Equality Testing of Compressed Strings

Markus Lohrey(✉)

Universität Siegen, Siegen, Germany
lohrey@eti.uni-siegen.de

Abstract. This paper gives a survey on efficient algorithms for checking equality of grammar-compressed strings, i.e., strings that are represented succinctly by so called straight-line programs.

1 Introduction

The investigation of the computational complexity of algorithmic problems for succinct data started with the work of Galperin and Wigderson [10]. In that paper, a graph with 2^n vertices is represented by a Boolean circuit with $2n$ inputs, and there is an edge between $u \in \{0,1\}^n$ and $v \in \{0,1\}^n$ if and only if the circuit outputs 1 on input u, v. This kind of succinct representation was further investigated in [3,5,30,34]. It turned out that for circuit-encoded graphs, an upgrading theorem holds. Basically, it says that if a graph problem is hard for a certain complexity class, then the succinct version of the problem is hard for the exponentially larger class (precise assumptions on the underlying reductions have to be made).

In this paper, we are concerned with another succinct representation that allows to encode long strings: *straight-line programs*, briefly SLPs. An SLP is a context-free grammar that produces a single string. The length of this string can be exponential in the size of the SLP. Thus, SLPs allow exponential compression in the worst case. There exist several grammar-based compressors that compute from a given input string a small SLP for that string [6].

Another line of research studies algorithmic problems for SLP-compressed strings, see [25] for a survey. In this paper we deal with the equality problem for SLP-compressed strings: Given two SLPs $\mathcal{G}$ and $\mathcal{H}$, we want to check whether the strings produced by $\mathcal{G}$ and $\mathcal{H}$ are equal. We call this problem *compressed equality checking* in the following. Obviously, a simple decompress-and-check strategy that first produces the strings derived by $\mathcal{G}$ and $\mathcal{H}$ and then compares these strings symbol by symbol needs exponential time. Surprisingly, in 1994 three independent conference papers by Plandowski [31], Hirshfeld, Jerrum, and Moller [15] (see [16] for a long version), and Mehlhorn, Sundar, and Uhrig [27] (see [28] for a long version) were published, where polynomial time algorithms for compressed equality checking were presented. Since then, improvements concerning the running time have been achieved in [2,17,23]. The currently fastest

This research is supported by the DFG-project LO 748/10-1.

F. Manea and D. Nowotka (Eds.): WORDS 2015, LNCS 9304, pp. 14–26, 2015.
DOI: 10.1007/978-3-319-23660-5_2

and probably also simplest algorithm is due to Jeż [17] and has a quadratic running time (under some assumptions on the machine model). In Sect. 3 we outline Jeż's algorithm.

Let us remark that both, Plandowski [31] and Hirshfeld et al. [15,16], use Theorem 1 as a tool to solve another problem. Plandowski derives from Theorem 1 a polynomial time algorithm for testing whether two given morphisms (between free monoids) agree on a given context-free language. Hirshfeld et al. use Theorem 1 in order to check bisimilarity of two normed context-free processes (a certain class of infinite state systems) in polynomial time.

The algorithms from [2,16,17,23,28,31] are all sequential, and it is not clear whether any of them allows an efficient parallelization. In fact, it is open whether compressed equality checking belongs to NC or whether it is P-complete. On the other hand, recently a randomized parallel algorithm for compressed (dis)equality checking was presented in [22]. More precisely it was shown that compressed equality checking belongs to the class coRNC^2. The algorithm from [22] reduces compressed equality checking to a restricted form of polynomial identity testing over the polynomial ring $\mathbb{F}_2[x]$. For this restricted form, the identity testing algorithm of Agrawal and Biswas [1] combined with the parallel modular powering algorithm of Fich and Tompa [8] works in coRNC^2. In Sect. 4 we outline this algorithm.

2 Straight-Line Programs

For a string $s \in \Sigma^*$ we denote with $|s|$ the length of s. A factor of s is a string u such that there exist strings x, y with $s = xuy$.

A *straight-line program*, briefly *SLP*, is basically a context-free grammar that produces exactly one string. To ensure this, the grammar has to be acyclic and deterministic (every variable has a unique production where it occurs on the left-hand side. Formally, an SLP is a tuple $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, S)$, where V is a finite set of variables (or nonterminals), Σ is the terminal alphabet, $S \in V$ is the start variable, and rhs maps every variable to a right-hand side $\mathrm{rhs}(A) \in (V \cup \Sigma)^*$. We require that there is a linear order $<$ on V such that $B < A$, whenever B occurs in $\mathrm{rhs}(A)$. Every variable $A \in V$ derives to a unique string $\mathrm{val}_{\mathcal{G}}(A)$ by iteratively replacing variables by the corresponding right-hand sides, starting with A. Finally, the string derived by $\mathcal{G}$ is $\mathrm{val}(\mathcal{G}) = \mathrm{val}_{\mathcal{G}}(S)$.

Let $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, S)$ be an SLP. The *size* of $\mathcal{G}$ is $|\mathcal{G}| = \sum_{A \in V} |\mathrm{rhs}(A)|$, i.e., the total length of all right-hand sides. The SLP $\mathcal{G}$ is in *Chomsky normal form* if for every $A \in V$, $\mathrm{rhs}(A)$ is either a symbol $a \in \Sigma$, or of the form BC, where $B, C \in V$. Every SLP can be transformed in linear time into an SLP in Chomsky normal form that derives the same string.

A simple induction shows that for every SLP $\mathcal{G}$ of size m one has $|\mathrm{val}(\mathcal{G})| \leq \mathcal{O}(3^{m/3})$ [6, proof of Lemma 1]. On the other hand, it is straightforward to define an SLP $\mathcal{H}$ of size $2n$ such that $|\mathrm{val}(\mathcal{H})| \geq 2^n$. This justifies to see an SLP $\mathcal{G}$ as a compressed representation of the string $\mathrm{val}(\mathcal{G})$, and exponential compression rates can be achieved in this way.

An SLP can be also viewed as a multiplicative circuit over a free monoid Σ^*, where the variables are the gates that compute the concatenation of its inputs. This view can be generalized by replacing Σ^* by any finitely generated monoid, see [26]. In algebraic complexity theory, the term "straight-line program" is also used for algebraic circuits that compute (multivariate) polynomials. In such a circuit, every internal gate either computes the sum or the product of its inputs, and the input gates of the circuit are labelled with constants or variables. In Sect. 4 we will use this kind of straight-line programs, and we use the term "algebraic straight-line programs" to distinguish them from string-generating straight-line programs.

Example 1. Consider the SLP $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, A_7)$ with $V = \{A_1, \ldots, A_7\}$, $\Sigma = \{a, b\}$, and the following right-hand side mapping: $\mathrm{rhs}(A_1) = b$, $\mathrm{rhs}(A_2) = a$, and $\mathrm{rhs}(A_i) = A_{i-1}A_{i-2}$ for $3 \le i \le 7$, Then $\mathrm{val}(\mathcal{G}) = abaababaabaab$, which is the 7^{th} Fibonacci string. The SLP $\mathcal{G}$ is in Chomsky normal form and $|\mathcal{G}| = 12$.

One of the most basic tasks for SLP-compressed strings is *compressed equality checking*:
<u>*input*</u>: Two SLPs $\mathcal{G}$ and $\mathcal{H}$
<u>*question*</u>: Does $\mathrm{val}(\mathcal{G}) = \mathrm{val}(\mathcal{H})$ hold?

Clearly, a simple decompress-and-compare strategy is very inefficient. It takes exponential time to compute $\mathrm{val}(\mathcal{G})$ and $\mathrm{val}(\mathcal{H})$. Nevertheless a polynomial time algorithm exists. This was independently discovered by Hirshfeld, Jerrum, and Moller [15,16], Mehlhorn, Sundar, and Uhrig [27,28], and Plandowski [31]:

Theorem 1. *Compressed equality checking can be solved in polynomial time.*

In Sect. 3 we give an outline of the currently fastest (and probably also simplest) algorithm for compressed equality checking, which is due to Jeż [17]. In Sect. 4, we sketch a new approach from [22] that yields a randomized parallel algorithm for compressed equality checking.

3 Sequential Algorithms

The polynomial time compressed equality checking algorithms of Hirshfeld et al. [15,16] and Plandowski [31] use combinatorial properties of strings, in particular the periodicity lemma of Fine and Wilf [9]. This lemma states that if p and q are periods of a string w (i.e., $w[i] = w[i+p]$ and $w[j] = w[j+q]$ for all positions $1 \le i \le |w| - p$ and $1 \le j \le |w| - q$) and $p + q \le |w|$ then also the greatest common divisor of p and q is a period of w. The algorithms from [16,31] achieve a running time of $\mathcal{O}(n^4)$, where $n = |\mathcal{G}| + |\mathcal{H}|$. An improvement to $\mathcal{O}(n^3)$ (for the more general problem of pattern matching), still using the periodicity lemma, was achieved by Lifshits [23].

In contrast to [16,23,31], the algorithm of Mehlhorn et al. [27,28] does not use the periodicity lemma of Fine and Wilf. Actually, in [28], Theorem 1 is not explicitly stated but follows immediately from the main result. Mehlhorn et al. provide an efficient data structure for a finite set of strings that supports the following operations:

- Set variable x to the symbol a.
- Set variable x to the concatenation of the values of variables y and z.
- Split the value of variable x into its length-k prefix and remaining part and store these strings in variables y and z.
- Check whether the values of variables x and y are identical.

The idea is to compute for each variable a signature which is a small number and that allows to do the equality test in constant time. The signature of a string is computed by iteratively breaking up the sequence into small blocks, which are encoded by integers using a pairing function. A single update operation $x := yz$ needs time $\mathcal{O}(\log n(\log m \log^* m + \log n))$ for the m^{th} operation, where n is the length of the resulting string (hence, $\log(n) \leq m$). This leads to a cubic time algorithm for compressed equality checking. An improvement of the data structure from [28] can be found in [2].

The idea from [2,28] of recursively dividing a string into smaller pieces and replacing them by new symbols (integers in [2,28]) was taken up by Jeż, who came up with an extremely powerful technique for dealing with SLP-compressed strings (and the related problem of solving word equations [18]). It also yields the probably simplest proof of Theorem 1. In the rest of the section we briefly sketch this algorithm. We ignore some details. For instance, it is required in [17] that after each step, the terminal alphabet (which gets larger) is an initial segment of the natural numbers, which is ensured by using the radix sort algorithm, see [17] for more details.

Let $s \in \Sigma^+$ be a non-empty string over a finite alphabet Σ. We define the string $\mathsf{block}(s)$ as follows: Assume that $s = a_1^{n_1} a_2^{n_2} \cdots a_k^{n_k}$ with $a_1, \ldots, a_k \in \Sigma$, $a_i \neq a_{i+1}$ for all $1 \leq i < k$ and $n_i > 0$ for all $1 \leq i \leq k$. Then $\mathsf{block}(s) = a_1^{(n_1)} a_2^{(n_2)} \cdots a_k^{(n_k)}$, where $a_1^{(n_1)}, a_2^{(n_2)}, \ldots, a_k^{(n_k)}$ are new symbols. For instance, for $s = aabbbaccb$ we have $\mathsf{block}(s) = a^{(2)} b^{(3)} a^{(1)} c^{(2)} b^{(1)}$. For the symbol $a^{(1)}$ we will simply write a. Let us set $\mathsf{block}(\varepsilon) = \varepsilon$.

For a partition $\Sigma = \Sigma_l \uplus \Sigma_r$ we denote with $s[\Sigma_l, \Sigma_r]$ the string that is obtained from s by replacing every occurrence of a factor ab in s with $a \in \Sigma_l$ and $b \in \Sigma_r$ by the new symbol $\langle ab \rangle$. For instance, for $s = abcbabcad$ and $\Sigma_l = \{a, c\}$ and $\Sigma_r = \{b, d\}$ we have $s[\Sigma_l, \Sigma_r] = \langle ab \rangle \langle cb \rangle \langle ab \rangle c \langle ad \rangle$. Since two different occurrences of factors from $\Sigma_l \Sigma_r$ must occupy disjoints sets of positions in s, the string $s[\Sigma_l, \Sigma_r]$ is well-defined.

Obviously, for all strings $s, t \in \Sigma^*$ we have

$$(s = t \iff \mathsf{block}(s) = \mathsf{block}(t)) \quad \text{and} \quad (s = t \iff s[\Sigma_l, \Sigma_r] = t[\Sigma_l, \Sigma_r]). \quad (1)$$

In the rest of this section, we assume that all SLPs $\mathcal{G} = (V, \Sigma, S, \mathrm{rhs})$ are in a kind of generalized Chomsky normal form: We require that for every variable $A \in V$, $\mathrm{rhs}(A)$ is either of the form $u \in \Sigma^+$, uBv with $u, v \in \Sigma^*$ and $B \in V$, or $uBvCw$ with $u, v, w \in \Sigma^*$ and $B, C \in V$. In other words, every right-hand side is non-empty and contains at most two occurrences of variables. In particular, we only consider SLPs that produce non-empty strings. This is not a crucial restriction for checking the equality $\mathrm{val}(\mathcal{G}) = \mathrm{val}(\mathcal{H})$, since we can first check easily in polynomial time, whether $\mathrm{val}(\mathcal{G})$ or $\mathrm{val}(\mathcal{H})$ produce the empty string.

For the following consideration, it is more convenient to have a single SLP $\mathcal{G}$ with two start variables S_1 and S_2; we write $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, S_1, S_2)$ for such an SLP. For Jeż's algorithm, it is important to assume that S_1 and S_2 do not occur in a right-hand side $\mathrm{rhs}(A)$ $(A \in V)$, which can be easily enforced by renaming variables. Let us write $\mathrm{val}_i(\mathcal{G}) = \mathrm{val}_{\mathcal{G}}(S_i)$ for $i \in \{1, 2\}$. Moreover, w.l.o.g. we always assume that $|\mathrm{val}_1(\mathcal{G})| \leq |\mathrm{val}_2(\mathcal{G})|$ (this property can be easily verified). The goal is to check $\mathrm{val}_1(\mathcal{G}) = \mathrm{val}_2(\mathcal{G})$ for a given SLP $\mathcal{G}$. Jeż's strategy [17] for checking this equality is to compute from $\mathcal{G}$ an SLP $\mathcal{H}$ such that $\mathrm{val}_i(\mathcal{H}) = (\mathsf{block}(\mathrm{val}_i(\mathcal{G})))[\Sigma_l, \Sigma_r]$ for $i \in \{1, 2\}$ and $\mathrm{val}_1(\mathcal{H}) \leq c \cdot |\mathrm{val}_1(\mathcal{G})|$ for some constant $c < 1$. This process is iterated. After at most $\log |\mathrm{val}_1(\mathcal{G})| \in \mathcal{O}(|\mathcal{G}|)$ many iterations it must terminate with an SLP such that $\mathrm{val}_1(\mathcal{G})$ has length one. Checking equality of the two strings produced by this SLP is easy. The main difficulty of this approach is to bound the size of the SLP during this process.

In the following, for an SLP $\mathcal{G}$ we denote with $|\mathcal{G}|_0$ (resp., $|\mathcal{G}|_1$) the total number of occurrences of terminal symbols (resp., nonterminal symbols) in all right-hand sides of $\mathcal{G}$. Thus, $|\mathcal{G}| = |\mathcal{G}|_0 + |\mathcal{G}|_1$. Moreover, let $\mathsf{Var}(\mathcal{G})$ be the set of variables of $\mathcal{G}$. For block compression, we have:

Lemma 1. *There is an algorithm* CompressBlocks *that gets as input an SLP* $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, S_1, S_2)$ *and computes in time* $\mathcal{O}(|\mathcal{G}|)$ *an SLP* $\mathcal{H}$ *such that the following properties hold, where* $k = |V|$*:*

- $S_1, S_2 \in \mathsf{Var}(\mathcal{H}) \subseteq V$,
- $\mathrm{val}_i(\mathcal{H}) = \mathsf{block}(\mathrm{val}_i(\mathcal{G}))$ *for* $i \in \{1, 2\}$,
- $|\mathcal{H}|_1 \leq |\mathcal{G}|_1 \leq 2k$ *and* $|\mathcal{H}|_0 \leq |\mathcal{G}|_0 + 4k$.

Note that this implies in particular that $\mathsf{block}(\mathrm{val}_i(\mathcal{G}))$ cannot contain too many different symbols. In fact, it is not hard to show that $\mathsf{block}(\mathrm{val}_i(\mathcal{G}))$ contains at most $|\mathcal{G}|$ many different symbols. For the proof of Lemma 1 one processes the SLP $\mathcal{G}$ bottom-up, i.e., if B occurs in the right-hand side of A, then A has to be processed before B. When A is processed, we remove from $\mathrm{rhs}(A)$ the maximal prefix (resp., suffix) of the form a^n and insert in front of (resp., after) every occurrence of A in a right-hand side the block a^n. When all nonterminals (except of S_1 and S_2) are processed then every maximal block a^n in a right-hand side is replaced by the single letter $a^{(n)}$.

After block compression the two strings produced by the SLP $\mathcal{G}$ do not contain a factor of the form aa. For a string $s \in \Sigma^*$ that does not contain a factor of the form aa $(a \in \Sigma)$, a simple probabilistic argument shows that there exists a partition $\Sigma = \Sigma_l \uplus \Sigma_r$ such that $s[\Sigma_l, \Sigma_r]$ is by a constant factor ($\frac{3}{4}$ up to the additive constant $\frac{1}{4}$) shorter than s. Using a standard derandomization, this partition can be computed in linear time. Moreover, using a technique for counting digrams in SLP-compressed strings (see also [13]), one can compute the partition in linear time even if s is succinctly represented by an SLP $\mathcal{G}$. Once the partition $\Sigma = \Sigma_l \uplus \Sigma_r$ is computed, one can prove the following lemma in a similar way as Lemma 1.

Algorithm 1. CheckEquality

```
Data: SLP $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, S_1, S_2)$ such that $|\mathrm{val}_1(\mathcal{G})| \leq |\mathrm{val}_2(\mathcal{G})|$
while $|\mathrm{val}_1(\mathcal{G})| > 1$ do
    $\mathcal{G} := \mathrm{CompressPairs}(\mathrm{CompressBlocks}(\mathcal{G}))$
end
check whether $\mathrm{val}_1(\mathcal{G}) = \mathrm{val}_2(\mathcal{G})$
```

Lemma 2. *There is an algorithm* CompressPairs *that gets as input an SLP* $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, S_1, S_2)$ *such that for* $i \in \{1, 2\}$, $\mathrm{val}_i(\mathcal{G})$ *does not contain a factor* aa *(*$a \in \Sigma$*) and computes in time* $\mathcal{O}(|\mathcal{G}|)$ *a partition* $\Sigma = \Sigma_\ell \uplus \Sigma_r$ *and an SLP* $\mathcal{H}$ *such that the following properties hold, where* $k = |V|$*:*

- $S_1, S_2 \in \mathsf{Var}(\mathcal{H}) \subseteq V$,
- $\mathrm{val}_i(\mathcal{H}) = \mathrm{val}_i(\mathcal{G})[\Sigma_l, \Sigma_r]$ *for* $i \in \{1, 2\}$,
- $|\mathcal{H}|_1 \leq |\mathcal{G}|_1 \leq 2k$ *and* $|\mathcal{H}|_0 \leq |\mathcal{G}|_0 + 4k$,
- $|\mathrm{val}_1(\mathcal{H})| \leq \frac{1}{4} + \frac{3}{4}|\mathrm{val}_1(\mathcal{G})|$.

Using Lemmas 1 and 2 we can prove Theorem 1: Assume that we have an SLP $\mathcal{G} = (V, \Sigma, \mathrm{rhs}, S_1, S_2)$ over the terminal alphabet Σ, where $m := |\mathrm{val}_1(\mathcal{G})| \leq |\mathrm{val}_2(\mathcal{G})|$. Moreover, let $k = |V|$. Algorithm 1 checks whether $\mathrm{val}_1(\mathcal{G}) = \mathrm{val}_2(\mathcal{G})$. Correctness of the algorithm follows from observation (1). It remains to analyze the running time of the algorithm. By the last point from Lemma 2, the number of iterations of the while loop is bounded by $\mathcal{O}(\log(m)) \leq \mathcal{O}(|\mathcal{G}|)$. Let $\mathcal{G}_i$ be the SLP after i iterations of the while loop. The number of variables of $\mathcal{G}_i$ is at most k. Hence, by Lemmas 1 and 2 we have $|\mathcal{G}_i| \leq |\mathcal{G}| + 8ki \in \mathcal{O}(|\mathcal{G}|^2)$. Since the i-th iteration takes time $\mathcal{O}(|\mathcal{G}_i|)$, the total running time is $\mathcal{O}(|\mathcal{G}|^3)$.

There is a simple way to improve the running time to $\mathcal{O}(|\mathcal{G}|^2)$ (under some assumptions on the underlying machine model). In the above calculation we ignored the fact that block and pair compression also reduce the size of the SLP. To make use of this, we modify Algorithm 1 as follows. In every second iteration of the while loop, we choose the partition (Γ_l, Γ_r) for pair compression according to the following variant of Lemma 2:

Lemma 3. *There is an algorithm* CompressPairs′ *with the same properties as algorithm* CompressPairs *from Lemma 2 except that the last property* $|\mathrm{val}_1(\mathcal{H})| \leq \frac{1}{4} + \frac{3}{4}|\mathrm{val}_1(\mathcal{G})|$ *in Lemma 2 is replaced by* $|\mathcal{H}|_0 \leq \frac{3}{4}k + 4k + \frac{3}{4}|\mathcal{G}|_0$.

The proof of this lemma is the same as for Lemma 2, except that the partition $\Sigma = \Sigma_\ell \cup \Sigma_r$ is chosen in such a way that the maximal factors from Σ^* in right-hand sides of $\mathcal{G}$ (there are at most $3k$ such factors since every right-hand side contains at most two variables) contain many factors of the form ab with $a \in \Sigma_l$, $b \in \Sigma_r$ (see Claim 1 in the proof of [19, Lemma 6] for a more precise statement).

Since Lemma 3 is used in every second iteration of the while loop, we get

$$|\mathcal{G}_{i+2}|_0 \leq \frac{3}{4}k + 4k + \frac{3}{4}(|\mathcal{G}_i|_0 + 12k) = \mathcal{O}(|\mathcal{G}|) + \frac{3}{4}|\mathcal{G}_i|_0$$

for every even i (we add the $12k$ since we apply CompressBlocks twice and CompressPairs once, before we apply CompressPairs$'$). A simple calculation shows that $|\mathcal{G}_i|_0 \in \mathcal{O}(|\mathcal{G}|)$ and hence $|\mathcal{G}_i| \in \mathcal{O}(|\mathcal{G}|)$ for all $i \geq 0$. The number of iterations of the while loop is still bounded by $\mathcal{O}(\log(m))$. This yields the running time $\mathcal{O}(\log(m) \cdot |\mathcal{G}|) \leq \mathcal{O}(|\mathcal{G}|^2)$.

This concludes our outline of Jeż's algorithm. We ignored some issues related to the machine model. More precisely, the time bound $\mathcal{O}(|\mathcal{G}|^2)$ only holds if the length m fit into a single machine word, see [17] for details. Let us finally remark that Jeż [17] obtains his result for the more general problem of fully compressed pattern matching, see Sect. 5.

Randomized algorithms for compressed equality checking are studied in [12, 32]. These algorithms are based on arithmetic modulo small prime numbers. The algorithm from [32] has a quadratic running time under the RAM model with *logarithmic cost measure*, which means that arithmetic operations on n-bit numbers need time $\mathcal{O}(n)$. If $\mathrm{val}(\mathcal{G}) = \mathrm{val}(\mathcal{H})$ then the algorithm will correctly output "yes"; if $\mathrm{val}(\mathcal{G}) \neq \mathrm{val}(\mathcal{H})$ then the algorithm may incorrectly output "yes" with a small error probability. In the next section, we will outline another randomized algorithm for compressed equality checking that allows an efficient parallelization.

4 A Parallel Algorithm

The polynomial time algorithms from [2,16,17,28,31] for compressed equality checking are all sequential, and it is not clear whether one of them allows a parallel implemention. It is in fact open, whether compressed equality checking belongs to the class NC of all problems that can be solved on a PRAM in polylogarithmic time using only polynomially many processors. In this section, we sketch a randomized parallel algorithm that was recently discovered in [22].

We use standard definitions concerning circuit complexity, see e.g. [35] for more details. In particular we will consider the class NC^i of all problems that can be solved by a circuit family $(\mathcal{C}_n)_{n \geq 1}$, where the size of $\mathcal{C}_n$ (the circuit for length-n inputs) is polynomially bounded in n, its depth is bounded by $\mathcal{O}(\log^i n)$, and $\mathcal{C}_n$ is built from input gates, NOT-gates and AND-gates and OR-gates of fan-in two. The class NC is the union of all classes NC^i. We assume circuit families to be logspace-uniform, which means that the mapping $a^n \mapsto \mathcal{C}_n$ can be computed in logspace.

To define a randomized version of NC^i, one uses circuit families with additional inputs. So, let the n^{th} circuit $\mathcal{C}_n$ in the family have n normal input gates plus m random input gates, where m is polynomially bounded in n. For an input $x \in \{0,1\}^n$ one defines the acceptance probability as

$$\mathsf{Prob}[\mathcal{C}_n \text{ accepts } x] = \frac{|\{y \in \{0,1\}^m \mid \mathcal{C}_n(x,y) = 1\}|}{2^m}.$$

Here, $\mathcal{C}_n(x,y) = 1$ means that the circuit $\mathcal{C}_n$ evaluates to 1 if the i^{th} normal input gate gets the i^{th} bit of the input string x, and the i^{th} random input gate gets the

i^{th} bit of the random string y. Then, the class RNC^i is the class of all problems A for which there exists a polynomial size circuit family $(\mathcal{C}_n)_{n\geq 0}$ of depth $\mathcal{O}(\log^i n)$ with random input gates that uses NOT-gates and AND-gates and OR-gates of fan-in two, such that for all inputs $x \in \{0,1\}^*$ of length n: (i) if $x \in A$, then $\mathsf{Prob}[\mathcal{C}_n$ accepts $x] \geq 1/2$, and (ii) if $x \notin A$, then $\mathsf{Prob}[\mathcal{C}_n$ accepts $x] = 0$. As usual, coRNC^i is the class of all complements of problems from RNC^i. Section B.9 in [14] contains several problems that are known to be in RNC, but which are not known to be in NC; the most prominent example is the existence of a perfect matching in a graph.

In this section, we will sketch a proof of the following result that was recently shown in [22]:

Theorem 2. *Compressed equality checking belongs to* coRNC^2.

Let us assume that we have a single SLP $\mathcal{G}$ in Chomsky normal form, and we want to check whether $\text{val}_{\mathcal{G}}(X) = \text{val}_{\mathcal{G}}(Y)$ for two variables X, Y. Without loss of generality we can assume that the terminal alphabet of $\mathcal{G}$ is $\{0,1\}$. In a first step, we compute the length $|\text{val}_{\mathcal{G}}(A)|$ for every variable A. For this, one has to evaluate addition circuits over the natural numbers, which is possible in NC^2 (see [22] for details). If $|\text{val}_{\mathcal{G}}(X)| \neq |\text{val}_{\mathcal{G}}(X)|$, then we reject. So, let us assume that $|\text{val}_{\mathcal{G}}(X)| = |\text{val}_{\mathcal{G}}(X)|$. We omit the index $\mathcal{G}$ in the following.

As for many other randomized algorithms (also the RNC-algorithm for the existence of a perfect matching in a graph) we now shift the problem to an algebraic problem about polynomials. A string $w = a_0 a_1 \cdots a_n \in \{0,1\}^*$ with $a_i \in \{0,1\}$ can be encoded as the polynomial

$$p_w(x) = \sum_{i=0}^{n} a_i \cdot x^i \in \mathbb{F}_2[x]$$

over the field $\mathbb{F}_2$. Clearly, if $|u| = |v|$, then $u = v$ if and only if $p_u(x) + p_v(x) = 0$. Hence, it remains to check whether $p_{\text{val}(X)}(x) + p_{\text{val}(Y)}(x)$ is the zero polynomial.

The polynomial $p_{\text{val}(X)}(x) + p_{\text{val}(Y)}(x)$ has exponential degree, but it can be defined by a small algebraic circuit, or equivalently, a small algebraic straight-line program (ASLP). ASLPs are defined analogously to our SLPs for strings, but variables evaluate to polynomials from $\mathbb{F}_2[x]$ (or another polynomial ring, in general). In right-hand sides, the operations of (polynomial) addition and multiplication as well as the constants $0, 1, x$ can be used. We translate the SLP $\mathcal{G}$ into an ASLP $\mathcal{H}$ for the polynomial $p_{\text{val}(X)}(x) + p_{\text{val}(Y)}(x)$ as follows: If $\text{rhs}(A) = a \in \{0,1\}$ in $\mathcal{G}$, then also $\text{rhs}(A) = a$ in $\mathcal{H}$, and if $\text{rhs}(A) = BC$ in $\mathcal{G}$, then $\text{rhs}(A) = B + x^n \cdot C$ in $\mathcal{H}$, where $n = |\text{val}_{\mathcal{G}}(B)|$ (this length can be precomputed in NC^2). Finally, we add a new start variable S to $\mathcal{H}$ and set $\text{rhs}(S) = X + Y$. It is easy to check, that $\mathcal{H}$ indeed produces the polynomial $p_{\text{val}(X)}(x) + p_{\text{val}(Y)}(x)$.

Note that the right-hand side $B + x^n \cdot C$ contains a big power of x. On the other hand, n is only exponential in the input size and could be replaced by a chain of multiplications. Testing whether $\mathcal{H}$ produces the zero polynomial is an instance of *polynomial identity testing* (PIT) over the ring $\mathbb{F}_2[x]$. This is a famous problem in complexity theory, which is known to be in coRP (the class of complements of problems from randomized polynomial time), but for which no polynomial time algorithm is known. Moreover, proving PIT to be in P would prove circuit lower bounds that currently seem to be out of reach, see [20].

So far, we have put compressed equality checking into the class coRP only. To lower this bound to coRNC^2 we have to make use of the particular form of the ASLPs in our situation. The right-hand side $B + x^n \cdot C$ can be replaced by $B + D$, where D is a fresh variable with $\mathrm{rhs}(D) = x^n \cdot C$. We now have obtained an ASLP $\mathcal{H}$, where every right-hand side has one of the following forms:

- a constant $a \in \{0, 1\}$,
- an addition $B + C$ of two variables B, C,
- a multiplication $x^n \cdot C$, where C is a variable and n is a number that is encoded in binary representation.

In [22] we called such an ASLP *powerful skew*: In a skew ASLP (or skew algebraic circuit), for every multiplication one of the two arguments has to be a constant or the variable x. The additional adjective "powerful" refers to the fact one of the arguments of every multiplication gate is a power of x, where the exponent is given in binary notation.

To test $\mathrm{val}(\mathcal{H}) = 0$ for a powerful skew ASLP, we can use any of the randomized PIT-algorithms. In order to get a coRNC^2-algorithm, the identity testing algorithm of Agrawal and Biswas [1] is the right choice. This algorithm computes the polynomial $\mathrm{val}(\mathcal{H})$ modulo a test polynomial $P(x) \in \mathbb{F}_2[x]$ of polynomial degree, which is randomly chosen from a suitable test space. Clearly, if $\mathrm{val}(\mathcal{H}) = 0$, then also $\mathrm{val}(\mathcal{H}) \bmod P(x) = 0$. On the other hand, by the specific choice of the test space, if $\mathrm{val}(\mathcal{H})$ is not the zero polynomial, then also $\mathrm{val}(\mathcal{H}) \bmod P(x)$ is not the zero polynomial with high probability. This part of the algorithm is the only place, were we use randomness.

It finally remains to compute $\mathrm{val}(\mathcal{H}) \bmod P(x)$, which will be done in NC^2, using the modular powering algorithm of Fich and Tompa [8]. More precisely, Fich and Tompa proved in [8] that the following problem can be solved in NC^2 (we only present here the result for the polynomial ring $\mathbb{F}_2[x]$, but in [8] a more general version is shown):

<u>*input*</u>: polynomials $p(x), q(x) \in \mathbb{F}_2[x]$ and a binary encoded natural number n.
<u>*output*</u>: $p(x)^n \bmod q(x)$

Using this result, we can replace in the ASLP $\mathcal{H}$ every power x^n by $x^n \bmod P(x)$ in NC^2. The resulting ASLP computes the same polynomial as $\mathcal{H}$ modulo $P(x)$. Moreover, the big powers x^n in right-hand sides of the form $x^n \cdot B$ are replaced by polynomials of polynomially bounded degree. This allows to compute the output polynomial explicitly in NC^2 using a standard reduction to matrix powering, see [22] for details. We still have to compute this output polynomial modulo $P(x)$, which can be done in NC^1 [7]. This concludes our proof sketch for Theorem 2.

The coRNC^2-algorithm for compressed equality checking easily generalizes to equality checking for SLP-compressed 2-dimensional pictures (and in fact, pictures of any dimension). Such a 2-dimensional picture is a rectangular array of symbols from a finite alphabet. To define 2-dimensional SLPs, one uses a horizontal and a vertical concatenation operation, which are both partially defined (for horizontal concatenation, the two pictures need to have the same height, and for vertical concatenation, the two pictures need to have the same width). This formalism was studied in [4], where it was shown that equality of SLP-compressed 2-dimensional pictures belongs to coRP using a reduction to PIT. Using the above technique, this bound was reduced to coRNC^2 in [22]. It is still open, whether equality of SLP-compressed 2-dimensional pictures can be checked in polynomial time.

5 Related Problems

A natural generalization of checking equality of two strings is pattern matching. In the classical *pattern matching problem* it is asked for given strings p (usually called the pattern) and t (usually called the text), whether p is a factor of t. There are many linear time algorithms for this problem on uncompressed strings. It is therefore natural to ask, whether a polynomial time algorithm for pattern matching on SLP-compressed strings exists. This problem is sometimes called *fully compressed pattern matching* and is defined as follows:

input: Two SLPs $\mathcal{P}$ and $\mathcal{T}$
question: Is $\mathrm{val}(\mathcal{P})$ a factor of $\mathrm{val}(\mathcal{T})$?

The first polynomial time algorithm for fully compressed pattern matching was presented in [21] by Karpinski, Rytter, and Shinohara. Further improvements with respect to the running time were achieved in [11,17,23,29]. The algorithms from [11,21,23,29] use the periodicity lemma of Fine and Wilf, similarly to the solutions of Plandowski and Hirshfeld et al. for compressed equality checking. In contrast, Jeż's algorithm from [17] is based on his recompression technique and is a refinement of the algorithm sketched in the previous section. It is the currently fastest algorithm. Its running time is $\mathcal{O}((|\mathcal{T}|+|\mathcal{P}|)\cdot\log|\mathrm{val}(\mathcal{P})|)$ under the assumption that $|\mathrm{val}(\mathcal{P})|$ can be stored in a single machine word, otherwise an additional factor $\log(|\mathcal{T}|+|\mathcal{P}|)$ goes in.

Let us finally mention a result of Lifshits [24], which together with Theorem 1 gives an impression of the subtle borderline between tractability and intractability for problems on SLP-compressed strings. A function $f : \Sigma^* \to \mathbb{N}$ belongs to the counting class $\#\mathsf{P}$ if there exists a nondeterministic polynomial time bounded Turing-machine M such that for every $x \in \Sigma^*$, $f(x)$ equals the number of accepting computation paths of M on input x. A function $f : \Sigma^* \to \mathbb{N}$ is $\#\mathsf{P}$-complete if it belongs to $\#\mathsf{P}$ and for every $\#\mathsf{P}$-function $g : \Gamma^* \to \mathbb{N}$ there is a logspace computable mapping $h : \Gamma^* \to \Sigma^*$ such that $h \circ f = g$. Functions that are $\#\mathsf{P}$-complete are computationally very powerful. By a famous result of Toda [33], every language from the polynomial time hierarchy can be decided in deterministic polynomial time with the help of a $\#\mathsf{P}$-function, i.e., $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$. For two strings $u = a_1 \cdots a_n$ and $v = b_1 \cdots b_n$ of the same length n, the *Hamming-distance* $d_H(u, v)$ is the number of positions $i \in \{1, \ldots, n\}$ such that $a_i \neq b_i$.

Theorem 3 ([24]). *The mapping* $(\mathcal{G}, \mathcal{H}) \mapsto d_H(\mathrm{val}(\mathcal{G}), \mathrm{val}(\mathcal{H}))$*, where* $\mathcal{G}$ *and* $\mathcal{H}$ *are SLPs is* $\#\mathsf{P}$*-complete.*

6 Open Problems

The main open problem in the context of compressed equality checking is the precise complexity of this problem. Theorem 2 suggests that compressed equality checking is not P-complete (showing $\mathsf{P} \subseteq \mathsf{RNC}$ would be a big surprise). Hence, we conjecture that compressed equality checking belongs to NC. For fully compressed pattern matching it is even open whether the problem belongs to coRNC (or RNC).

Another open problem is whether the quadratic running time of Jeż's algorithm for compressed equality checking can be further improved.

References

1. Agrawal, M., Biswas, S.: Primality and identity testing via chinese remaindering. J. Assoc. Comput. Mach. **50**(4), 429–443 (2003)
2. Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts. In: Proceedings of SODA 2000, pp. 819–828. ACM/SIAM (2000)
3. Balcázar, J.L.: The complexity of searching succinctly represented graphs. In: Fülöp, Z. (ed.) ICALP 1995. LNCS, vol. 944, pp. 208–219. Springer, Heidelberg (1995)
4. Berman, P., Karpinski, M., Larmore, L.L., Plandowski, W., Rytter, W.: On the complexity of pattern matching for highly compressed two-dimensional texts. J. Comput. Syst. Sci. **65**(2), 332–350 (2002)
5. Borchert, B., Lozano, A.: Succinct circuit representations and leaf language classes are basically the same concept. Inf. Process. Lett. **59**(4), 211–215 (1996)
6. Charikar, M., Lehman, E., Lehman, A., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory **51**(7), 2554–2576 (2005)
7. Eberly, W.: Very fast parallel polynomial arithmetic. SIAM J. Comput. **18**(5), 955–976 (1989)
8. Fich, F.E., Tompa, M.: The parallel complexity of exponentiating polynomials over finite fields. In: Proceedings of STOC 1985, pp. 38–47. ACM (1985)
9. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proc. Am. Math. Soc. **16**, 109–114 (1965)
10. Galperin, H., Wigderson, A.: Succinct representations of graphs. Inf. Control **56**(3), 183–198 (1983)
11. Gasieniec, L., Karpinski, M., Plandowski, W., Rytter, W.: Efficient algorithms for Lempel-Ziv encoding (extended abstract). In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, pp. 392–403. Springer, Heidelberg (1996)
12. Gasieniec, L., Karpinski, M., Plandowski, W., Rytter, W.: Randomized efficient algorithms for compressed strings: the finger-print approach (extended abstract). In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 39–49. Springer, Heidelberg (1996)

13. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Fast q-gram mining on SLP compressed strings. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 278–289. Springer, Heidelberg (2011)
14. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: Limits to Parallel Computation: P-Completeness Theory. Oxford University Press, Oxford (1995)
15. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial-time algorithm for deciding equivalence of normed context-free processes. In: Proceedings of FOCS 1994, pp. 623–631. IEEE Computer Society (1994)
16. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial algorithm for deciding bisimilarity of normed context-free processes. Theor. Comput. Sci. **158**(1&2), 143–159 (1996)
17. Jeż, A.: Faster fully compressed pattern matching by recompression. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 533–544. Springer, Heidelberg (2012)
18. Jeż, A.: Recompression: a simple and powerful technique for word equations. In: Proceedings of STACS 2013. LIPIcs, vol. 20, pp. 233–244. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013)
19. Jeż, A., Lohrey, M.: Approximation of smallest linear tree grammars. Technical report, arXiv.org (2014). http://arxiv.org/abs/1309.4958
20. Kabanets, V., Impagliazzo, R.: Derandomizing polynomial identity tests means proving circuit lower bounds. Comput. Complex. **13**(1–2), 1–46 (2004)
21. Karpinski, M., Rytter, W.: Pattern-matching for strings with short descriptions. In: Galil, Z., Ukkonen, E. (eds.) CPM 1995. LNCS, vol. 937, pp. 205–214. Springer, Heidelberg (1995)
22. König, D., Lohrey, M.: Parallel identity testing for skew circuits with big powers and applications. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9235, pp. 445–458. Springer, Heidelberg (2015)
23. Lifshits, Y.: Processing compressed texts: a tractability border. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 228–240. Springer, Heidelberg (2007)
24. Lifshits, Y., Lohrey, M.: Querying and embedding compressed texts. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 681–692. Springer, Heidelberg (2006)
25. Lohrey, M.: Algorithmics on SLP-compressed strings: a survey. Groups Complex. Cryptology **4**(2), 241–299 (2012)
26. Lohrey, M.: The Compressed Word Problem for Groups. SpringerBriefs in Mathematics. Springer, Heidelberg (2014)
27. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality-tests in polylogarithmic time. In: Proceedings of SODA 1994, pp. 213–222. ACM/SIAM (1994)
28. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997)
29. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight-line programs. In: Hein, J., Apostolico, A. (eds.) CPM 1997. LNCS, vol. 1264, pp. 1–11. Springer, Heidelberg (1997)
30. Papadimitriou, C.H., Yannakakis, M.: A note on succinct representations of graphs. Inform. Control **71**(3), 181–185 (1986)
31. Plandowski, W.: Testing equivalence of morphisms on context-free languages. In: van Leeuwen, J. (ed.) ESA 1994. LNCS, vol. 855, pp. 460–470. Springer, Heidelberg (1994)
32. Schmidt-Schauß, M., Schnitger, G.: Fast equality test for straight-line compressed strings. Inf. Process. Lett. **112**(8–9), 341–345 (2012)

33. Toda, S.: PP is as hard as the polynomial-time hierarchy. SIAM J. Comput. **20**(5), 865–877 (1991)
34. Veith, H.: Succinct representation, leaf languages, and projection reductions. Inf. Comput. **142**(2), 207–236 (1998)
35. Vollmer, H.: Introduction to Circuit Complexity. Springer, Heidelberg (1999)