possibly simplest cost function - the lexicographic ordering. The only other PLS-complete problem using a lexicographic cost function that we know of is FLIP, which asks to minimize the m -bit output of a circuit C , where the solutions are all n -bit inputs and the neighborhood relation is defined by flipping a single bit.

Thus, our result unifies two desirable properties - our PLS-complete problem is very combinatorial in nature (in contrast to FLIP) and uses a very simple cost function (in contrast to LOCALMAXCUT)

1.1 SAT Solving and Symmetry Breaking

When encoding a combinatorial problem as a CNF formula F (think of “is there a k -Ramsey graph on n vertices?”), the formula will often contain many symmetries. To make the problem easier for SAT solvers, one can take the statement

The satisfying assignment α should be a local lexicographical minimum with respect to those symmetries,

encode it as a CNF formula G and feed $F \wedge G$ to the SAT solver. Clearly, $F \wedge G$ is satisfiable if and only if F is. In case that $F \wedge G$ is unsatisfiable, SAT solvers are often expected to produce a proof of unsatisfiability. A popular proof system used in this context is DRAT [15]. However, it is not known to what extent DRAT can handle symmetry breaking [7], that is, whether a short DRAT-refutation of $F \wedge G$ can be transformed into a short DRAT-refutation of F . In this context, Thapen [14] asked whether there exists a polynomial algorithm that, given a CNF formula F , a handful of symmetries thereof, and a satisfying assignment α , finds a satisfying assignment β that is a local lexicographical minimum with respect to those symmetries. In this paper, we show that this problem is PLS-complete, which is evidence that such a polynomial time algorithm might not exist.

2 Preliminaries

2.1 Total search problems and PLS

The class FNP is the functional correspondent of NP. $\text{TFNP} \subset \text{FNP}$ is the subset of *total* search problems, i.e., problems that always have a solution. As this is a semantic class, TFNP has no known complete problems, and thus it is usually studied via its subclasses. These subclasses are based on the combinatorial principle that proves the existence of a solution. These principles include the existence of sinks in directed acyclic graphs (PLS)[8], the parity argument for directed and undirected graphs (PPAD, PPA) or the pigeonhole principle (PPP) (all introduced in [11]). Nonetheless, not all problems in TFNP can be categorized in one of the known subclasses, FACTORING being a prime example.

By the above characterization, PLS requires finding a sink of a directed acyclic graph G . This would be possible in polynomial time if G was given explicitly. Instead, G is always given implicitly via a circuit that computes the successor list of a given node. To make sure that G is acyclic, we have a second circuit computing a topological ordering, that is, a “cost” function that is strictly decreasing along the edges. A solution to the problem is a sink of G or an edge (u, v) with $\text{cost}(u) \leq \text{cost}(v)$, i.e., violating the decreasing cost condition. This guarantees the totality of the problem.

An alternative definition is the following. For a PLS problem P we have a set of instances I . Each instance $i \in I$ has a set of feasible solutions S (e.g. for the Euclidean traveling salesman problem the solutions are exactly the Hamilton cycles of K_n). Additionally, we require the following polynomial-time computable algorithms (usually given as circuits):

1. an algorithm that decides whether a given s is a feasible solution.
2. an algorithm that computes a starting solution $s \in S$
3. an algorithm that computes for a feasible solution s the neighborhood $N(s)$
4. an algorithm that computes for a solution s the cost $\text{cost}(s)$

A feasible solution s is a *local minimum* if $\text{cost}(s) \leq \text{cost}(s')$ for all $s' \in N(s)$. The definition is given for a minimization problem, but can be also defined in terms of a maximization problem.

We can easily transform this into a directed acyclic graph by keeping only those neighbors in $N(s)$ having strictly smaller cost—or even keeping only the one neighbor $s' \in N(s)$ of minimal cost (breaking ties arbitrarily).

Similar to NP, there is also a hardness structure in PLS. Let P and Q be two problems in PLS. P reduces to Q via a *PLS-reduction* (f, g) for functions f and g such that f maps an instance I from P to an instance $f(I)$ of Q and g maps a solution s of $f(I)$ to a solution $g(I, s)$ of P so that if s is a local minimum in $f(I)$ then also $g(I, s)$ is a local minimum in I . This was defined in [8] and the first natural PLS-complete problem is FLIP. There solutions are n -bit strings, the cost is calculated by a given circuit C and the neighborhood are all n -bit strings with a Hamming distance of 1.

The obvious greedy algorithm to find a solution for a PLS-problem is as follows: Use the given algorithms to compute the start solution and always select the best neighbor until there is no better solution. This solution is called the *standard solution* and the algorithm the *standard algorithm*. For the problem FLIP, finding the standard solution for a given start solution is PSPACE-complete [12, Lemma 4].

Reductions that preserve the PSPACE-completeness are called *tight* [13]. For this, we consider the transition graph $TG(I)$ of an instance I of the problem P , that has a directed edge from each feasible solution x to all of its neighbors $N(x)$.

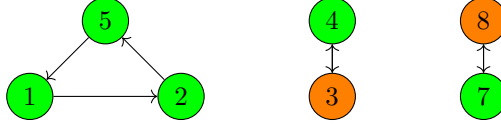
A PLS-reduction (f, g) from P to Q is called *tight* if for every instance I of P there exists a set \mathcal{R} of feasible solutions for $f(I)$ such that

1. \mathcal{R} contains all local optima of $f(I)$
2. For every solution s of I it is possible to construct in polynomial time a feasible solution $t \in \mathcal{R}$ such that $g(I, t) = s$
3. If the transition graph of $TG(f(I))$ contains a path from q to q' such that both q and q' are in \mathcal{R} and all other intermediate nodes are not in \mathcal{R} , let $p = g(I, q)$ and $p' = g(I, q')$ be the corresponding solutions in P . Then either $p = p'$ or there is an arc from p to p' in $TG(I)$.

An interesting subclass of PLS is CLS that is supposed to capture continuous local search problems. Recently, it was shown that $\text{CLS} = \text{PLS} \cap \text{PPAD}$ [5].

2.2 Permutation groups

A permutation of a set V is a bijection $\pi : V \rightarrow V$. For permutations π_1, \dots, π_k , we denote by $\langle \pi_1, \dots, \pi_k \rangle$ the subgroup of S_V generated by the π_i . Checking membership $\pi \in \langle \pi_1, \dots, \pi_k \rangle$ is non-trivial but can be done in polynomial time [6, Section 1].



■ **Figure 1** An example of the scenario for one permutation π .

3 Related Work

Many problems are known to be PLS-complete, whose reductions mostly start from FLIP. An influential reduction technique was used by Krentel [10] to show that finding a local minimum of a weighted CNF-formula is PLS-complete. The idea is to have multiple copies of the circuit, so called *test circuits* that precompute the effects of a flip. This idea is pushed further in [13] in order to show that finding a local minimum for weighted positive NAE(not all equal) 3-SAT is PLS-complete, and this is used to show that other problems as finding a LOCALMAXCUT or finding a stable configurations of Hopfield networks are PLS-complete.

Other direct reductions from FLIP are used in [4] to show that finding a pure Nash equilibrium in an asymmetric network congestion game is PLS-complete and in [3] to show that maximum constraint assignment, a generalization of CNF-SAT, is PLS-complete. These reductions are especially of interest to us as FLIP is to the best of our knowledge the only problem in PLS with lexicographical weights and hence needed for our reduction.

Numerous optimization problems on permutation groups given via generators π_1, \dots, π_k are studied in [2]. These include finding a $\pi \in \langle \pi_1, \dots, \pi_k \rangle$ that minimizes $\sum_{i \in V} c(i, \pi(i))$ for some cost function $c : V^2 \rightarrow \mathbb{R}$. All these problems are shown to be NP-complete via a reduction from finding a fixed-point free permutation, which is shown to be NP-complete.

Our problem has previously been studied in [1], but there the interest was in *global* optima. This was proven to be NP-hard even for abelian groups.

4 The one permutation problem

We start our investigation by studying the special case that $k = 1$, i.e., we have only one generator π_1 and our subgroup is $\langle \pi_1 \rangle$. In this case one can efficiently find a local optimum[9, Section 8.2]. We show this with a different algorithm again. In contrast, we will show that surprisingly finding a global optimum is NP-complete. To the best of our knowledge, this has only been known for *two* permutations [1].

4.1 Finding a local optimum

The problem is efficiently solvable when $k = 1$, i.e., we are only given a single permutation π . We describe a different way of solving it in contrast to [9]. We can transform π into the cycle notation and annotate each element with the bit that it is mapped to by the string $\mathbf{x} \in \{0, 1\}^n$. We call a cycle *interesting* if \mathbf{x} is non-constant on it. Additionally, we identify each cycle with its smallest member.

Consider the permutation $(1\ 2\ 5)(3\ 4)(7\ 8)$ and the string 001001 depicted in Figure 1. We color an element green if it is mapped to zero and orange if it is mapped to one by the string. The left cycle is the cycle of 1 and not interesting whereas the other two are which are identified by 3 and 7.

The permutation has no effect on elements on non-interesting cycles. Due to the cost function, lower indices are more costly than higher indices. We look for the interesting cycle that contains the position with the least index among all interesting cycles and let l be the

smallest index on it. There are indices i and $j := \pi(i)$ on this cycle with $x_i = 0$ and $x_j = 1$. Let k be such that $\pi^k(l) = i$. Then $\mathbf{x} \circ \pi^k$ has a 0 at position i but $\mathbf{x} \circ \pi^{k+1}$ has a 1. In other words, $\mathbf{x} \circ \pi^k$ is a local optimum

In example in Figure 1 we have $x = 3$, $j = 4$ and $d = 1$. The local optimal permutation is hence π .

4.2 Finding a global optima

► **Theorem 1** ([1]). *2-PERMUTATION GLOBAL ORBIT MINIMUM is NP-hard.*

This follows via a reduction from Independent Set where we encode a graph as a bit string, one bit per potential edge, and the permutations basically allow us to move the vertices of the independent set I to the front, generating a large prefix of $\binom{|I|}{2}$ many 0's.

Since 1-PERMUTATION LOCAL ORBIT MINIMUM was so clearly solvable in polynomial time (even by the greedy algorithm), it comes as a surprise that global optimization, even for *one* permutation, is NP-hard:

► **Theorem 2.** *1-PERMUTATION GLOBAL ORBIT MINIMUM is NP-hard.*

We will define an intermediate NP-complete problem called DISJUNCTIVE CHINESE REMAINDER, short DCR. For two numbers $t, m \in \mathbb{N}$ and a set $S \subseteq \mathbb{N}$, we write

$$t \not\equiv s \pmod{m}$$

to state that $t \not\equiv s \pmod{m}$ for all $s \in S$. Now in the DCR decision problem, we are given moduli m_1, \dots, m_l and sets of “forbidden remainders” S_1, \dots, S_l with $S_i \subseteq \mathbb{Z}_{m_i} := \{0, 1, \dots, m_i - 1\}$. All numbers are given in unary and the moduli are not required to be pairwise co-prime. DCR asks for a solution $t \in \mathbb{N}$ of the system

$$t \not\equiv S_i \pmod{m_i} \quad \forall i = 1, \dots, l.$$

This is clearly in NP: if there is a solution $x \in \mathbb{N}$, then there is one with $0 \leq t \leq \text{lcm}(m_1, m_2, \dots, m_l)$ and thus t has polynomially many bits in binary. Verifying that this t is a solution can now be done by division with remainder.

► **Lemma 3.** *DCR is NP-complete.*

Proof. We reduce from 3-Colorability. Given a graph $G = (V, E)$ with $|V| = n$, we let $3 = p_1, p_2, \dots, p_n$ be the first n prime numbers greater than 2. By the prime number theorem, p_n is polynomial in n , thus the p_i can be found in polynomial time by a brute force search using naive prime number testing.

We let $N = p_1 \cdot p_2 \cdot \dots \cdot p_n$ and define the following function from \mathbb{Z}_N to 3-colorings of the vertices V : given a number $0 \leq t \leq N - 1$, the corresponding coloring $c_t : V \rightarrow \{r, g, b\}$ is defined by

$$c_t(v_i) = \begin{cases} r & \text{if } t \equiv 0 \pmod{p_i} \\ g & \text{if } t \equiv 1 \pmod{p_i} \\ b & \text{else.} \end{cases}$$

By the Chinese Remainder Theorem, this is a surjective function and thus every 3-coloring can be encoded by one single number $0 \leq x \leq N - 1$. For an edge $e = \{u, v\}$, we write a constraint that makes sure that u and v receive different colors. For each pair $(a, b) \in \mathbb{Z}_{p_u} \times \mathbb{Z}_{p_v}$ with

$(a, b) = (0, 0)$ or $(a, b) = (1, 1)$ or $a \geq 2, b \geq 2$, we compute the unique number $c \in \mathbb{Z}_{p_u q_v}$ with $c \equiv a \pmod{p_u}$ and $c \equiv b \pmod{q_v}$. Let S_e be the set of all numbers c thus constructed, set $m_e := p_u \cdot p_v$ and write the constraint

$$x \notin S_e \pmod{m_e}. \quad (1)$$

If e is the i^{th} edge of the graph, we set $m_i = p_u p_v$ and $S_i = S_e$. We see that x satisfies (1) if and only if x encodes a properly colored edge e . ◀

► **Lemma 4.** *DCR reduces to 1-PERMUTATION GLOBAL ORBIT MINIMUM.*

Proof. Given an instance of DCR we set $M = m_1 + m_2 + \dots + m_l$ and define a permutation π on $[M]$ that has l disjoint cycles, one of length m_i for each i . We label the elements of the i^{th} cycle consecutively with the numbers $0, \dots, m_i - 1$. We define a bit string $\mathbf{x} \in \{0, 1\}^M$ that has exactly one 1 in each cycle, placed on the element that has label 0. The orbit element $\mathbf{x} \circ \pi^t$ still has exactly one 1 in cycle i , but now at the vertex labeled $t \bmod m_i$.

For each cycle i , let F_i be the elements on it whose labels are in the set S_i of forbidden remainders and let $F := F_1 \cup \dots \cup F_l$. We order the elements of $[M]$ such that the elements of F come first. There exists a solution $t \in \mathbb{N}$ to the DCR instance if and only if the string $\mathbf{x} \circ \pi^t$ has no 1 in any position in F . ◀

5 PLS-Hardness

We now state and prove our main result:

► **Theorem 5.** *k -PERMUTATION LOCAL ORBIT MINIMUM is PLS-complete.*

In order to view it as a PLS-problem we have an instance I consisting of the permutations π_1, \dots, π_k and the string $s \in \{0, 1\}^N$. Solutions are permutations from $\langle \pi_1, \dots, \pi_k \rangle$ as we can efficiently recognize whether permutations are in the group generated by π_1, \dots, π_k . The start solution is the identity. Neighbors of π are permutations $\pi \circ \pi_i$ for $i \in \{1, \dots, k\}$. The cost of a permutation π is $\sum_{i=1}^N s(\pi(i)) 2^{N-i}$, where $s(\pi(i))$ is the digit of the position that i is mapped to by π . All these can be computed in polynomial time, hence the problem is in PLS.

A solution π is cheaper than a solution σ for a string s if there is an integer i such that for all $j < i$ we have that $s(\sigma(j)) = s(\pi(j))$ and $s(\pi(i)) < s(\sigma(i))$ as the cost is a geometric sum.

We can turn the minimization problem into a maximization problem by inverting the string.

5.1 High-level idea

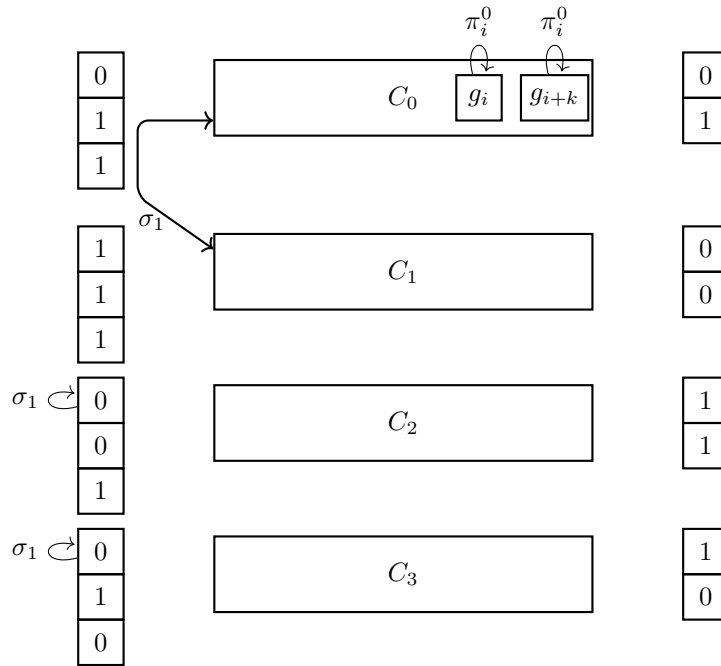
We reduce from the PLS-complete problem FLIP. This problem is especially suitable since it uses a lexicographic cost function, too. Formally FLIP is defined as follows:

► **Definition 6.** *An instance of FLIP consists of a circuit C with n inputs and m outputs. Feasible solutions are all input assignments, i.e., the set $\{0, 1\}^n$. The cost of a solution \mathbf{x} is the output of $C(\mathbf{x}) \in \{0, 1\}^m$. Two solutions are neighbors if they differ in a single bit. The cost function is defined by reading the output as a number in binary; in other words, by the lexicographic order on $\{0, 1\}^m$. We are asked to find a solution whose cost is minimal among all its neighbors.*

We use an idea by Krentel[10] and have $n + 1$ copies C_0, C_1, \dots, C_n of the circuit C , where C_0 is fed as an input \mathbf{x} and C_i is fed as an input $\mathbf{x} \oplus \mathbf{e}_i$, i.e., \mathbf{x} with the i -th bit flipped. The setup is depicted in Figure 2.

The permutation group consists of two types of permutations. The first kind π_i^j simulates flipping the output of gate i in circuit j ; we allow for gates and hence circuits to be temporarily evaluated incorrectly. The input and output of a gate are not saved anywhere but syntactically built into the permutations and string. An exception is the input and output of the circuit, which we save for later usage. We use some positions as very important control bits to ensure that the correct evaluation is always possible.

The second type of permutation σ_j swaps the circuit 0 with the circuit j and flips the j -th input bit for all other circuits. This simulates a step in the FLIP-problem.



■ **Figure 2** A high level overview of the reduction.

5.2 Definition of the reduction

We assume without loss of generality that the circuit C consists only of NAND-gates. Additionally, we assume that no input is directly passed to an output, i.e. on every path from an input to an output, there is at least one gate.

We will now describe the set V of *positions*, the permutations in S_V , and how strings in $\{0, 1\}^V$, called *assignments*, correspond to the circuits C_0, \dots, C_n computing their values on an input \mathbf{x} and its neighbors $\mathbf{x} \oplus \mathbf{e}_i$. We face two main challenges:

1. We need an operation of the form “flip bit i ”, but permutations can only permute positions, not flip bits. We solve this by replacing position i by two positions i_0, i_1 and encoding $[y_i = 0]$ by $[y_{i_0} = 0, y_{i_1} = 1]$ and $[y_i = 1]$ by $[y_{i_0} = 1, y_{i_1} = 0]$. Flipping bit i then corresponds to the permutation that swaps i_0 and i_1 , i.e., the transposition (i_0, i_1) . We call the one-position-per-bit view the *condensed view* and the two-positions-per-bit view the *expanded view*. We will give details in the full version due to space restrictions. For now, we phrase things in the condensed view.

2. Usually, when we flip an input x_i to a circuit C , we imagine the change propagating instantaneously through the circuit C , potentially changing its output. Here, we need to allow for a way for this change to proceed gradually; therefore, we allow gates to be temporarily in an incorrect state.

When we have a position $i \in V$ and an assignment $\mathbf{y} \in \{0, 1\}^V$ and $y_i = b$, we sometimes say i is assigned value b and sometimes the label of i is b .

State of a gate. The *state of a gate* is a triple (x, y, b) where x, y are the two input bits and b is the output bit. If $b = \neg(x \wedge y)$ we call (x, y, b) *correct*, because b is what it is supposed to be: the NAND of x and y ; otherwise, we call it *incorrect*. A gate g is represented by a gadget of four positions $g_{00}, g_{01}, g_{10}, g_{11}$, ordered as a 2×2 square, as shown in Figure 3. A state (x, y, b) is encoded as follows: position g_{xy} is labeled b , the three remaining ones are labeled with $\neg b$. Note that not all labelings of the gadget correspond to a gate state, only those where the number of positions labeled 1s is one or three.

► **Observation 7.** *The triple (x, y, b) is correct if and only if the position g_{11} in its representation is labeled 0.*

This is the core reason why we use this representation: we can determine correctness of a gate by reading just one bit. This will be important later when defining a cost function: having a 1 at those *control positions* is bad.

Input and output variables. Let $x_i^{(j)}$ be the i -th input variable to the j -th circuit (so $1 \leq i \leq n$ and $0 \leq j \leq n$). We introduce one position for each $x_i^{(j)}$. Similarly, let $c_k^{(j)}$ be the k -th output value of the j -th circuit; we introduce one position for each $c_k^{(j)}$.

A central definition is that of a well-behaved assignment. Basically, it formalizes when an assignment encodes the partial evaluation of the inputs by the $n + 1$ circuits.

- **Definition 8.** *An assignment $\mathbf{y} \in \{0, 1\}^V$ is called well-behaved if the following hold:*
- *For each gate g , the four positions $g_{00}, g_{01}, g_{10}, g_{11}$ are the encoding of a gate state $(a_1, a_2, b) \in \{0, 1\}^3$.*
 - *If the output of a gate g is the l -th input of some gate h , then the corresponding input and output values agree, i.e., if the state of h is (a'_1, a'_2, b') then $b = a'_l$.*
 - *If the l -th input of g is an input variable $x_i^{(j)}$, then $x_i^{(j)}$ is assigned the value a_l .*
 - *If g is the k -th output gate of circuit j , then its output value b is the same as the label of $c_k^{(j)}$.*
 - *The labels of the input values $x_i^{(j)}$ equal $x_i^{(0)}$ if $i \neq j$, and are unequal if $i = j$; in words, if the labels of the input positions of C_0 form a vector $\mathbf{x} \in \{0, 1\}^n$, then those of C_j form the vector $\mathbf{x} \oplus e_j$.*

Next, we describe the permutations on the positions. They come in two types: (1) flipping a gate and (2) swapping two circuits.

Flipping a gate. If g is in state (a_1, a_2, b) , then flipping g means replacing b by $\neg b$, and for each gate h (in state (a'_1, a'_2, b')) into which g feeds as l -th input, flipping a'_l . Since our permutations do not work on the *state* of a gate but on its representation in the four-position gadget (Figure 3), we work as follows: we flip the labels in all positions of g , i.e., $g_{00}, g_{01}, g_{10}, g_{11}$; if g is the first input of h , we swap positions of h horizontally: swap h_{00} with h_{10} and h_{01} with h_{11} ; if g is the second input of h , we perform a vertical swap: h_{00}

with h_{01} and h_{10} with h_{11} . This operation changes the state of g from incorrect to correct (or vice versa) and may also change correctness of h . If g happens to be the k -th output bit of circuit j , then this operation also flips position $c_k^{(j)}$. See Figure 4 for an example of two consecutive gates being flipped. We call this permutation π_g . If g is the i -th gate in circuit j , we may also call it π_i^j . Note that if \mathbf{y} is a well-behaved then $\mathbf{y} \circ \pi_g$ is well-behaved, too.

Swapping two circuits. We want a permutation that simulates flipping the i -th input bit to C , the circuit in the instance of FLIP. We achieve this by swapping C_0 with C_i – that is, swapping every position (input values, values in gate gadgets, output values) in C_0 with its corresponding position in C_i , and simultaneously flipping the i -th input bit $x_i^{(j)}$ for every circuit C_j with $j \in \{1, \dots, n\} \setminus \{i\}$; naturally, if this $x_i^{(j)}$ is the first input to a gate g in C_j , we have to perform the “horizontal swap” at g outlined above, and if it is the second input to g , a “vertical swap” at g . We call this permutation σ_i . Again, if \mathbf{y} is well-behaved then $\mathbf{y} \circ \sigma_i$ is well-behaved, too.

The starting string $\mathbf{y}_{\text{start}} \in \{0, 1\}^V$. This is the assignment where C_0 has input $\mathbf{0} \in \{0, 1\}^n$ and C_i has input $\mathbf{e}_i \in \{0, 1\}^n$ and all gates have output 0 (whether correctly or incorrectly). This is certainly well-behaved (or rather, can be made well-behaved by making sure that input to gate h matches output of gate g should they be connected).

We now have a set V of positions, an assignment $\mathbf{y}_{\text{start}} \in \{0, 1\}^V$, and a set of permutations-gate-flippers π_g for each gate g and circuit-swappers σ_i for each $i \in [n]$. They generate a subgroup G of S_V . It is clear that the orbit of $\mathbf{y}_{\text{start}}$ under G is the set of well-behaved assignments.

5.3 The cost function

We have promised to use a lexicographic ordering as a cost function. That is, if $\mathbf{y}, \mathbf{y}' \in \{0, 1\}^V$ are two assignments, then \mathbf{y} is better than \mathbf{y}' (meaning lower cost) if $\mathbf{y} \prec_{\text{lex}} \mathbf{y}'$. Thus, to define the cost function it suffices to specify an ordering on the positions in V .

- Positions in C_0 come before positions in C_1 and so on.
- Within a circuit, most important are the control positions g_{11} of the gates, followed by the output gates, followed by all remaining positions.
- Within control positions in the same circuit, the order follows the topological ordering of the circuit, i.e., if g feeds into h , then g 's control position comes before h 's.

5.4 Proof of Correctness

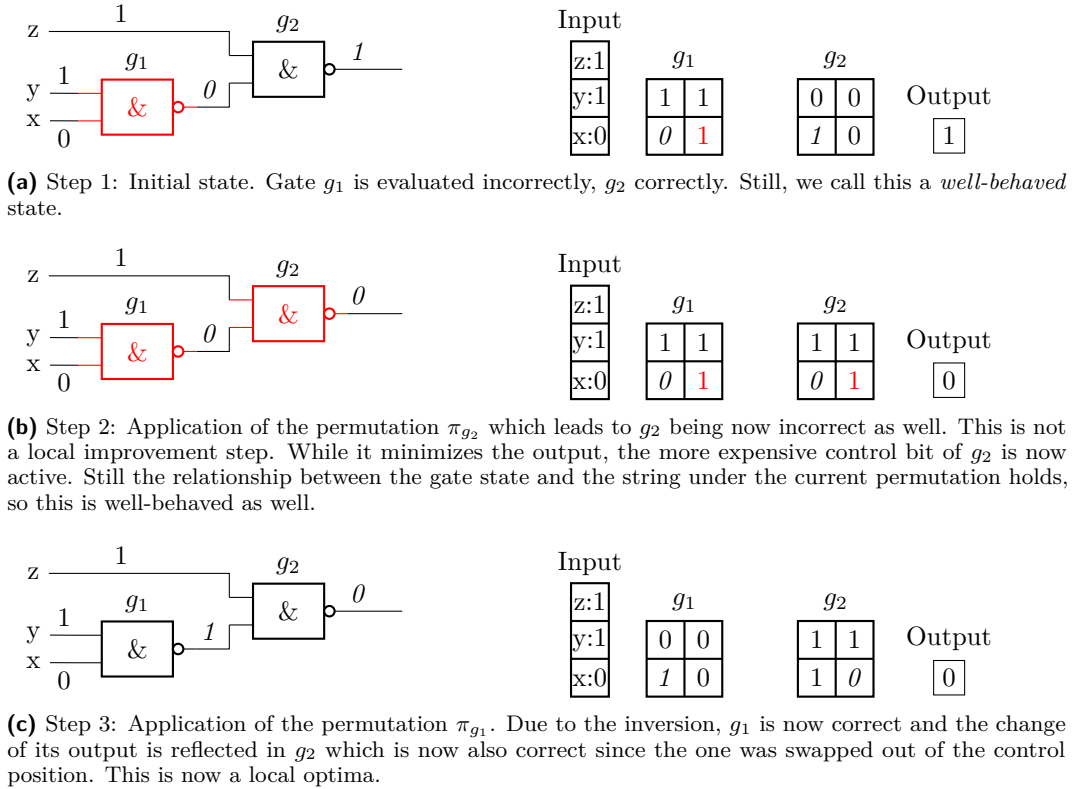
In this section we now prove the correctness and tightness of our reduction. In order to show the correctness we have to show that if $\pi \in G$ is a local optimum of the k -PERMUTATION LOCAL ORBIT MINIMUM instance, i.e., if the assignment $\mathbf{y} := \mathbf{y}_{\text{start}} \circ \pi$ cannot be further improved by applying a π_g or a σ_j , then the input to C_0 encoded in \mathbf{y} corresponds to a local optimum of FLIP.

Well-behaved permutations for a string s and a circuit C are interesting, because they realize the evaluation of the circuit somehow. The only problem is that the gate does not have the correct output in the current permutation with the string s compared to C .

► **Lemma 9.** *Let \mathbf{y} be a well-behaved assignment and suppose that $y(g_{11}) = 0$ for every gate – every control position is labeled 0. Then all output positions are mapped to correct results according to C . That is, if $\mathbf{x} \in \{0, 1\}^n$ is the vector to which the positions of circuit C_0 are mapped under \mathbf{y} , then*

	$x = 0$	$x = 1$
$y = 0$	$g_{00} : 0$ $g_{00} : 01$	$g_{01} : 1$ $g_{01} : 10$
$y = 1$	$g_{10} : 1$ $g_{10} : 10$	$g_{11} : 1$ $g_{11} : 10$

■ **Figure 3** An example gate configuration with the input $x = 0$ and $y = 0$ and the output 0. This is incorrect for a NAND-gate as indicated by the control bit. In light grey we denote the expanded encoding of the positions.



■ **Figure 4** Step-by-step evaluation of the circuit and reduction process. Each step shows the circuit state (left) and the reduction state (right). Currently incorrect gates are marked red and the output of a gate is identifiable via italics.

1. the input positions of C_i are mapped to $\mathbf{x} \oplus \mathbf{e}_i$ (this actually holds for every well-behaved \mathbf{y} , control positions being 0 or 1),
2. the output positions of C_0 are mapped to $C(\mathbf{x})$;
3. the output positions of C_j are mapped to $C(\mathbf{x} \oplus \mathbf{e}_j)$

Proof. This follows from Observation 7 and induction over the sequence of gates. ◀

The previous lemma tells us that all control positions should be mapped to zero in any local optimum so that all output positions are mapped to the correct output of C given the input. We show that we can always apply a permutation to achieve this.

► **Lemma 10.** *Let π be a permutation from G . We consider a gate g_i in the circuit j . The control position of g_i is mapped to position of the form $g_{i,k}^j$ by π*

Proof. This can be proven by induction on the structure of π in the permutation group. Any generator preserves this property as it either does not affect this position at all $\pi_{i'}^{j'}$ for $i \neq i'$ or $j \neq j'$. If both i' and j' are equal to i and j , then the position is simply inverted which preserves this property. Additionally, any permutation σ_j either maps a gate to itself or swaps the gate, so that the claim holds here as well. ◀

► **Lemma 11.** *If \mathbf{y} is well-behaved and some control position g_{11} is 1 under \mathbf{y} , then \mathbf{y} is not a local optimum.*

Proof. Among all control positions assigned 1, let g_{11} be the highest-ranking (i.e., of smallest index). Now apply π_g , i.e., flip gate g , which inverts the bit by the previous lemma. Under $\mathbf{y} \circ \pi_g$, the control position g_{11} is now correct; successor gates h in the same circuit might now become incorrect, but their control positions have lower rank by our ordering; gates in other circuits are not affected. Thus, $\mathbf{y} \circ \pi_g \prec_{\text{lex}} \mathbf{y}$, and \mathbf{y} is not a local optimum. ◀

► **Corollary 12.** *In any local optimum all sub circuits are correctly evaluated.*

The previous lemmas suffice to show the main result.

► **Lemma 13.** *Let $\mathbf{y} = \mathbf{y}_{\text{start}} \circ \pi \in \{0, 1\}^V$ be a local optimum an instance of K -PERMUTATION LOCAL ORBIT MINIMUM. Suppose the input variables of C_0 are mapped to some $\mathbf{x} \in \{0, 1\}^n$. Then \mathbf{x} is a local optimum of the FLIP instance.*

Proof. Since \mathbf{y} is well-behaved, the input variables of C_i are mapped to $\mathbf{x} \oplus \mathbf{e}_i$. Since \mathbf{y} is a local minimum, by the corollary, the output values of C_j are in fact mapped to the correct output $C(\mathbf{x} \oplus \mathbf{e}_j)$.

Suppose now that the mapped input is not a local optimum for the FLIP instance. Then there must be neighbor with a better output, i.e., $C(\mathbf{x} \oplus \mathbf{e}_j) \prec_{\text{lex}} C(\mathbf{x})$. Now consider $\mathbf{y}' = \mathbf{y} \circ \sigma_j$. Under \mathbf{y}' , the control positions of C_0 are all mapped to 0 (because those of C_j were under \mathbf{y}); the output of C_0 under \mathbf{y}' is better than under \mathbf{y} because $C(\mathbf{x} \oplus \mathbf{e}_j) \prec_{\text{lex}} C(\mathbf{x})$. Control positions in C_i with $i \geq 1$ might now be 1, but their priority is less than that of C_0 's output. Thus, \mathbf{y}' is better than \mathbf{y} , and \mathbf{y} is not a local optimum. ◀

This concludes the proof of Theorem 5. Every permutation used in the reduction has an order of two, so it is an involution. Still, these permutations do not form a commutative group due to the σ_i permutations. This is to be expected as even finding a global optimum for the lexicographical leader of a string under an abelian permutation group where every element has an order of two is polynomial time solvable[1, Section 3.1].

We additionally note that the reduction is tight and hence finding a local optimal bitstring under permutations via the standard algorithm is PSPACE-complete.

► **Theorem 14.** *The given reduction is tight.*

Proof. Let I be an instance of FLIP and (f, g) the previously defined reduction. We use as \mathcal{R} simply the set of all permutations in the group of the generators which necessarily contains all local optima. We can find for any solution s of I in polynomial time a solution π with

$g(I, \pi) = s$ in \mathcal{R} by applying the σ_j permutations to construct the needed input string. This requires applying at most n permutations which can be done in polynomial time. Finally, we see that any path where only the endpoints are in \mathcal{R} must be an edge. If the edge is due to an π_i^j permutation the input does not change and hence both endpoints are mapped to the same solution. If the edge is alternatively due to a σ_j permutation this changes the input in one variable and is hence an edge in $TG(I)$ as it corresponds directly to a flip there. \blacktriangleleft

6 Realizing the permutations in propositional formula

In the problem **LOCALLY MINIMAL SOLUTION** we are given a CNF formula F over some variables V , a satisfying assignment $\alpha : V \rightarrow \{0, 1\}$, and a list of permutations π_1, \dots, π_k on V such that F is invariant under π_i (that is, when applying π_i to each variable occurrence in F , the resulting formula F' is equal to F up to a re-ordering of the clauses and the literals therein). The task now is to find a satisfying assignment β of F such that $\beta \circ \pi_i \succeq_{\text{lex}} \beta$. This is clearly in PLS: whether F is invariant under the π_i and whether β satisfies F are both easy to check. Note that it is not required that β be in the orbit of α under $\langle \pi_1, \dots, \pi_k \rangle$.

► **Theorem 15.** *LOCALLY MINIMAL SOLUTION is PLS-complete.*

Proof. We will define a formula F whose satisfying assignments correspond exactly the well-behaved assignments to the positions, as defined in Definition 8. We first describe how to encode one circuit of the total $n + 1$ circuits. We have input variables x_1, \dots, x_n to the circuit; we introduce one *gate output variable* g_{out} for each gate; and four *gate control variables* $g_{00}, g_{01}, g_{10}, g_{11}$ for the four positions in the square-representation of that gate, as in Figure 3. For each such variable u we introduce its twin \tilde{u} and add $(u \leftrightarrow \neg \tilde{u})$. In other words, the (positive) literal \tilde{u} simulates the negative literal \bar{u} . Take a gate g , let u, v be its inputs and w its output. The following formula F_g ensures that the gate control variables g_{ab} are set correctly as required for a well-behaved assignment:

$$\begin{aligned} (u \wedge v \wedge w) &\rightarrow (\tilde{g}_{00} \wedge \tilde{g}_{01} \wedge \tilde{g}_{10} \wedge g_{11}) \\ (u \wedge v \wedge \tilde{w}) &\rightarrow (g_{00} \wedge g_{01} \wedge g_{10} \wedge \tilde{g}_{11}) \\ (u \wedge \tilde{v} \wedge w) &\rightarrow (\tilde{g}_{00} \wedge \tilde{g}_{01} \wedge g_{10} \wedge \tilde{g}_{11}) \\ (u \wedge \tilde{v} \wedge \tilde{w}) &\rightarrow (g_{00} \wedge g_{01} \wedge \tilde{g}_{10} \wedge g_{11}) \\ (\tilde{u} \wedge v \wedge w) &\rightarrow (\tilde{g}_{00} \wedge g_{01} \wedge \tilde{g}_{10} \wedge \tilde{g}_{11}) \\ (\tilde{u} \wedge v \wedge \tilde{w}) &\rightarrow (g_{00} \wedge \tilde{g}_{01} \wedge g_{10} \wedge g_{11}) \\ (\tilde{u} \wedge \tilde{v} \wedge w) &\rightarrow (g_{00} \wedge \tilde{g}_{01} \wedge \tilde{g}_{10} \wedge \tilde{g}_{11}) \\ (\tilde{u} \wedge \tilde{v} \wedge \tilde{w}) &\rightarrow (\tilde{g}_{00} \wedge g_{01} \wedge g_{10} \wedge g_{11}) \end{aligned}$$

F_g can easily be written as a CNF formula. The permutation π_g amounts to flipping the output of gate g and flipping the corresponding inputs at those gates h that g 's output feeds into. Thus, π_g is

$$(w\tilde{w})(g_{00}\tilde{g}_{00})(g_{01}\tilde{g}_{01})(g_{10}\tilde{g}_{10})(g_{11}\tilde{g}_{11}) \circ (\text{stuff at successor gates}) \quad (2)$$

F_g is invariant under π_g (even when we write it as a CNF). Next, there is a permutation σ that flips the input u of g . This swaps the two rows of the square representation of g :

$$(\text{stuff at predecessor gates}) \circ (u\tilde{u})(g_{00}g_{10})(\tilde{g}_{00}\tilde{g}_{10})(g_{01}g_{11})(\tilde{g}_{01}\tilde{g}_{11}) \quad (3)$$

If u is the output of some gate h , then σ is π_h and “stuff at predecessor gate” is what we describe in (2); otherwise u is an input variable x_i and “stuff at predecessor gate” does nothing – for now.

This formula describes well-behaved assignments in one of the $n + 1$ circuits C_0, \dots, C_n . We now create $n + 1$ copies of this formula, introducing a fresh version of each variable, so the j^{th} input variable x_j becomes $x_j^{(i)}$ in C_i , and g_{00} becomes $g_{00}^{(i)}$ and so on. We create a formula H to ensure that $x_j^{(i)}$ and $x_j^{(0)}$ differ if and only if $i = j$:

$$H := \bigwedge_{\{i_1, i_2\} \in \binom{\{0, \dots, n\}}{2}} \bigwedge_{j=1}^n \left\{ \begin{array}{ll} \left(x_j^{(i_1)} \leftrightarrow x_j^{(i_2)} \right) \wedge \left(\tilde{x}_j^{(i_1)} \leftrightarrow \tilde{x}_j^{(i_2)} \right) & \text{if } j \notin \{i_1, i_2\} \\ \left(x_j^{(i_1)} \leftrightarrow \tilde{x}_j^{(i_2)} \right) \wedge \left(\tilde{x}_j^{(i_1)} \leftrightarrow x_j^{(i_2)} \right) & \text{if } j \in \{i_1, i_2\} . \end{array} \right\} \quad (4)$$

The permutation σ_i flipping circuit C_i and C_0 and inverting $x_i^{(i')}$ for all other i' can be written as

$$\begin{aligned} & \left(u_j^{(i)} u_j^{(0)} \right) \left(\tilde{u}_j^{(i)} \tilde{u}_j^{(0)} \right) \quad (\text{for each input and gate output and control variable } u) \\ & \circ \left(x_i^{(i')} \tilde{x}_i^{(i')} \right) \circ \left(\text{do (3) at gates having } x_i^{(i')} \text{ as input} \right) \quad (\text{for all } i' \in \{1, \dots, n\} \setminus \{i\}) \end{aligned}$$

This forms the final formula $F = H \wedge \bigwedge_{i=0}^n \bigwedge_{\text{gate } g} F_g^{(i)}$. Its satisfying assignments correspond exactly to the well-behaved assignments described above and that each π_g and each σ_i is indeed a symmetry of F . The order of the variables is as in the previous proof: of highest priority are the control variables $g_{11}^{(0)}$ (following the topological order of the gates in the circuit); then the output variables of C_0 ; then to the control variables of the other circuits; then all the rest. It is easy to provide an “initial” satisfying assignment $\alpha \in \text{SAT}(F)$. Finally, if β is some satisfying assignment of F that is locally minimal, i.e., cannot be improved by applying any π_g or σ_i , then β represents a configuration in which all circuits C_0, \dots, C_n are correctly evaluated and no C_i outputs something better than C_0 ; in other words, a local optimum of FLIP. This shows that LOCALLY MINIMAL SOLUTION is PLS-complete. ◀

7 Conclusion and open questions

We have shown that the problem is PLS-hard in general and hence a polynomial time algorithm is unlikely unless $\text{P} = \text{PLS}$. In the theory of PLS-complete problems we thereby demonstrated another example of a hard problem with lexicographic weights. The used permutations are realistic in the sense that they can occur in an actual CNF formula.

Our above reduction requires polynomially many permutations. There is an efficient algorithm for the case of *one* permutation. What about when we are given a constant number of permutations?

What about if the given permutations form an Abelian group? Is it still PLS-hard to find a local minimum?

The neighborhood of a solution π is in our work defined as $\pi \circ \pi_i$ for some generator π_i . An alternative neighborhood would be $\pi_i \circ \pi$, which would place the generator between the string and the current permutation. While the results of the one permutation case trivially hold again, the hardness proof does not transform to this formulation.

References

- 1 László Babai and Eugene M. Luks. Canonical labeling of graphs. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 171–183. ACM, 1983. doi:10.1145/800061.808746.
- 2 Christoph Buchheim and Michael Jünger. Linear optimization over permutation groups. *Discret. Optim.*, 2(4):308–319, 2005. doi:10.1016/J.DISOPT.2005.08.005.
- 3 Dominic Dumrauf and Burkhard Monien. On the PLS-complexity of maximum constraint assignment. *Theor. Comput. Sci.*, 469:24–52, 2013. doi:10.1016/J.TCS.2012.10.044.
- 4 Alex Fabrikant, Christos H. Papadimitriou, and Kunal Talwar. The complexity of pure nash equilibria. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 604–612. ACM, 2004. doi:10.1145/1007352.1007445.
- 5 John Fearnley, Paul W. Goldberg, Alexandros Hollender, and Rahul Savani. The complexity of gradient descent: $\text{CLS} = \text{PPAD} \cap \text{PLS}$. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 46–59. ACM, 2021. doi:10.1145/3406325.3451052.
- 6 Merrick L. Furst, John E. Hopcroft, and Eugene M. Luks. Polynomial-time algorithms for permutation groups. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 36–41. IEEE Computer Society, 1980. doi:10.1109/SFCS.1980.34.
- 7 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction – CADE-25 – 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 591–606. Springer, 2015. doi:10.1007/978-3-319-21401-6_40.
- 8 David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *J. Comput. Syst. Sci.*, 37(1):79–100, 1988. doi:10.1016/0022-0000(88)90046-3.
- 9 Leszek Aleksander Kolodziejczyk and Neil Thapen. The strength of the dominance rule. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPICs*, pages 20:1–20:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICs.SAT.2024.20.
- 10 Mark W. Krentel. On finding locally optimal solutions. In *Proceedings: Fourth Annual Structure in Complexity Theory Conference, University of Oregon, Eugene, Oregon, USA, June 19-22, 1989*, pages 132–137. IEEE Computer Society, 1989. doi:10.1109/SCT.1989.41819.
- 11 Christos H. Papadimitriou. On graph-theoretic lemmata and complexity classes (extended abstract). In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 794–801. IEEE Computer Society, 1990. doi:10.1109/FSCS.1990.89602.
- 12 Christos H. Papadimitriou, Alejandro A. Schäffer, and Mihalis Yannakakis. On the complexity of local search (extended abstract). In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 438–445. ACM, 1990. doi:10.1145/100216.100274.
- 13 Alejandro A. Schäffer and Mihalis Yannakakis. Simple local search problems that are hard to solve. *SIAM J. Comput.*, 20(1):56–87, 1991. doi:10.1137/0220004.
- 14 Neil Thapen. personal communication.
- 15 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014 – 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.