Subsequence Matching and Analysis Problems for Formal Languages

Jane Open Access
Dummy University Computing Laboratory, [optional: Address], Country
My second affiliation, Country

Joan R. Public[1]
Department of Informatics, Dummy College, [optional: Address], Country

Abstract

To be added.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Jane Open Access*: (Optional) author-specific funding acknowledgements
Joan R. Public: [funding]

Acknowledgements I want to thank …

1 Introduction

A word v is a subsequence of a word w, denoted $v \leq w$ in the following, if there exist (possibly empty) words $x_1, \ldots, x_{\ell+1}$ and $v_1, \ldots, v_\ell$ such that $v = v_1 \cdots v_\ell$ and $w = x_1 v_1 \cdots x_\ell v_\ell x_{\ell+1}$. In other words, v can be obtained from w by removing some of its letters.

The concept of subsequence appears and plays important roles in many different areas of theoretical computer science. Prime examples are combinatorics on words, formal languages, automata theory, and logics, where subsequences are studied in connection to piecewise testable languages [59, 58, 31, 32, 34], in connection to subword-order and downward-closures [25, 36, 35, 62, 63, 5]), in connection to binomial equivalence, binomial complexity, or to subword histories [52, 21, 39, 38, 56, 44, 53]. Subsequences are important objects of study also in the area of algorithm-design and complexity; to this end, we mention some classical algorithmic problems such as the computation of *longest common subsequences* or of the *shortest common supersequences* [**?**, **?**, **?**, 43, **?**, **?**, 7, **?**], the testing of the Simon congruence of strings and the computation of the arch-factorisation and universality of words [26, 22, 60, 61, 17, 9, **?**, 19, 23, **?**]; see [**?**] for a survey on combinatorial pattern matching problems related to subsequences. Moreover, these algorithmic problems and other closely related ones recently regained interest in the context of fine-grained complexity [12, 13, 3, 1, 2]. Nevertheless, subsequences appear also in more applied settings: for modelling concurrency [51, 57, 14], in database theory (especially *event stream processing* [6, 24, 64]), in data mining [40, 41], or bioinformatics [11].

The focus of this paper is the study of the subsequences of words of a formal language; the main idea behind it being to extend the fundamental problems related to matching subsequences in a string and to the analysis of the sets of subsequences of a single string to the case of sets of strings. To this end, grammars (or automata) are succinct representations of the (finite or infinite) sets of words they generate (resp., accept), so we are interested in matching and analysis problems related to the set of subsequences of the words of a language, given by the grammar generating it (resp., the automaton accepting it). This research direction is, clearly, not new. To begin with, we recall the

[1] Optional footnote, e.g. to mark corresponding author

famous result of Higman [28] which states that the downward closure of every language (i.e., the set of all subsequences of the words of the respective language) is regular. Clearly, it is not always possible to compute an automaton accepting the downward closure of a given language, but gaining a better understanding when it is computable is an important area of research, as the set of subsequences of a language plays meaningful roles in practical applications (e.g., abstractions of complex systems, see [62, 63, 5] and the references therein). Computing the downward closure of a language is a general (although, often inefficient) way to solve subsequence-matching problems for languages; for instance, we can immediately check, using a finite automaton for the downward closure, if a word occurs as subsequence of a word of the respective language. However, it is often the case that more complex analysis problems regarding the subsequences occurring in the words of a language cannot be solved efficiently (or, sometimes, at all) using the downward closure; such a problem is to check if a given word occurs as subsequence in all the words of a language (chosen from a complex enough class, such as the class of context-free languages).

As a direct predecessor of this paper, motivated by similar questions, [4] approached algorithmic matching and analysis problems related to the universality of regular languages (for short, REG). More precisely, a word over Σ is called k-universal if its set of subsequences includes all words of length k over Σ; the study of these universal words was the focus of many recent works [33, 54, 8, 18, 55] and the motivation for studying universality properties in the context of subsequences is discussed in detail in, e.g., [18, 4]. The main problems addressed in [4] are the following: for $L \in$ REG, over the alphabet Σ, and a number k, decide if there exists a k-universal word in L (resp., if all words of L are k-universal). The authors of [4] discussed efficient algorithms solving these problems and complexity lower bounds. In this paper, we extend the work of [4] firstly by proposing a more structured approach for the algorithmic study of the subsequences occurring in words of formal languages and secondly by considering more general classes of languages, both from the Chomsky hierarchy (such as the class of context-free languages or that of context-sensitive languages) and non-classical (the class of languages accepted by deterministic finite automata with translucent letters).

Our work on subsequence-matching and analysis problems in languages defined by context-free grammars also extends a series of results related to matching subsequences in words given as a straight line program (for short, SLP; a context-free grammar generating a single word), or checking whether a word given as a SLP is k-universal for some given k, see [**?**, 55]. In our paper, we consider the case when the input context-free languages and the contex-free grammars generating them are unrestricted.

The approached problems and an overview of our results: As mentioned above, we propose a more structured approach for matching- and analysis-problems related to subsequences of the words of a formal language. More precisely, we defined and investigate the following five problems.

▶ **Problem 1** (∃-Subsequence). *Given a machine/grammar M and a word w, is there a word $v \in L(M)$ such that $w \leq v$?*

▶ **Problem 2** (∀-Subsequence). *Given a machine/grammar M and a word w, do we have for all words $v \in L(M)$ that $w \leq v$?*

▶ **Problem 3** (∃-k-universal). *Given a machine/grammar M and integer k, check if there is a k-universal word in $L(M)$?*

▶ **Problem 4** (∀-k-universal). *Given a machine/grammar M and integer k, check if all words of $L(M)$ are k-universal.*

Alternatively, strictly from the point of view of designing an algorithmic solution, the problem above can be approached via its complement: that is, deciding if there exists at least one word in $L(M)$ which is not k-universal.

▶ **Problem 5** (∞-universal). *Given a machine/grammar M decide if there exist m-universal words in $L(M)$, for all positive integers m.*

To give some intuition on our terminology, Problems 1 and 3 can be seen as *matching problems* (find a word which contains a certain subsequence or set of subsequences), while the other three problems are *analysis problems* (decide properties concerning multiple words of the language).

Going a bit more into details, in the main part of this paper, we investigate these problems for the case when the language L is chosen from the class of context-free (for short, CF) languages (given by a CF grammar in Chomsky normal form), or from the class of context-sensitive (for short, CS) languages (given by a CS grammar), or from the class of languages accepted by deterministic finite automata with translucent letters (given by an automaton of the respective kind). The choice of presentation of the languages from given classes, unsurprusingly, makes a big difference with respect to hardness. For instance, certain singleton languages can be encoded by straight-line programs (essentially CF grammars) exponentially more succinctly than by classical DFA, which of course introduces significant extra computation into solving subsequence-related queries. But, before approaching these classes of languages, we provide a series of general decidability results on these five problems, for which the choice of grammar or automaton as the way of specifying the input language L is not consequential.

Some relevant citation?

For short, our results are the following. We first give (in Section 3) a series of simple sufficient conditions on the class $\mathcal{C}$ (related to the computation of downward closures as well as to decidability properties for the respective class) which immediately lead to decision procedures for the considered problems; however, these procedures are inherently inefficient, even for classes such as the class of context-free languages (for short, CFL). In this context, generalizing the work of [4], we approach (in subsequent sections of this paper) each of the above problems for $\mathcal{C}$ being the class of CFL and, resp., the class of CS languages. While all the problems are undecidable for CS languages, we present efficient algorithms for the case of CFL. In particular, the results obtained for CFL are similar to the corresponding results obtained for REG (i.e., if a problem was solvable in polynomial or FPT-time for REG, we obtain an algorithm from the same class for CFL). In that regard, it seemed natural to search for a class of languages which does not exhibit this behaviour, while retaining the decidability of (at least some of) these problems. To this end, we identify the class of languages accepted by deterministic finite automata with translucent letters (a class of automata which does not process the input in a sequential fashion) and show (in the final section of this paper) a series of initial promising results related to them.

2 Preliminaries

Let $\mathbb{N} = \{1, 2, \ldots\}$ denote the natural numbers and set $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ as well as $[n] = \{1, \ldots, n\}$ and $[i, n] = \{i, i+1, \ldots, n\}$ for all $i, n \in \mathbb{N}_0$ with $i \leq n$.

An *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$ is a finite set of symbols, called *letters*. A *word* or *string* w is a finite concatenation of letters from a given alphabet with the number of these letters giving its *length* $|w|$. The word with no letters is the *empty word* ε of length 0. The set of all finite words over the alphabet Σ, denoted by Σ^*, is the free monoid generated by Σ with concatenation as operation. A subset $L \in \Sigma^*$ is called a *language*. Let Σ^n denote all words in Σ^* exactly of length $n \in \mathbb{N}_0$.

For $1 \leq i \leq j \leq |w|$ define the i^{th} letter of w by $w[i]$ and the *factor* of w starting at position i and ending at position j as $w[i, j] = w[i] \cdots w[j]$. If $i = 0$ the factor is also called a prefix, while if $j = |w|$ it is called a suffix of w. Let alph(w) denote the set of all letters of Σ ocurring in w. A word $u \in \Sigma^*$ is called *subsequence* of w, denoted $u \leq w$, if there exist $w_1, \ldots, w_{n+1} \in \Sigma^*$ such that $w = w_1 u[1] w_2 u[2] \cdots w_n u[n] w_{n+1}$. A word $w \in \Sigma^*$ is called *k-universal* (w.r.t. Σ), for $k \in \mathbb{N}_0$, if every $v \in \Sigma^k$ is a subsequence of w. The *universality-index* $\iota(w)$ is the largest k such that w is k-universal.

▶ **Definition 6.** *A grammar over an alphabet Σ is a 4-tuple $G = (V, \Sigma, P, S)$ consisting of: a set $V = \{A, B, C, \dots\}$ of non-terminal symbols, a set $\Sigma = \{a, b, c, \dots\}$ of terminal symbols with $V \cap \Sigma = \emptyset$, a non-empty set $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ of productions and a start symbol S.*

We represent productions $(p, q) \in P$ by $p \to q$. In G, $u = xpz$ with $x, z \in (V \cup \Sigma)^*$ is directly derivable to $v = xqz$ if a production $(p, q) \in P$ exists; in this case, we write $u \Rightarrow_G v$; the subscript G is ommitted when there is no danger of confusion. More generally, for some $n \geq 0$, we say that u is derivable to v in n steps (denoted $w \Rightarrow_G^n v$) if there exist $w_0, w_1, \dots, w_n \in (V \cup \Sigma)^*$ with

$$u = w_0 \Rightarrow_G w_1 \wedge w_1 \Rightarrow_G w_2 \wedge \cdots \wedge w_{n-1} \Rightarrow_G w_n = v.$$

If u is derivable to v in $m \geq 0$ steps, for some $m \geq 0$, we write $u \Rightarrow_G^* v$; in other words $\Rightarrow_G^*$ is the reflexive and transitive closure of $\Rightarrow_G$. With $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$ we denote the language generated by G. We call a sequence $S \Rightarrow \cdots \Rightarrow w \in L(G)$ a derivation. The number of steps used in the derivation is the derivation length. In some cases it may be easier to view derivations in a grammar G as a derivation (parse) tree. These are rooted, ordered trees. The inner nodes of such trees are labeled with non-terminals and the leaf-nodes are labeled with symbols $X \in (V \cup \Sigma)$. An inner node A has, from left to right, the children $X_1, \dots, X_k$ for some integer $k \geq 1$, if and only if the grammar contains the production $A \to X_1 \dots X_k$. As such, if we concatenate, from left to right, the leafs of a derivation tree T with root A we get a word α (called in the following the border of T) such that $A \to^* \alpha$. The depth of a derivation tree is the length of the longest simple-path starting with the root and ending with a leaf.

▶ **Definition 7.** *A grammar $G = (V, \Sigma, P, S)$ with $P \subseteq V \times (V \cup \Sigma)^+$ is a* context-free *grammar (for short, CFG). A language L is context-free (for short, CFL) if and only if there is a CFG G with $L(G) = L$.*
A grammar $G = (N, \Sigma, P, S)$, where for all $(p, q) \in P$ we have $|p| \leq |q|$, is a context-sensitive *grammar (for short, CSG). A language L is context-sensitive (for short, CSL) if and only if there is a CSG G with $L(G) = L$.*

The definitions above tacitly assume that CFLs and CSLs do not contain the empty word ε. Indeed, for the problems considered here, we can make this assumption. Whether $\varepsilon \in L$ or not plays no role in deciding Problems 1, 3, and 5, while $\varepsilon \in L$ immediately leads to a negative answer for Problems 2 and 4. So, for simplicity, we only address languages that, by definition, *do not* contain the empty word (see also the discussions in [37, 29] about how the presence of ε in formal languages can be handled).

Please read and check the part above about the empty word.

Also, note that every unary CFL is regular [50], so when discussing our problems for the class of CFLs we assume that the input languages are over an alphabet with at least two letters.

▶ **Definition 8.** *A CFG G is in Chomsky normal form (CNF) if and only if*
- *$P \subseteq V \times (V^2 \cup \Sigma)$ and*
- *for all $A \in V$, there exists some $w_A \in \Sigma^*$ such that $A \Rightarrow^* w_A$ and*
- *for all $A \in V$, there exist some $w'_A, w''_A \in \Sigma^*$ such that $S \Rightarrow^* w'_A A w''_A$.*

(these last two properties essentially say that every non-terminal of G is useful*)*

When we discuss our problems in the case of CFLs, we assume out input is a CFG G in CNF. This does not change our results since, according to [37] and the references therein, we can transform any grammar G in polynomial time into a CFG G' in CNF where $|G'| \in \mathcal{O}(|G|^2)$ and $L(G) = L(G')$.

For basic notions on finite automata, we refer to [29]. A *non-deterministic finite automaton (for short, NFA)* A is a quintuple $A = (Q, \Sigma, S, \delta, F)$, where Q is a set of states, Σ is the input alphabet, δ is a mapping from $Q \times \Sigma$ to 2^Q, $S \in Q$ is the set of initial states, and $F \in Q$ is the set of final states.

If we have $|S| = 1$ and $|\delta(q, a)| = 1$ for all $q \in Q, a \in \Sigma$, then A is called *deterministic* (for short, DFA). A word is accepted whenever the reading tape is empty and we end up in a final state. A state $q \in Q$ is called accessible (resp., co-accessible) in A if there exists a path connecting q_0 to q (resp., q to a final state). A state is called a-deficient, for $a \in \Sigma$, if $\delta q, a$ is not defined. Define the language of A, i.e., the set of words *accepted* by A, by $L(A)$. Note that the class of languages accepted by NFAs is equal to the class of languages accepted by DFAs and it is equal to REG. For a word w accepted by a finite automaton, let π_w denote one accepting path labelled with w, i.e., such a path is unique for DFAs; in the case when there are multiple such paths, we simply choose one of them. Since we usually are interested in only one path for a word $w \in L(A)$, we refer to it as π_w.

▶ **Definition 9.** *For any language $L \subset \Sigma^*$ the* downward closure *$L{\downarrow}$ of L is defined as the language containing all subsequences of words of L, i.e., $L{\downarrow} = \{v \in \Sigma^* \mid \exists w \in L : v \leq w\}$.*

The following lemma is folklore.

▶ **Lemma 10.** *Given a word $w \in \Sigma^*$, with $|w| = n$ and $|\Sigma| = \sigma$, we can construct in time $\mathcal{O}(n\sigma)$ a minimal DFA, with $n + 1$ states, accepting the set of words which have w as a subsequence.*

Proof. Let $n = |w|$. We construct a DFA $A = (Q, \Sigma, q_0, \{f\}, \delta)$ accepting the set of words which have w as a subsequence. We start by defining the set of states $Q = \{0, 1, \ldots, n\}$, the initial state $q_0 = 0$, and the single final state $f = n$. Now, for the transition function, we set, for $i \in [n]$, $\delta(i - 1, a) = i$ if and only if $w[i] = a$; otherwise, we set $\delta(i, a) = i$, for $i \in Q$ and $a \in \Sigma$.

The correctness of the construction (i.e., the fact that A accepts exactly the set of words v with $w \leq v$) is immediate. Indeed, to go from state 0 to state n we need to read a word which contains, in order but not necesarilly on consecutive positions, the letters $w[1], w[2], \ldots, w[n]$. That is, we need to read a word which has w as a subsequence.

To see that this DFA is minimal, it is enough to observe that the words $w[1 : i]$, for $i \geq 0$, are not equivalent w.r.t. the Myhill-Nerode equivalence. So, any DFA accepting the set of words v with $w \leq v$ needs to have at least $n + 1$ states.

This concludes our proof. ◀

▶ **Definition 11.** *The arch factorization of a word $w \in \Sigma^*$ is given by $w = ar_1(w) \ldots ar_k(w)r(w)$ for $k \in \mathbb{N}$ with $\iota(ar_i(w)) = 1$ and $ar_i(w)[|ar_i(w)|] \notin \text{alph}(ar_i(w)[1, |ar_i(w)| - 1])$, for all $i \in [1, k]$. Furthermore, $\text{alph}(r(w)) \subsetneq \Sigma$ applies. The words $ar_i(w)$ are the arches and $r(w)$ the rest of w. The modus $m(w)$ of w is defined as $ar_1(w)[|ar_1(w)|] \ldots ar_k(w)[|ar_k(w)|]$.*

For more details about the arch factorization and the universality index see [10, 27].

Finally, we present the following relationship between the arch factorization of a word $w \in \Sigma^*$ and the length of a shortest absent subsequences to w.

▶ Remark 12. The modus $m(w)$ of a word $w \in \Sigma^*$ can be used to directly construct a shortest absent subsequence (SAS) for w. We use the symbols of $m(w)$ and append any symbol $\sigma \in \Sigma$ to $m(w)$, where σ does not appear in the rest of w. This results in the fact that $\iota(w) \leq \ell - 1$, where ℓ is the length of a SAS. For the converse, see [10]. This results in $\iota(w) = \ell - 1$.

We can show the following lemma.

▶ **Lemma 13.** *For $k > 0$ and an alphabet Σ with $|\Sigma| = \sigma$ we can construct in time $\mathcal{O}(2^\sigma k \text{ poly}(\sigma))$ a minimal DFA, with $(2^\sigma - 1)k + 1$ states, accepting the set of k-universal words over Σ.*

Proof. Assume, for simplicity, that $\Sigma = \{1, 2, \ldots, \sigma\}$.

We construct a DFA $A = (Q, \Sigma, q_0, \{f\}, \delta)$ accepting the set of k-universal words. We start by defining the set of states $Q = \{(i, S) \mid 0 \leq i \leq k - 1, S \subsetneq \Sigma\} \cup \{(k)\}$, the initial state $q_0 = (0, \emptyset)$, and the single final state $f = (k)$. The transition function is defined as follows:

- for $a \in \Sigma$, $\delta((k), a) = (k)$;
- for $0 \leq t < k$, $S \subset \Sigma$, and $a \in \Sigma$, if $a \notin S$ and $S \cup \{a\} \neq \Sigma$, then $\delta((t, S), a) = (t, S \cup \{a\})$;
- for $0 \leq t < k-1$, $S \subset \Sigma$, and $a \in \Sigma$, if $a \notin S$ and $S \cup \{a\} = \Sigma$, then $\delta((t, S), a) = (t+1, \emptyset)$;
- for $S \subset \Sigma$, and $a \in \Sigma$, if $a \notin S$ and $S \cup \{a\} = \Sigma$, then $\delta((k-1, S), a) = (k)$;
- for $0 \leq t < k$, $S \subset \Sigma$, and $a \in S$, then $\delta((t, S), a) = (t, S)$.

To show that A accepts the set of k-universal words, we make the following observations:

- The set of words which label paths starting with $(0, \emptyset)$ and ending with $(0, S)$, for some set $S \subsetneq \Sigma$, are exactly those words w, with $\text{alph}(w) = S$. This can be shown easily by induction on $|S|$.
- The set of words which label paths starting with $(0, \emptyset)$ and ending with $(1, \emptyset)$ are exactly the 1-universal words w, such that $r(w) = \varepsilon$. This follows immediately from the previous observation.
- For $1 \leq i \leq k-2$, the set of words which label paths starting with $(0, \emptyset)$ and ending with (i, S), for some set $S \subsetneq \Sigma$ with $S \neq \emptyset$, are exactly those words w, with $\iota(w) = i$ and $\text{alph}(r(w)) = S$, and the set of words which label paths starting with $(0, \emptyset)$ and ending with $(i+1, \emptyset)$ are exactly those words w, with $\iota(w) = i+1$ and $r(w) = \varepsilon$. This can be shown by induction. We first assume that the property holds for $j = i-1$ (and note that it holds for $j = 0$). Then we show, by induction on $|S|$, that the set of words which label paths starting with $(0, \emptyset)$ and ending with (i, S), for some set $S \subsetneq \Sigma$, are exactly those words w, with $\iota(w) = i$ and $\text{alph}(r(w)) = S$. Finally, it immediately follows that the set of words which label paths starting with $(0, \emptyset)$ and ending with $(i+1, \emptyset)$ are exactly those words w, with $\iota(w) = i+1$ and $r(w) = \varepsilon$.
- The set of words which label paths starting with $(0, \emptyset)$ and ending with $(k-1, S)$, for some set $S \subsetneq \Sigma$ with $S \neq \emptyset$, are exactly those words w, with $\iota(w) = k-1$ and $\text{alph}(r(w)) = S$, and the set of words which label paths starting with $(0, \emptyset)$ and ending with (k) are exactly those words w, with $\iota(w) \geq k$. Again, we can show by induction on $|S|$, that the set of words which label paths starting with $(0, \emptyset)$ and ending with $(k-1, S)$, for some set $S \subsetneq \Sigma$, are exactly those words w, with $\iota(w) = i$ and $\text{alph}(r(w)) = S$. Finally, to reach (k) directly from a state $(k-1, S)$ we need to have that $\Sigma \setminus S = \{a\}$ and we read a in state $(k-1, S)$; this means that the word we have read is k-universal. Once (k) is reached, it is never left no matter what we read, so the words that lead from $(0, \emptyset)$ to (k) are exactly the k-universal words (so with $\iota(w) \geq k$).

Further, we show that A is minimal. Consider $0 \leq p \leq \sigma$ and $0 \leq i \leq k-1$, and a strict subset $S = \{j_1, \ldots, j_p\}$ of Σ. Now, consider the words $w_{i,S} = (12 \cdots \sigma)^i (j_1 \cdots j_p)$. It is immediate that $w_{i,S}$ and $w_{j,S'}$ are not equivalent under the Myhill-Nerode equivalence (defined for the language of k-universal words over Σ) if $(i, S) \neq (j, S')$. Moreover, none of these words is equivalent to $(12 \cdots \sigma)^k$. So, the Myhill-Nerode equivalence has at least $(2^\sigma - 1)k + 1$ states. As A has exactly as many states, our statement follows. ◀

The computational model we use is the standard unit-cost RAM with logarithmic word size: for an input of size n, each memory word can hold $\log(n)$ bits. Arithmetic and bitwise operations with numbers in $[1, n]$ are, thus, assumed to take $\mathcal{O}(1)$ time. Numbers larger than n, with ℓ bits, are represented in $\mathcal{O}(\ell / \log n)$ memory words, and working with them takes time proportional to the number of memory words on which they are represented. In some of the problems, we assume that we are given a number k, binary encoded, and one finite automaton $\mathcal{A}$, specified as the set of states, set of input letters, set of transitions, and initial and final states. The size of the input is, as such, the size of the binary encoding of k, which is $\lceil \log_2 k \rceil$, plus the size S of the encoding of $\mathcal{A}$ (which is lower bounded by the number of edges in the graph associated to the automaton). So, one memory word can hold $\log(S + \log_2(k))$ bits. If we get some grammar G instead of an automaton, G is specified by the variables, input alphabet, productions and the start state.

For a more detailed general discussion on this model see, e. g., [16].

Some of our algorithms (e.g., for Problem 3) run in exponential time and use exponential space w.r.t. the size n of the input. Following the literature dealing with such exponential algorithms (see, e.g., [20]), we use the $\mathcal{O}^*$-notation. By definition, for functions f and g we write $f(n) = \mathcal{O}^*(g(n))$ if $f(n) = \mathcal{O}(g(n)n^{\mathcal{O}(1)})$. In other words, the $\mathcal{O}^*$-notation hides polynomial factors, just as the $\mathcal{O}$-notation hides constants. Using this notation, we can assume that our single-exponential time and single-exponential space algorithms run on a RAM model where arithmetic and bitwise operations with single exponential numbers (w.r.t. the size n of the input) as well as accessing the memory-words (given their address, like in an usual RAM) are assumed to take $\mathcal{O}^*(1)$ time. Working with such a computational model allows us to analyse the actual algorithms rather than the various intricacies of the computational model.

In particular, when expressing the complexity of some of our algorithms, we get functions which are exponential in σ (the size of the alphabet) but polynomial in the number n of states of the input NFA or in the value of the input number k (the parameter of the considered problems). For clarity of the exposure, although we use the $\mathcal{O}^*$-notation, we explicitly write the dependency on n, m, and k, and only hide the polynomial dependency on σ.

this section of General Results, only holds for CF and CS, right? Must we explicitly mention it?

3 General Results

We consider the problems introduced in Section 1, for the case when the language L is chosen from a class $\mathcal{C}$, and give a series of sufficient conditions for them to be decidable.

Consider a class $\mathcal{G}$ of grammars (resp., a class $\mathcal{A}$ of automata) generating (resp., accepting) the languages of the class $\mathcal{C}$. For simplicity, for the rest of this section, we assume that in all the problems we take as input a grammar G_L such that $L(G_L) = L$, but note that all the results hold for the case when we consider that the languages are given by an automaton from the class $\mathcal{A}$ accepting them.

Let $\mathcal{C}'$ be the class of languages $L \cap R$, where $L \in \mathcal{C}$ and $R \in$ REG. We use two hypotheses:

H1. Given a grammar G of the class $\mathcal{G}$ we can algorithmically construct a non-deterministic automaton A accepting the downward closure of $L(G)$.

H2. Given a grammar G of the class $\mathcal{G}$ and an NFA A, we can algorithmically decide whether the language $L(G) \cap L(A)$ is empty.

for the next paragraph check if the phrasing needs changing, in light of new formulations of the problems.

We show, in a first theorem, that, under H1, Problems 1, 3, and 5 are decidable. Intuitively, this follows since deciding all these problems for L is equivalent to deciding them for $L{\downarrow}$, the downward closure of L. As $L{\downarrow}$ is regular, the decidability of these problems follows immediately (e.g., using [4]).

▶ **Theorem 14.** *If H1 holds, then Problems 1, 3, and 5 are decidable.*

Proof. We start by observing that the following straightforward properties hold:

- for a word w, there exists $v \in L$ such that $w \leq v$ if and only if there exists $v' \in L{\downarrow}$ with $v' \leq w$.
- for $k > 0$, there exists $v \in L$ such that v is k-universal if and only if there exists $v' \in L{\downarrow}$ with v' k-universal.

We now focus on Problems 1, 3, and 5. For all problems, consider a grammar G generating the language L. According to H1, in all cases, we construct a non-deterministic automaton A accepting $L{\downarrow}$, the downward closure of L.

For Problem 1, it is sufficient to check if $L(A) = L{\downarrow}$ contains the word w, which is clearly decidable.

For Problem 3 we need to decide if L contains a k-universal word. By our observations, it is enough to check if L contains a k-universal word. This can be decided, for the automaton A, according to [4].

For Problem 5 we need to decide if L contains a k-universal word, for all $k \leq 0$. This is also decidable, for A, according to the results of [4].

This concludes our proof. ◀

In our second result, we show that, under H2, Problems 1, 2, 3, and 4 are decidable. Intuitively, this follows from the fact that the set of words which have (resp., do not have) a given word as a subsequence, as well as the set of k-universal words (resp., the set of words which are not k-universal) are all regular. Therefore, the respective problems can be reduced to checking the emptiness of the intersection of the given language L to a language in REG, for which we can construct a finite automaton accepting it (by Lemmas 10 and 13).

▶ **Theorem 15.** *If H2 holds, then Problems 1, 2, 3, and 4 are decidable.*

Proof. Consider each of the inputs of Problems 1, 2, 3, and 4 given as some grammar G, which generates the language L.

For Problem 1, by Lemma 10 we construct a DFA B accepting the regular language of words which have w as a subsequence. We now check if the intersection of L (given as the grammar G which generates it) and $L(B)$ is empty or not, which is decidable, under H2. If the intersection is empty, then the answer to the considered instance of Problem 1 is answered negatively; otherwise, it is answered positively.

For Problem 2, by Lemma 10 we construct again a DFA B accepting the regular language of words which have w as a subsequence. By making the final state of B non-final, and all the other states final, we obtain a DFA B' which accepts exactly all words which do not have w as a subsequence. We now check if the intersection of L (given as the grammar G which generates it) and $L(B')$ is empty or not, which is decidable, under H2. If the intersection is empty, then the answer to the considered instance of Problem 1 is answered positively; otherwise, it is answered negatively.

For Problem 3, by Lemma 13 we construct a DFA B accepting the regular language of k-universal words. We now check if the intersection of L (given as the grammar G which generates it) and $L(B)$ is empty or not, which is decidable, under H2. If the intersection is empty, then the answer to the considered instance of Problem 3 is answered negatively; otherwise, it is answered positively.

For Problem 4, by Lemma 13 we construct again a DFA B accepting the regular language of k-universal words. By making the final state of B non-final, and all the other states final, we obtain a DFA B' which accepts exactly all words which are not k-universal. We now check if the intersection of L (given as the grammar G which generates it) and $L(B')$ is empty or not, which is decidable, under H2. If the intersection is empty, then the answer to the considered instance of Problem 4 is answered positively; otherwise, it is answered negatively.

This concludes the proof of this theorem. ◀

It is worth noting that, even for classes which fulfill both hypotheses above (such as the CFL languages), there are several reasons why the algorithms resulting from the above theorems are not efficient. On the one hand, constructing an automaton which accepts the downward closure of a language (so what we assume to be possible under H1) cannot always be done efficiently. For instance, in the case of CFLs, this may take inherently exponential time w.r.t. the size of the input grammar [**?**]; in this paper, we present more efficient algorithms for Problems 1, 3, and 5 in the case of CFLs, which do not rely on Theorem 14. On the other hand, the results of Theorem 15 rely, at least partly, on the construction of a DFA accepting all k-universal words, which takes exponential time in the worst case, as it may have an exponential number of states (both w.r.t. the size of the input alphabet and w.r.t. the binary representation of the number k, which is given as input for some of these problems). We also give more efficient algorithms for Problems 1, 2, 3, and 4 in the case of CFLs, using different approaches.

Interestingly, the class of CSL does not fulfil any of the above hypotheses. In fact, as our last general result, we show that all five problems are undecidable for CSL.

▶ **Theorem 16.** *Problems 1, 2, 3, 4, 5 are undecidable for the class of CSL, specified by a CSG generating them.*

Proof. To obtain the undecidability of all the problems, we show reductions from the emptiness problem for Context Sensitive Languages. Assume that we have a CSL L, specified by a grammar G, as the input for the emptiness problem for CSL. Assume L is over the alphabet $\Sigma = \{1, \ldots, \sigma\}$, and that the CSG G, has the starting symbol S. Let 0 be a fresh letter (i.e., $0 \notin \Sigma$).

To show the undecidability of Problems 1 and 2, we construct a new grammar G' which has all the non-terminals, terminals, and productions of G and, additionally, G' has a new starting symbol S' and the productions $S' \to \sigma S$ and $S' \to 0$.

Now, on the one hand, deciding whether $L(G)$ is empty is equivalent to deciding whether there exists a word $w \in L(G')$ which contains σ as a subsequence. As the emptyness problem is undecidable for CSL (given as grammars), it follows that Problem 1 is also undecidable for this class of languages.

On the other hand, deciding whether $L(G)$ is empty is equivalent to deciding whether all words of $L(G')$ contain 0 as a subsequence (this is true if and only if the production $S' \to \sigma S$ cannot be the first production in the derivation of any terminal-word). As the emptyness problem is undecidable for CSL (given as grammars), it follows that Problem 2 is also undecidable for this class of languages.

To show the undecidability of Problem 3, we construct a new grammar G' which has all the non-terminals, terminals, and productions of G and, additionally, G' has a new starting symbol S' and the production $S' \to 12 \cdots \sigma S$. Clearly, $L(G')$ contains a 1-universal word (over Σ) if and only if $L(G) \neq \emptyset$. Thus, it follows that Problem 3 is also undecidable for this class of languages.

To show the undecidability of Problem 4, we construct a new grammar G' which has all the non-terminals, terminals, and productions of G and, additionally, G' has a new starting symbol S' and the productions $S' \to 012 \cdots \sigma$ and $S' \to S$. Clearly, all the words of $L(G')$ are 1-universal (over $\Sigma \cup \{0\}$) if and only if $L(G) = \emptyset$ (as any word which would be derived in G' starting with the production $S' \to S$ would not contain 0). Hence, Problem 4 is also undecidable for CSL.

To show the undecidability of Problem 5, we construct a new grammar G' which has all the non-terminals, terminals, and productions of G and, additionally, G' has a new starting symbol S' and a fresh non-terminal R and the productions $S' \to 012 \cdots \sigma$, $S' \to RS$, $R \to 01 \cdots \sigma R$, and $R \to 01 \cdots \sigma$. Clearly, $L(G')$ contains m-universal words (over $\Sigma \cup \{0\}$) for all $m \geq 1$ if and only if $L(G) \neq \emptyset$ (as we can use R to pump arches in the words of $L(G')$ if and only if there exists at least one derivation where S can be derived to a terminal word). Accordingly, Problem 5 is also undecidable for CSL. ◀

Given that all the problems become undecidable for $\mathcal{C} = \text{CSL}$, we now focus our investigation on classes of languages strictly contained in the class of CSLs.

4 Problems 1 and 2

For the rest of this section, assume that $|w| = n$ and $|\Sigma| = \sigma$. Let us begin by noting that Problems 1 and 2 can be solved in polynomial time for the class REG following the approach of Theorem 15. Indeed, in this case, we assume that L is specified by the NFA A, with s states, with $L(A) = L$, and then we either have to check the emptiness of the intersection of $L = L(A)$ with the language accepted by the DFA constructed in Lemma 10, or, resp., with the complement of this language; both these tasks clearly take polynomial time.

We now consider the problems for the class of CFLs. We first recall the following folklore lemma.

▶ **Lemma 17.** *Let $G = (V, \Sigma, P, S)$ be a CFG in CNF and let $A = (Q, \Sigma, q_0, F, \delta)$ be a DFA. Then we can construct in polynomial time a CFG G_A such that $L(G') = L(G) \cap L(A)$.*

Proof. We first define the CFG $G' = (V', \Sigma, P', S')$ as follows:

- $V' = \{S'\} \cup \{(q, A, q') \mid A \in V, q, q' \in Q\}$;
- $P' = \{S' \to (q_0, S, f) \mid f \in F\} \cup \{(q, A, q') \to (q, B, q'')(q'', C, q') \mid A \to BC \in P, q, q', q'' \in Q\} \cup \{(q, A, q') \to a \mid A \to a \in P, q, q' \in Q, q' \in \delta(q, a)\}$.

It can now be easily shown by induction on the length m of the derivation that $S' \Rightarrow^m_{G'} w$ if and only if $w \in L(G) \cap L(A)$. As described in Section 2, G' can then be further transformed into a grammar G_A in CNF with $L(G_A) = L(G')$. ◀

We can now state the main result of this section.

▶ **Theorem 18.** *Problems 1 and 2 are decidable in polynomial time for CFL.*

Proof. We assume that L is specified by a CFG in CNF G. Let A be the DFA constructed in Lemma 10, which accepts all words having w as a subsequence, and let B be the DFA obtained from A by making all its non-final states final, and its only final state non-final (i.e., the complement DFA for A, accepting $\Sigma^* \setminus L$). The construction of A and B takes polynomial time.

By Lemma 17, in both cases we can construct CFG G_A and G_B, resp., accepting $L \cap L(A)$, resp. $L \cap L(B)$.

Further, we follow the approach of Theorem 15. For Problem 1, we check if the intersection of L and $L(A)$ is empty or not. This can be done in polynomial time by checking if G_A generates any word (see, e.g., [29]). If the intersection is empty, then the answer to the considered instance of Problem 1 is answered negatively; otherwise, it is answered positively. For Problem 2, we check if the intersection of L and $L(B)$ is empty. Again, this can be done in polynomial time by checking if G_B generates any word. If the intersection is empty, then the answer to the considered instance of Problem 2 is answered positively; otherwise, it is answered negatively. ◀

5 Problems 3 and 5

Let us begin by noting that in [4] it was shown that for a given NFA A with s states (with input alphabet Σ, where $|\Sigma| = \sigma$) and an integer $k \geq 0$, we can decide whether $L(A)$ contains a k-universal word (i.e., Problem 3 for the class REG) in time $\mathcal{O}(poly(n, \sigma)2^\sigma)$; in other words, Problem 3 is fixed parameter tractable (FPT) w.r.t. the parameter σ. A polynomial time algorithm was given for Problem 5, relying on the observation that, given NFA A, the language $L(A)$ contains words with arbitrarily large universality if and only if A contains a state q, which is reachable from the initial state and from which one can reach a final state, and a cycle which contains this state, whose label is 1-universal. Coming back to Problem 3 for REG, the same paper shows that it is actually NP-complete. This is proved by a reduction from the Hamiltonian Path problem (HPP, for short), in which a graph with n vertices, the input of HPP, is mapped to an input of Problem 3 consisting in an automaton with $\mathcal{O}(n^2)$ states over an alphabet of size n. This reduction also implies that, assuming that the Exponential Time Hypothesis (ETH, for short) holds, there is no $2^{o(\sigma)}\text{poly}(s, \sigma)$-time algorithm solving Problem 3 (as this would imply the existence of an $2^{o(n)}$-time algorithm solving HPP); see [42] for more details related to the Exponential Time Hypothesis and HPP.

Further, we consider Problems 3 and 5 for the class CFL, and we assume that, in both cases, we are given a CFL L by a CFG G in CNF, with n non-terminals.

To transfer the lower bound derived for Problem 3 in the case of REG (specified as NFAs) to the larger class of CFL (specified as CFGs in CNF), we recall the folklore result that a CFG $G = (V, \Sigma, P, S)$ in CNF can be constructed in polynomial time from an NFA (by constructing a

regular grammar from the NFA, and then putting the grammar in CNF, see [29]). So, the same reduction from [4] can be used to show that, assuming ETH holds, there is no $2^{o(\sigma)}\text{poly}(n,\sigma)$-time algorithm solving Problem 3. This reduction shows also that Problem 3 is NP-hard; whether this problem is in NP remains open.

We now focus on the design of a $2^{\mathcal{O}(\sigma)}\text{poly}(n,\sigma)$-time algorithm solving Problem 3 (which would also show that this problem is FPT) and that Problem 5 can be solved in polynomial time. So, from now on, we assume that we are given a CFG $G = (V, \Sigma, P, S)$ in CNF and an integer $k \geq 1$ (in binary representation). Let us observe that Problem 5 requires deciding whether $\iota_\exists(L)$ is finite, and, if yes, Problem 3 requires checking whether $\iota_\exists(L) \geq k$.

from the above, it seems that $\iota_\exists$ and $\iota_\forall$ need to be defined. Am I correct? If so, would $\iota_\exists(L)$ be the largest ι of a word in L, while $\iota_\forall(L)$ the minimum one?

We start with a series of combinatorial observations. We say that $A \in V$ generates a *1-universal cycle* if and only if there exists a derivation $A \Rightarrow^* w_1 A w_2$ with $w_1, w_2 \in \Sigma^*$ and $\iota(w_1) \geq 1$ or $\iota(w_2) \geq 1$. We can show the following result.

▶ **Lemma 19.** *Let $G = (V, \Sigma, P, S)$ be a CFG in CNF and $L = L(G)$, the language generated by G. Then $\iota_\exists(L)$ is infinite if and only if there exists a non-terminal $X \in V$ such that X has a 1-universal cycle.*

Proof. Assume we have a non-terminal $A \in V$ which generates a *1-universal cycle*. This means that there exists a derivation $A \Rightarrow^* w_1 A w_2$ with $w_1, w_2 \in \Sigma^*$ and $\iota(w_1) \geq 1$ or $\iota(w_2) \geq 1$. As G is in CNF, we have that there exist $w'_A, w''_A \in \Sigma^*$ and the derivation $S \Rightarrow^* w'_A A w''_A$, and, also, that there exists $w_A \in \Sigma^*$ such that $A \Rightarrow^* w_A$. We immediately get that, for all $n \geq 1$, the following derivation is valid: $S \Rightarrow^* w'_A A w''_A \Rightarrow^* w'_A w_1 A w_2 w''_A \Rightarrow^* w'_A (w_1)^2 A (w_2)^2 w''_A \Rightarrow^* w'_A (w_1)^n A (w_2)^n w''_A \Rightarrow^* w'_A (w_1)^n w_A (w_2)^n w''_A = w$. As $\iota(w_1) \geq 1$ or $\iota(w_2) \geq 1$, it follows that $\iota(w) \geq n$. So, $\iota_\exists(L)$ is infinite.

We now show the converse implication. We show by induction on the number of non-terminals in the grammar G that if $\iota_\exists(L)$ is infinite, then G has at least one non-terminal $X \in V$ such that X has a 1-universal cycle.

The result is immediate if G has a single non-terminal, i.e., the start symbol S. We now assume that our statement holds for CFLs generated by grammars with at most m non-terminals, and assume that L is a CFL generated by a CFG G with $m+1$ non-terminals. We want to show that G has at least one non-terminal $X \in V$ such that X has a 1-universal cycle. We can assume, w.l.o.g., that S does not have a 1-universal cycle (otherwise, the result already holds).

Now, consider for each $A \in V$ the CFG in CNF $G_A = (V \setminus \{S\}, \Sigma, A, P')$, where P' is obtained from P by removing all productions involving S. It is immediate that if there exists some $A \in V$ such that $\iota_\exists(L(G_A))$ is infinite, then, by induction, G_A contains a non-terminal $X \in V$ such that X has a 1-universal cycle. As G_A is obtained from G by removing some productions and one non-terminal, it is clear that X would also have a 1-universal cycle in G, so our statement holds. Let us now assume, for the sake of a contradiction, that for each $A \in V$ there exists an integer $N_A \geq 1$ such that $\iota_\exists(L(G_A)) \leq N_A$. Take $N = 1 + \max\{N_A \mid A \in V\}$. As $\iota_\exists(L)$ is infinite, then there exists a word $w \in L(G)$ with $\iota(w) = 2N + 3$.

Since $w \in L(G)$, $S \Rightarrow^* w$ holds. Let T_S be the parse tree with root S and p the longest simple-path of T_S starting in S and having the end-node S (in the case when there are more such paths, we simply choose one of them). We denote by T_S^p the sub-tree of T_S rooted in the end-node of p. If w' is the word obtained by reading the leaves of T_S^p left-to-right, then we have the following derivation corresponding to T_S: $S \Rightarrow v_S S v'_S \Rightarrow^* v_S w' v'_S = w$, where $v_S v'_S \in \Sigma^*$. Since, by our assumption, S does not have a 1-universal cycle, we get that $\iota(v_S) = 0$, $\iota(v'_S) = 0$, and that $\iota(w') \geq 2N + 1$.

introduce parse tree and simple-path?

Further, we consider T_S^p, and note that no other node of this tree, except the root, is labelled with S. Assume that the first step in the derivation $S \Rightarrow^* w'$ is $S \Rightarrow AB$, for some non-terminals $A, B \in V$

and production $S \to AB$, and that the children of the root S in the tree T_S^p are the sub-trees T_A and T_B. Let w_A be the border of T_A and w_B be the border of T_B. Clearly, it follows that at least one of the words w_A and w_B is N-universal. We can assume, w.l.o.g., that $\iota(w_A) \geq N$. But $w_A \in L(G_A)$ and $\iota_\exists(L(G_A)) < N$ (by the definition of N). This is a contradiction with our assumption that $\iota(L(G_X))$ is finite for all $X \in V$. So, there exists $X \in V$ for which $\iota_\exists(L(G_X))$ is infinite and, as we have seen, this means that our statement holds. ◀

So, according to Lemma 19, if the CFG G, which is the input of our problem, contains at least one non-terminal $X \in V$ which has a 1-universal cycle, we can answer positively the instances of Problems 3 and 5 defined by G and, in the case of Problem 3, by some integer $k \geq 1$. We show that one can decide in polynomial time whether such a non-terminal exists in a grammar. However, if G does not contain any non-terminal which has a 1-universal cycle, while the instance of Problem 5 can be already answered negatively, it is unclear what is the answer to Problem 3. To address this case, we try to find a way to efficiently construct a word of maximal universality index, and, for that, we first need another combinatorial result.

▶ **Lemma 20.** *Let $G = (V, \Sigma, P, S)$ be a CFG in CNF, with $|V| = n$, $|\Sigma| = \sigma$, and $L = L(G)$. Furthermore, assume $\iota_\exists(L)$ is finite. There exists a word w of L with $\iota(w) = \iota_\exists(L)$ such that the derivation tree of w has depth at most $4n\sigma$.*

Proof. Let $w_0 \in L$ be a word such that $\iota(w_0) = \iota_\exists(L)$, and let T_0 be its derivation tree. Assume that T_0 has depth greater than $4n\sigma$. Then there exists a simple-path p in T_0 from the root to a leaf of length at least $4n\sigma + 1$ (i.e., contains $4n\sigma + 2$ nodes on it). By the pigeonhole-principle, there is one non-terminal $A \in V$ which occurs at least 4σ times on this path. Therefore, there exists the derivation $S \Rightarrow^* v_0 A v'_0 \Rightarrow^* v_0 v_1 A v'_1 v'_0 \Rightarrow^* \dots \Rightarrow^* v_0 v_1 \cdots v_{4\sigma-1} A v'_{4\sigma-1} \cdots v'_1 v'_0 \Rightarrow^* v_0 v_1 \cdots v_{4\sigma-1} w'_0 v'_{4\sigma-1} \cdots v'_1 v'_0 = w_0$, with $v_0, v'_0, \dots, v_{4\sigma-1}, v'_{4\sigma-1}, w'_0 \in \Sigma^*$.

added w'_0 also to terminals

Because $\iota_\exists(L)$ is finite, it follows from Lemma 19 that A has no 1-universal cycle, so $\iota(v_1 \cdots v_{4\sigma-1}) = 0$ and $\iota(v'_{4\sigma-1} \cdots v'_1) = 0$.

We now go with i from 1 to $4\sigma - 1$ and construct a set M_ℓ as follows. We maintain a set U, which is initialized with $\text{alph}(r(v_0))$; we also initialize $M_\ell = \emptyset$. Then, when considering i, if $\text{alph}(v_i) \not\subseteq U$, we let $U \leftarrow U \cup \text{alph}(v_i)$ and $M_\ell \leftarrow M_\ell \cup \{i\}$; before moving on and repeating this procedure for $i + 1$, if $U = \Sigma$, we set $U \leftarrow \emptyset$.

Let us note that, during this process, because $\iota(v_1 \cdots v_{4\sigma-1}) = 0$, we set $U \leftarrow \emptyset$ at most once. Also, since M_ℓ is updated only when $\text{alph}(v_i) \not\subseteq U$, it means that M_ℓ is updated at most $2\sigma - 2$ times. So $|M_\ell| \leq 2\sigma - 2$.

Similarly, we can construct a set M_r. We go with i from $4\sigma - 1$ downto 1 and maintain a set U, which is initialized with $\text{alph}(r(v_0 v_1 \cdots v_{4\sigma-1} w'_0))$; we also initialize $M_r = \emptyset$. Then, when considering i, if $\text{alph}(v'_i) \not\subseteq U$, we let $U \leftarrow U \cup \text{alph}(v'_i)$ and $M_r \leftarrow M_r \cup \{i\}$; before moving on and repeating this procedure for $i - 1$, if $U = \Sigma$, we set $U \leftarrow \emptyset$. As before, we get that M_r is updated at most $2\sigma - 2$ times, and $|M_r| \leq 2\sigma - 2$.

here I think it is $i - 1$ not $+$

It is worth noting that the indices stored in M_ℓ and M_r indicate the words v_i and v'_i, resp., which contain letters that are important when computing the arch-factorization of w_0. The indices not contained in these sets indicate words v_i or v'_i, resp., which are simply contained in an arch, and all the letters of these words already appeared in that arch before the start of v_i and v'_i, resp.

is not "meaningful" a better choice?

As $|M_\ell| + |M_r| \leq 4\sigma - 4$, we immediately get that there exists $i \in [1, 4\sigma]$ such that $i \notin M_\ell \cup M_r$. It is now immediate that the derivation $S \Rightarrow^* v_0 A v'_0 \Rightarrow^* v_0 v_1 A v'_1 v'_0 \Rightarrow^* \dots \Rightarrow^* v_0 v_1 \cdots v_{i-1} A v'_{i-1} \cdots v'_1 v'_0 \Rightarrow^* v_0 v_1 \cdots v_{i-1} v_{i+1} A v'_{i+1} v'_{i-1} \cdots v'_1 v'_0 \Rightarrow^* v_0 v_1 \cdots v_{i-1} v_{i+1} \cdots v_{4\sigma-1} w'_0 v'_{4\sigma-1} \cdots v'_{i+1} v'_{i-1} \cdots v'_1 v'_0$ w_1 produces a word w_1 such that $\iota(w_1) = \iota(w_0)$; let T_1 be the tree corresponding to this derivation. Clearly, the total length of the simple-paths connecting the root to leaves in the derivation tree T_1 is

strictly smaller than the total length of the simple-paths connecting the root to leaves in the derivation tree T_0.

If T_1 still has root-to-leaf simple-paths of length at least $4n\sigma$, we can repeat this process and obtain a tree where the total length of the simple-paths connecting the root to leaves is even smaller. This process is repeated as long as we obtain trees having at least one root-to-leaf simple-path of length at least $4n\sigma$. Clearly, this is a finite process, whose number of iterations is bounded by, e.g., the sum of the length of root-to-leaf simple-paths of T_0. When we obtain a tree T where all root-to-leaf simple-paths are of length at most $4n\sigma$, we stop and note that the border of this tree is a word w, with $\iota(w) = \iota_\exists(L)$. This concludes our proof. ◀

We now come to the algorithmic consequences of our combinatorial lemmas. Firstly, we show that Problem 5 can be decided in polynomial time.

▶ **Theorem 21.** *Problem 5 can be solved in $\mathcal{O}(XXX)$-time*

Proof. TO BE ADDED. ◀

Further, we show that Problem 3 is fixed parameter tractable w.r.t. the parameter σ. Recall that we had an ETH-conditional lower bound for the time complexity of algorithms solving this problem of $2^{o(\sigma)}\text{poly}(s, \sigma)$.

▶ **Theorem 22.** *Problem 3 can be solved in $\mathcal{O}(XXX)$-time.*

Proof. TO BE ADDED. ◀

We observe that the algorithm supporting the result of Theorem 38 uses exponential space. However, there is also a simple PSPACE-algorithm solving this problem. TO BE ADDED!

We use a dynamic programming approach for the algorithm that results in a polynomial runtime for a constant size alphabet. As input we get a CFG $G = (V, \Sigma, P, S)$ in CNF with $|\Sigma| = \sigma$ and $|V| = m$. Additionally we get an positive integer k. We construct a matrix

$$M[i, A, S_p^A, S_s^A] = \ell,$$

with $A \in V$, $S_p^A, S_s^A \subseteq 2^\Sigma$ and $\ell \in \mathbb{N}$. Here ℓ is the maximum number of arches, so that we obtain a word w from A in a maximum of i steps, so that w contains ℓ arches and a prefix x with $\text{alph}(x) = S_p^A$ and a suffix y with $\text{alph}(y) = S_s^A$. Because of Lemma 20 i is upper bounded by $4m\sigma$. Since we form the maximum at the end, we initialize all entries with $-\infty$. In the base case for $i = 0$ we consider direct productions $(A, a) \in P$ for $A \in V$ and $a \in \Sigma$. The following applies

$$M[i, A, \{a\}, \emptyset] = 0 \text{ and } M[i, A, \emptyset, \{a\}] = 0. \tag{1}$$

This means that we consider the case where the symbol a is in the prefix and the case where it is in the suffix. The other set is empty, as we only have one letter. Accordingly, the entry itself is also 0, because a symbol alone does not form an arch.

In the induction step, we consider $(A, BC) \in P$ with $A, B, C \in P$ and the entry $M[i, A, S_p^A, S_s^A]$. Let u be a word derivable from B and v a word derivable from C. If u has a suffix x with $\text{alph}(x) = S_s^B$ and v has a prefix y with $\text{alph}(y) = S_p^C$ so that $S_s^B \cup S_p^C = \Sigma$, we concatenate u and v and thus obtain a new arch. Accordingly, we add $M[i-1, B, S_p^A, S]$ and $M[i-1, C, S', S_s^A]$ and add one for the new arch. We then take the maximum of all these combinations. We get

$$c = \max_{S_s^B \cup S_p^C = \Sigma} (M[i-1, B, S_p^A, S_s^B] + M[i-1, C, S_p^C, S_s^A] + 1). \tag{2}$$

If A has t productions we get $c_1, \dots, c_t$. To determine $M[i, A, S_p^A, S_s^A]$ we take the maximum over $c_1, \dots, c_t$. To determine the result at the end, we take the maximum over the entries of S, as we only consider words w that lie in L. This means that there is a derivation of S, resulting in w. We compare the maximum with k. This gives us the following equation.

$$k \leq \max(M[4m\sigma, S, S_p^S, S_s^S]) \tag{3}$$

If the equation is fulfilled, a word $w \in L$ exists so that $\iota(w) \geq k$ applies. If the equation is not fulfilled, no such word exists. We therefore obtain the term.

Check the notation in following theorem

▶ **Theorem 23.** *Given a CFG $G = (V, \Sigma, P, S)$ with $L = L(G)$, $|\Sigma| = \sigma$, $|V| = m$ and an integer k. We can decide whether a $w \in L$ exists with $\iota(w) \geq k$ in $\mathcal{O}(2^{2\sigma} \cdot 4m^2\sigma)$ time.*

Proof. Let $i \in [1, 4\sigma m]$ be fixed. For a fixed i we get a runtime of $\mathcal{O}(2^{2\sigma})$, since we go over all sets S_p and S_s to determine an entry $M[i, A, S_p, S_s]$. These are all subsets of Σ. This means that we have $2^{2\sigma}$ subsets over which we take the maximum. Since $i \in [1, 4\sigma m]$ applies, we perform this step $4\sigma m$ times for each $A \in V$, which gives us $4\sigma m^2$ iterations. This results in a total of $\mathcal{O}(2^{2\sigma} \cdot 4m^2\sigma)$. ◀

If the alphabet size is constant, we obtain a polynomial runtime accordingly.

6 Problem 4

6.1 Problem 4 for CFL

Basically, our algorithm is inspired by the CYK algorithm. The CYK algorithm is used to decide for a word w and a grammar G in CNF whether $w \in L(G)$ applies. For a more detailed description see [30]. We first introduce the necessary terms and concepts for the algorithm. Then we give a description of the algorithm itself. We enforce that $|\Sigma| > 1$ applies because the language $L(G)$ is regular for a unary alphabet.

6.1.1 Basic Notation and Concepts

Firstly, we introduce the concept of a pseudo arch. A pseudo arch is similar to an arch, but does not require that it is one universal. Intuitively, we use this to overestimate the number of actual arches in our algorithm. Since we only have pseudo arches, we also need a pseudo modus.

▶ **Definition 24.** *A word $v \in \Sigma^*$ is a pseudo arch if $v[|v|] \notin \text{alph}(v[1, |v|-1])$ holds. A pseudo arch factorization of a word $w \in \Sigma^*$ is defined as $w = v_1 \dots v_n$, where v_i is a pseudo arch for $i \in [1, n]$. The pseudo modus for a fixed pseudo arch factorization is defined as the concatenation of the unique last letters of each pseudo arch, i.e., $\tilde{m}(w) = v_1[|v|] \dots v_n[|v|]$.*

Next, we consider how the length of a shortest absent subsequence (SAS) of a word $w = uv$ with $w, u, v \in \Sigma^*$ results from the lengths ℓ_u, ℓ_v of the SAS of u and v. For our application, we assume that the first arch of v forms a new arch with the rest of u.

▶ **Example 25.** Let $\Sigma = \{a, b, c\}$, $u = abbcabacab$ and $v = abcaaabbbca$. Where $u = (abbc) \cdot (abac) \cdot ab$ is the arch factorization of u and $v = (abc) \cdot (aaabbbc) \cdot a$ is the arch factorization of u. Consider $w = uv$. We concatenate ab and (abc), resulting in the arch $(ababc)$.

In this case intuitively, the last letter of a SAS of u is added to the concatenation of u and v, as this symbol is located in the first arch of v. The length ℓ_v remains unchanged, which means that the length of an SAS ℓ_w of w is $\ell_w = \ell_u - 1 + \ell_v$.

▶ **Lemma 26.** *Let $u, v \in \Sigma^*$ with ℓ_u we denote the length of an SAS of u. Similarly, ℓ_v is the length of SAS of v. For $w = uv$, this results in $\ell_w = \ell_u + \ell_v - 1$, where ℓ_w is the length of an SAS of w. In addition, the concatenation of $r(u)$ and the first arch of v forms a new arch.*

Proof. Let $u, v \in \Sigma$. According to Definition 11 the following applies $u = ar_1(u) \dots ar_n(u)r(u)$ and $v = ar_1(v) \dots ar_m(v)r(v)$ with $n, m \in \mathbb{N}$. Additionally we get $m(u) = ar_1(u)[|ar_1(u)|] \dots ar_k(u)[|ar_k(u)|]$ and $m(v) = ar_1(v)[|ar_1(v)|] \dots ar_k(v)[|ar_k(v)|]$, where $m(u)$ denotes the modus of u and $m(v)$ the modus of v. According to Remark 12, a shortest absent subsequence (SAS) of u is constructed by $m(u) \cdot b$ with $b \in \Sigma \setminus \text{alph}(r(u))$. We proceed identically for v. Thus $\ell_u = |m(u)| + 1$, where ℓ_u denotes the length of an SAS of u. This applies analogously for ℓ_v. For $w = uv$ we obtain $w = ar_1(u) \dots ar_n(u)r(u)ar_1(v) \dots ar_m(v)r(v)$. Since we assume that $r(u)$ and $ar_1(v)$ form a new arch $ar'_1(v)$, we get $w = ar_1(u) \dots ar_n(u)ar'_1(v) \dots ar_m(v)r(v)$. From this follows for $m(w)$ that $m(w) = ar_1(u)[|ar_1(u)|] \dots ar_k(u)[|ar_k(u)|]ar'_1(v)[|ar'_1(v)|] \dots ar_k(v)[|ar_k(v)|] = m(u) \cdot m(v)$. Therefore, we construct a SAS for w as $m(u) \cdot m(v) \cdot b$ with $b \in \Sigma \setminus \text{alph}(r(v))$. So $\ell_w = |m(u)| + |m(v)| + 1 = |m(u)| + 1 + |m(v)| + 1 - 1 = \ell_u + \ell_v - 1$. ◀

With pseudo arches, we construct an absent subsequence from the pseudo modus. However, this does not have to be an SAS, as we potentially overestimate the number of actual arches with the pseudo arches. Accordingly, we refer to absent subseqeunces that we generate with the pseudo modus as pseudo SAS.

In our algorithm, we consider the lengths of such pseudo SAS and we store these lengths in a matrix M. In addition, we remember the initial letter and the final letter, since when combining two pseudo SAS s_1, s_2 we have to make sure that the initial letter of s_2 is identical to the final letter of s_1. This follows from the fact that we are only considering pseudo arches, so it is not necessary for the last letter of s_1 to appear in the first pseudo arch connected to s_2.

Let $G = (V, \Sigma, P, S)$ be a grammar and we consider $(A, BC) \in P$ with $A, B, C \in V$. We combine words derivable from B with words derivable from C. Since we consider pseudo arches and therefore also pseudo SAS for these words, we only combine words for which we can combine the pseudo SAS accordingly. This means that the final letter of the pseudo SAS of a word derivable from B corresponds to the initial letter of an pseudo SAS of a word derivable from C.
Therefore, we define a special type of prefix and suffix for a word $w \in \Sigma^*$. For which we require that certain symbols from Σ appear or are absent.

▶ **Definition 27.** *Given a word $w \in \Sigma^*$, we call u an a-prefix of w for $a \in \Sigma$ if u is a prefix of w, $u[|u|] = a$ and $a \notin \text{alph}(u[1, |u| - 1])$. The word u is a b-suffix for $b \in \Sigma$ if u is a suffix of w and $b \notin \text{alph}(u)$ applies.*

Thus, the prefix defines the first letter of a pseudo SAS and the suffix defines the last letter of a pseudo SAS. There can be pseudo arches between the prefix and the suffix. If we combine a suitable suffix with a suitable prefix, we obtain a new pseudo arch.

▶ Remark 28. Let $u \in \Sigma^*$ be a word with an a-prefix and a b-suffix and let $v \in \Sigma^*$ be a word with a b-prefix and a c-suffix for $a, b, c \in \Sigma$. If we concatenate u with v, we combine a b-suffix with a b-prefix. This combination creates a pseudo arch, as there is no b in the b-suffix and the b-prefix contains a single b in the last position.

The fact that a new pseudo arch is created during the concatenation is similar to the case from Lemma 26, where we fill up the first arch of the second word with the symbols from the rest of the first word. This results in the following for the length of a pseudo SAS in the concatenation.

▶ Remark 29. Let $u \in \Sigma^*$ be a word with an a-prefix and a b-suffix and let $v \in \Sigma^*$ be a word with a b-prefix and a c-suffix for $a, b, c \in \Sigma$. Let ℓ_u be the length of a pseudo SAS of u, ℓ_v the length of a

pseudo SAS of v and ℓ_w the length of a pseudo SAS of w with $w = uv$. According to Remark 28 w has a new pseudo arch. It follows that $\ell_w = \ell_u + \ell_v$ because, unlike Lemma 26, we get a new pseudo arch and not just fill up the first arch of v with the rest of u.

Since we know the initial letters and final letters from the entries of M, we use Remark 29 in the induction step to combine the entries of M. Finally, we use Remark 12 to determine $\iota_\forall$ at the end, since according to Remark 12 $\iota(w) = \ell_w - 1$, where ℓ_w is the length of an SAS of $w \in \Sigma^*$.

6.1.2 Algorithm for Universal Universality

We first describe how exactly the matrix M is structured and then present the procedure to fill the entries of M. We use a dynamic programming approach to build M. Therefore we construct a matrix

$$M[i, A, a, b] = k \tag{4}$$

with $i, k \in \mathbb{N}$, $A \in V$ and $a, b \in \Sigma$. Where k is the length of the shortest string that begins with an a and ends with an b and is also an absent subsequence of at least one word w that can be derived from A with a maximum derivation height of i. This means that w has an a-prefix and a b-suffix. Additionally a can be ε to denote an empty prefix. For example the word *aababb* can be seen as an word with an a-prefix and a c-suffix with $c \in \Sigma \setminus \{a, b\}$ but on the other hand we can push all letters in the suffix such that we get an empty prefix and still have a c-suffix. We use them empty prefix to pump up pseudo arches such that they can become real arches at some point.

Since we are looking for a word with a minimum universality index for $\iota_\forall(L)$ among all words in L, it is sufficient to only consider derivations for which it is true that for all $A \in V$ the variable A does not occur in a subtree that has A as a root. This results in the following upper limit for i.

▷ Claim 30. For the algorithm, it is sufficient to consider $i \in [1, |V|]$.

Proof. Let $G = (V, \Sigma, P, S)$ be a CFG and L, the language generated by G. Furthermore, let $w \in L$ be a word so that for the derivation of w it holds that an $A \in V$ exists, so that the parse tree T_A with root A again contains an A. This A spans a tree T'_A. The tree T_A generates a word v and T'_A generates a word v'. It holds that $v = xv'y$ with $x, y \in \Sigma^*$. Thus $\iota(v) \geq \iota(v')$ follows. Since we are looking for the minimum universality index for $\iota_\forall(L)$, we replace T_A with T'_A which gives us a word w' instead of w with $\iota(w') \leq \iota(w)$, since $w' = uv'z$ and $w = uvz$ with $u, z \in \Sigma^*$ and $\iota(v') \leq \iota(v)$. ◀

This means that $|V|$ is the upper limit for i. As we are looking for the minimum universality index for $\iota_\forall$, we initialize all entries of M with ∞. We first introduce the part without the empty prefix.

For the base case, we consider $i = 1$ and direct productions $(A, a) \in P$. The symbol a is a pseudo arch, since a occurs exactly once in the pseudo arch in the last position. Accordingly, we need any other symbol $b \in \Sigma$ to jump out of the pseudo arch and construct a pseudo SAS of length 2. It follows that

$$M[1, A, a, b] = 2 \tag{5}$$

holds, for $A \in V$ and $a, b \in \Sigma$. If there is no production $(A, a) \in P$, the entry $M[1, A, a, b]$ is not updated. If $(S, a) \in P$ exists, we cancel directly and return 0, since it follows from $|\Sigma| > 1$ that $\iota_\forall(L) = 0$.

In the induction step, we consider productions of the form $(A, BC) \in P$ with $A, B, C \in V$. In order to calculate the entry $M[i, A, a, b]$, there are three cases in which we combine words derived from B and C and thereby create a new pseudo arch.

▶ **Example 31.** We consider the production $(A, BC) \in P$. Here, $u = abbbaaab$ can be derived from B and $v = abababaaba$ can be derived from C. The word u has the b-prefix ab, since ab is a prefix of u and b occurs exactly once in the last position. Similarly, b is an a-suffix, because it is a suffix of u and does not contain the letter a. The word v has the a-prefix a and the b-suffix a.

This means that u has a suffix that matches a prefix from v. We therefore combine the a-suffix of u with the a-prefix of v. The concatenation creates the new pseudo arch ba.

In all cases we consider a word u with an a-prefix and an x-suffix that is derivable from B and a word v with an x-prefix and a b-suffix hat is derivable from C with $x \in \Sigma$. If we combine an x-suffix with an x-prefix, we get a new pseudo arch according to Remark 28, as the x-suffix is missing an x that is added by the x-prefix.

The difference in the cases is that we choose different derivation depths for B and C. In the first case, both variables have a derivation depth of at most $i-1$. Accordingly, we add the entries $M[i-1, B, a, x]$ and $M[i-1, C, x, b]$, since the concatenation of u and v creates a new pseudo arch and according to Remark 29 the lengths of the pseudo SAS add up in this case. Since we want to achieve the minimal universality index, we take the minimum over all x. It follows

$$m_1 := \min_{x \in \Sigma} M[i-1, B, a, x] + M[i-1, C, x, b]. \tag{6}$$

For cases two and three, we derive one variable directly and choose a derivation with depth of at most $i-1$ for the other variable. Accordingly, we add either $M[i-1, B, a, x]$ and $M[1, C, x, b]$ or $M[1, B, a, x]$ and $M[i-1, C, x, b]$. We add entries for the same reason as in the first case. Furthermore, we again take the minimum over all x. This results in

$$m_2 := \min_{x \in \Sigma} M[i-1, B, a, x] + M[1, C, x, b] \text{ and } m_3 := \min_{x \in \Sigma} M[1, B, a, x] + M[i-1, C, x, b]. \tag{7}$$

These updates never produce pseudo arches that contain more than two symbols. This is relatively easy to see for the case $i \leq 2$. For $i = 1$ we only consider direct derivations and thus words that consist of only one symbol. For $i = 2$, we combine these words, creating new pseudo arches that contain a maximum of two symbols.

▶ **Example 32.** Let $\Sigma = \{a, b, c\}$ and we consider the production $(A, BC) \in P$. Here $u = b$ is derivable from B and $v = a$ is derivable from C. We connect u and v by exploiting the fact that u has the a-suffix b and v has the a-prefix a. This creates the pseudo arch ba. Let us then consider the production $(D, AE) \in P$ with A mapping to $w_1 = ba$ and E mapping to $w_2 = bb$. Now w_1 has the b-suffix a and w_2 has the b-prefix b. Accordingly, we connect w_1 and w_2 and the new pseudo arch ab is created. This again has only two symbols.

Now we have to make sure that pseudo arches can become real arches at some point. Therefore we introduce the part of the algorithm related to the empty prefix. If we consider a production $(A, a) \in P$, a is a word with an a-prefix and any suffix, but it is also a word with an empty prefix and a b-suffix with $b \neq a$. So we move the a into the suffix. As a result, the length of a pseudo SAS is 1, as we have no symbol in the prefix and therefore need exactly one symbol to jump out of the word. Accordingly, for $i = 1$ we get the case

$$M[1, A, \varepsilon, b] = 1. \tag{8}$$

The entry $M[1, A, \varepsilon, b]$ exists for $A \in V$ and $b \in \Sigma$ if there is a production $(A, a) \in P$ with $a \neq b$. In the update step, we now use empty prefixes to enlarge a prefix or suffix. This means that we again consider productions $(A, BC) \in P$ with $A, B, C \in V$ and the entry $M[i, A, a, b]$. To calculate the entry $M[i, A, a, b]$, we consider words u, v, where u is derived from B and v is derived from C. We assume that either u or v has an empty prefix.

We first consider the case where u has an empty prefix and we use this to fill the prefix of v.

▶ **Example 33.** Let $\Sigma = \{a, b, c\}$ and we consider the production $(A, BC) \in P$. Here $u = b$ is derivable from B and $v = ac$ is derivable from C. Furthermore, we require that uv again has an a-prefix and a b-suffix. Thus we consider u to be a word with an empty prefix and an a-suffix. The word v has an a-prefix and a b-suffix. So that uv again has an a-prefix, we add the symbol from the a-suffix of u to the a-prefix of v. This does not create a new pseudo arch, but we add new symbols to the a-prefix.

Therefore we add $M[1, B, \varepsilon, a]$ and $M[i-1, C, a, b]$ and get

$$m_4 := M[1, B, \varepsilon, a] + M[i-1, C, a, b] - 1. \tag{9}$$

In the second case, v has an empty prefix and we use it to fill the suffix of u.

▶ **Example 34.** Let $\Sigma = \{a, b, c\}$ and we consider the production $(A, BC) \in P$. Here $u = b$ is derivable from B and $v = c$ is derivable from C. In this case, u has an a-suffix, but v has no a-prefix to complete the a-suffix of u. Nevertheless, we concatenate u and v by considering v such that v has an empty prefix and an a-suffix. In addition, u has an a-suffix. This means that we get $w = uv = bc$, where we have added the letter c to the a-suffix of u. In this way, we enlarge a suffix without creating a new pseudo arch.

We add $M[i-1, B, a, b]$ and $M[1, C, \varepsilon, b]$ and get

$$m_5 := M[i-1, B, a, b] + M[1, C, \varepsilon, b] - 1. \tag{10}$$

We also subtract one in each case, as no new pseudo arch is created under the assumption of an empty prefix and therefore Remark 29 does not apply.
The entry $M[i, A, a, b]$ results from $m_1, \dots, m_5$ and $M[i-1, A, a, b]$ as follows

$$M[i, A, a, b] := \min\{m_1, m_2, m_3, m_4, m_5, M[i-1, A, a, b]\}. \tag{11}$$

In the update, we also take into account the entry $M[i-1, A, a, b]$, as it is possible that we get a word with depth at most $i-1$, with less pseudo arches. Once all entries of M have been calculated, $\iota_\forall(L)$ results from

$$\iota_\forall(L) := \min_{a \in \Sigma \cup \{\emptyset\}, b \in \Sigma} M[|V|, S, a, b] - 1. \tag{12}$$

Since we only consider words w for which $w \in L$ applies, it is sufficient to take the minimum over the entries of S. Furthermore, it is sufficient to consider $i = |V|$ because we also consider the shorter derivatives with each update by Equation (11). Since an entry stores the length of a pseudo SAS, we subtract one according to Remark 12 to obtain $\iota_\forall(L)$.
For the runtime of the Algorithm the following holds.

▶ **Theorem 35.** *To calculate $\iota_\forall(L)$ for a CFL L given by a CFG $G = (V, \Sigma, P, S)$ in CNF, the algorithm requires $\mathcal{O}(|V|^2 \cdot |\Sigma|^3)$ time.*

Proof. Let $i \in [1, |V|]$ be fixed initially. For a fixed i, we obtain $\mathcal{O}(|V| \cdot |\Sigma|^3)$, since for i we iterate over all combinations of $A \in V$, $a \in \Sigma \cup \{\emptyset\}$ and $b \in \Sigma$. This means that we compute $\mathcal{O}(|V| \cdot |\Sigma|^2)$ entries. In addition, we need $\mathcal{O}(|\Sigma|)$ time per entry, because for m_1, m_2, m_3 we go over all $x \in \Sigma$ with which we connect an x-suffix with an x-prefix. This gives us a total of $\mathcal{O}(|V| \cdot |\Sigma|^3)$. Since $i \in [1, |V|]$ applies, we obtain a runtime of $\mathcal{O}(|V|^2 \cdot |\Sigma|^3)$ for the construction of M. In order to calculate $\iota_\forall(L)$ at the end using M, we need $\mathcal{O}(|\Sigma|^2)$ time, since after Equation (12) we have the minium over all entries of S, where $i = |V|$. This means that we consider $\mathcal{O}(|\Sigma|^2)$ entries, since we go over all $a, b \in \Sigma$. Thus we get $\mathcal{O}(|V|^2 \cdot |\Sigma|^3)$ for the calculation of M and $\mathcal{O}(|\Sigma|^2)$ to finally calculate $\iota_\forall(L)$, this results in a total runtime of $\mathcal{O}(|V|^2 \cdot |\Sigma|^3)$. ◀

Thus, we have an algorithm that calculates $\iota_\forall$ for a CFL L in polynomial time. Where L is given by a grammar in CNF. With this algorithm we can solve Problem 4 in polynomial time for CFL.

▶ **Corollary 36.** *Problem 4 is decidable in polynomial time for a given CFG G and some positive integer k.*

This is done by using the presented algorithm to compute $\iota_\forall(L)$ and then check if $\iota_\forall(L) \geq k$. If this the case all words are k-universal, otherwise there exists some word w with $\iota(w) < k$.

▶ **Theorem 37.** *Problem 5 can be solved in $O(\max(n^3, n^2\sigma))$-time, for a machine with n states.*

Proof. By Lemma 19, it is enough to check whether G contains a non-terminal $X \in V$ such that X has a 1-universal cycle. More precisely, we want to check if there exists a non-terminal X such that $X \Rightarrow^* wXw'$, where $\text{alph}(w) = \Sigma$ or $\text{alph}(w') = \Sigma$.

In the following, we show how to decide if there exists a non-terminal X such that $X \Rightarrow^* wXw'$, where $\text{alph}(w) = \Sigma$ (the case when $\text{alph}(w') = \Sigma$ being similar). Recall that all non-terminals of G are useful (they can be reached from the starting symbol, and a terminal word can be derivated from them). We further note that such a non-terminal $X \in V$ exists if and only if G contains, for some non-terminal X, derivations $X \Rightarrow^* w_a X w'_a$, with $w_a \in \Sigma^* a \Sigma^*$ and $w'_a \in \Sigma^*$, for all $a \in \Sigma$.

For this we construct a series of data structures.

First, we construct an $n \times n$ matrix M, indexed by the non-terminals of G, where $M[A][B] = 1$, for some $A, B \in R$, if and only if there exist a derivation $A \Rightarrow^* \alpha B \beta$, where $\alpha, \beta \in \Sigma^*$; otherwise, $M[A][B] = 0$. This matrix can be trivially computed in $O(V^3)$. Basically, we define a relation R over V, where $(A, B) \in V$ if and only if $A \to BC$ or $A \to CB$ are productions of G (for some non-terminal C). Constructing this relation takes, in the worst case $O(n^3)$ time. Then, we compute the transitive closure of this relation, using the Floyd-Warshall algorithm, which takes $O(n^3)$ time.

Secondly, we construct an $n \times \sigma$ matrix M', indexed by the non-terminals and terminals of G, respectively, where $M'[A][a] = 1$, for some $A \in V$ and $a \in \Sigma$, if and only if there exist a derivation $A \Rightarrow^* \alpha a \beta$, where $\alpha, \beta \in \Sigma^*$; otherwise, $M'[A][a] = 0$. This matrix can be trivially computed in $O(n^2\sigma)$. Basically, we first initialize all elements of M' with 0. Then, we set $M'[A][a] = 1$ whenever $A \to a$ is a production of G. Then, for all $A, B \in V$ and $a \in \Sigma$, if $M[A][B] = 1$ and $B \to a$ is a production of G, we set $M'[A][a] = 1$ as well.

The correctness of the constructions above is straightforward (given that a terminal-word can be constructed from every non-terminal).

Thirdly, we construct an $n \times n$ matrix L, indexed by the non-terminals of G, where $L[A][B] = 1$, for some $A, B \in V$, if and only if there exist a production $A \to BC$ in G and a derivation $C \Rightarrow^* \alpha A \beta$, where $\alpha, \beta \in \Sigma^*$; otherwise, $L[A][B] = 0$. This can be computed in $O(n^3)$ time, in the worst case. We initialize all elements of L with 0 and, then, for every production $A \to BC$, we set $L[A][B] = 1$ if and only if $M[C][A] = 1$. Again, the correctness of the construction is immediate.

Finally, we construct an $n \times \sigma$ matrix L', indexed by the non-terminals and terminals of G, respectively, where $L'[A][a] = 1$, for some $A \in V$ and $a \in \Sigma$, if and only if there exist a derivation $A \Rightarrow^* \alpha a \beta A \gamma$, where $\alpha, \beta, \gamma \in \Sigma^*$; otherwise, $L'[A][a] = 0$. This can be computed in $O(n^2\sigma)$ time, in the worst case. We initialize all elements of L' with 0. Then, for every non-terminal $A \in \Sigma$ and every terminal $a \in V$, we set $L'[A][a] = 1$ if and only if there exists a non-terminal B with $M'[B][a] = 1$ (i.e., there exist a derivation $B \Rightarrow* \alpha a \beta'$) and $L[A][B] = 1$ (i.e., there exists a production $A \to BC$ in G and a derivation $C \Rightarrow^* \beta'' A \gamma$). The correctness of the construction follows from the explanations given alongside it.

After L' is computed, we conclude that there exists a non-terminal $X \in V$ for which we have derivations $X \Rightarrow w_a X w'_a$, with $w_a \in \Sigma^* a \Sigma^*$ and $w'_a \in \Sigma^*$, for all $a \in \Sigma$, if and only if there exists such a non-terminal X with $L'[X][a] = 1$, for all $a \in \Sigma$. This can be checked in $O(n\sigma)$ time.

In conclusion, our approach uses $O(\max(n^3, n^2\sigma))$ time to check whether G contains a non-terminal $X \in V$ such that X has a 1-universal cycle, and the statement follows. ◀

Further, we show that Problem 3 is fixed parameter tractable w.r.t. the parameter σ; this also means that the respective problem can be solved in polynomial time for constant-size alphabets (i.e., $\sigma \in O(1)$). Recall that we had an ETH-conditional lower bound for the time complexity of algorithms solving this problem of $2^{o(\sigma)}\text{poly}(n, \sigma)$.

▶ **Theorem 38.** *Problem 3 can be solved in $\mathcal{O}(2^{3\sigma}n^4\sigma)$ time, for a machine with n states.*

Proof. Recall that the input of our problem is a CFG $G = (V, \Sigma, P, S)$ in CNF with $|\Sigma| = \sigma \geq 2$ and $|V| = n$. Additionally we get a positive integer k (given in binary representation).

To solve Problem 3, we first check, using the algorithm from Theorem 37, whether $\iota_\exists(L)$ is finite. If $\iota_\exists(L)$ is infinite, then we answer the given instance of the problem positively. Otherwise, we proceed as follows.

We use a dynamic programming approach, to compute the maximal universality index of a word of L. This essentially uses the result of Lemma 20 which states that such a word is the border of a derivation tree of depth at most $N = 4n\sigma$.

We construct a 4-dimensional matrix $M[\cdot, \cdot, \cdot, \cdot]$, with elements $M[i, A, S_p^A, S_s^A]$ with $A \in V$, $S_p^A, S_s^A \subseteq \Sigma$ and $i \leq N$. By definition, $M[i, A, S_p^A, S_s^A] = \ell$ if ℓ is the maximum number with the property that there exists a word w, which labels the border of a derivation tree of height i rooted in A, so that w starts with a prefix x, with $\text{alph}(x) = S_p^A$, followed by ℓ arches, and a suffix y with $\text{alph}(y) = S_s^A$.

We explain how the elements of this matrix are computed. We initialize all entries with $-\infty$.

In the base case, for $i = 1$ we only need to consider direct productions $(A, a) \in P$ for $A \in V$ and $a \in \Sigma$. The following applies: $M[1, A, \{a\}, \emptyset] = 0$ and $M[1, A, \emptyset, \{a\}] = 0$.

This means that we consider the case where the symbol a is in the prefix and the case where it is in the suffix. Clearly, in each of these cases, the other set is empty, as we only have one letter. Accordingly, the entry itself is also 0, because a symbol alone does not form an arch.

In the inductive step, we have that the depth of the considered derivation trees is $i > 1$, and the topmost level of such a derivation tree is defined by a production of the from $A \to BC$. So, we fix such a production $A \to BC \in P$ with $A, B, C \in V$ and we see what role it plays in computing the entry $M[i, A, S_p^A, S_s^A]$. Let u be a word derivable from B and v a word derivable from C. If u has a suffix x with $\text{alph}(x) = S_s^B$ and v has a prefix y with $\text{alph}(y) = S_p^C$ so that $S_s^B \cup S_p^C = \Sigma$, we can concatenate u and v and thus obtain a new arch. Accordingly, a possible candidate for $M[i, A, S_p^A, S_s^A]$ is obtained by adding $M[i-1, B, S_p^A, R]$ and $M[i-1, C, \Sigma \setminus R, S_s^A]$, for some subset $R \subsetneq \Sigma$, and add one for the new arch. We then take the maximum over all these combinations of alphabets R and $\Sigma \setminus R$. We get

$$c_{A\to BC} = \max_{R\subseteq\Sigma}(M[i-1, B, S_p^A, R] + M[i-1, C, \Sigma \setminus R, S_s^A] + 1). \tag{13}$$

If A has t productions $p_1, \ldots, p_t$ we get $c_{p_1}, \ldots, c_{p_t}$. Then $M[i, A, S_p^A, S_s^A]$ is the maximum of $c_{p_1}, \ldots, c_{p_t}$.

The inductive step is repeated for i from 2 to $4n\sigma$. Then, to determine $\iota_\exists(L)$, we take the maximum Q over the entries of $M[4n\sigma, S, \emptyset, T]$, over all subsets $T \subsetneq \Sigma$, as we only consider words w that lie in L, so words that can be derived from S.

To solve the given instance of Problem 3, we compare Q with k. Accordingly, if $k \leq Q = \max_{T\subseteq\Sigma}(M[4m\sigma, S, \emptyset, T])$, then we answer the respective instance positively. Otherwise, we answer it negatively.

The correctness of the above algorithm is rather straightforward. For each A and i and subsets R_1, R_2 of Σ, we construct in a bottom-up fashion the derivation trees rooted in A, of depth i, whose

border w starts with a prefix x, with $\text{alph}(x) = R_1$, followed by as many arches as possible, and a suffix y with $\text{alph}(y) = R_2$. The dynamic programming approach used to construct such a tree is to consider all possible choices for the production rewriting the root of the respective tree, and then find the subtrees corresponding to the non-terminals on the level 1 with a maximum number of arches, which can be joined together to obtain a new arch through this joining, while still fulfilling the conditions regarding the prefix and suffix. If any of these trees would be chosen in another way, then a tree with a lower or equal number of arches would be obtained instead.

As far as the complexity of this approach is concerned, let us fix some $i \in [4\sigma n]$. For $i = 1$, the algorithm runs in $\mathcal{O}(n\sigma)$, in the worst case. For this i, the inductive step of the dynamic programming runs in $\mathcal{O}(n^3 2^{3\sigma})$ time, in the worst case. Indeed, we consider all productions $A \to BC$, then all sets S_p^A, S_s^A, and try to determine an entry $M[i, A, S_p^A, S_s^A]$. For that we go over all choices of a set R allowing us to combine $M[i-1, B, S_p^A, R]$ and $M[i-1, C, \Sigma \setminus R, S_s^A]$. As said, we iterate this step $4n\sigma$ times, which leads to an overall complexity of $\mathcal{O}(2^{3\sigma} n^4 \sigma)$. ◀

We observe that the algorithm supporting the result of Theorem 38 uses exponential space. However, there is a simple (non-deterministic) PSPACE-algorithm solving this problem. Such an algorithm constructs non-deterministically the left derivation of a word $w \in L$ of universality at least k (i.e., a derivation where the leftmost non-terminal is always rewritten); note that w is also non-deterministically guessed, and it is never constructed, written, or stored explicitly in our algorithm. Assume that this left derivation of w is $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_t = w$. Moreover, assume that $w_i = w_i' w_i''$, where $w_i \in \Sigma^*$ and $w_i'' \in V^*$; it is immediate that $|w_i''|$ is upper bounded by the depth of the derivation tree associated to the above derivation of w. Further, there are two observations that confirm our claim. Firstly, at an intuitive level, this derivation can be split into sequences of derivations in which the leftmost non-terminal is rewritten several times using productions of the type $A \to BC$ and then it is turned into a terminal; these derivations correspond to constructing a simple-path, from the root to a leaf, in the derivation tree: this is the lefmost path which was not yet constructed. Due to Lemma 20, such a sequence of derivations has at most $4n\sigma$ steps. Secondly, during the simulation of this left derivation, at step i, we do not keep track of the maximal prefix w_i' consisting only of terminal letters of the sentential form w_i, but of $\iota(w_i')$ and $\text{alph}(r(w_i'))$ and of w_i''; this is enough for our purpose: computing the universality of the derived word. So, the information memorized by our algorithm can be clearly stored in polynomial space. If, and only if, at the end of the derivation, the maintained universality index is at least k, we accept the input grammar and number k.

7 Problem 5

7.1 Problem 5 for CFL

Since we want to check if we find for every positive integer m a word $w \in L$ with $\iota(w) \geq m$, we need to check if we can produce words with arbitrary large universality therefore we check if the grammar G contains some variable A with a one universal cycle. According to Lemma 19 we are looking for an A with the following properties:

1. A can be derived from S,
2. A is terminable, i.e. there is a derivation $A \Rightarrow^* w$ with $w \in \Sigma^*$,
3. there are words $w_1, w_2 \in \Sigma^*$ with $\iota(w_1) + \iota(w_2) \geq 1$ and $A \Rightarrow^* w_1 A w_2$.

It is therefore sufficient to find a variable that produces and terminates a universal cycle. In the following, we present a polynomial time algorithm that finds such a variable. For this, we use a dynamic programming approach similar to $\iota_\forall(L)$. That is, we assume that we obtain L as CFG $G = (V, \Sigma, P, S)$ in CNF. Our procedure is divided into three steps. In the first step, we construct a

matrix M that stores for each $A \in V$ whether a parse tree of depth i exists, such that we find for every $a \in \Sigma$ a word w derivable in i steps from A with $a \in \text{alph}(w)$.

In the second step, we create matrices R and L. The matrix R stores for all $A \in V$ whether a parse tree with depth i exists, so that we obtain the sentential form wB with $a \in \text{alph}(w)$ for $a \in \sigma$ and $B \in V$. For $A \in V$ we iterate over all $a \in \Sigma$. The matrix L stores analogously to R whether a parse tree of height i of A exists, so that we get Bw.

In the last step, we use R and L to check whether an $A \in V$ exists with $A \Rightarrow^* wA$ or $A \Rightarrow* Aw$, so that we cover all $a \in \Sigma$. If this is the case, we insert the dififerent parse trees into each other and obtain an arch. Since we can therefore produce arbitrary many arches, it follows that $\iota_\exists(L)$ is infinite.

In the first step, we construct the matrix

$$M[i, A, a] = k$$

with $i \in [1, |V|]$, $A \in V$, $a \in \Sigma \cup \{\#\}$ and $k \in \{0, 1\}$. Here $k = 1$ applies if $a \neq \#$ and a word w with $a \in \text{alph}(w)$ exists, so that we obtain w by a parse tree of depth i. If $a = \#$, then $k = 1$ if a $b \in \Sigma$ exists with $M[i, A, b] = 1$ for a fixed i and A. This is a quick lookup for the update step. Here, i is limited upwards by $|V|$, as we only check whether the derived word contains a certain letter. It is therefore sufficient to consider each variable exactly once. This is similar to Claim 30 for the algorithm for problem 4.

For the base case, we obtain $M[1, A, a] = 1$ if a production $(A, a) \in P$ exists. If $M[1, A, a] = 1$ for any $a \in \Sigma$, we set $M[1, A, \#] = 1$. In the induction step, we consider derivations of the form (A, BC) with $A, B, C \in V$. This gives us four cases. In the first case, we check whether we can derive a word w from B in $i - 1$ steps so that $a \in \text{alph}(w)$ holds. In addition, we require that C is terminable in at most $i - 1$ steps. We obtain the following condition

$$c_1 := \min(M[i-1, B, a], \max_{j \in [1,i-1]} M[j, C, \#]). \tag{14}$$

With the minimum we make sure that both conditions are fulfilled, because if an entry is 0, it is the minimum and therefore $c_1 = 0$. In the second case, we check whether a word w with $a \in \text{alph}(w)$ can be derived from C in $i - 1$ steps and also make sure that B can be terminated in a maximum of $i - 1$ steps, we get

$$c_2 := \min(M[i-1, C, a], \max_{j \in [1,i-1]} M[j, B, \#]). \tag{15}$$

Analogous to c_1 we form the minimum. In the third case, we no longer require that a word w with $a \in \text{alph}(w)$ is derivable in exactly $i - 1$ steps, instead, it should only apply to a $j \leq i - 1$. In the third case, we first check this condition for C. Since we still build a tree with depth i, B must be terminable in exactly $i - 1$ steps, resulting in

$$c_3 := \min(M[i-1, B, \#], \max_{j \in [1,i-1]} M[j, C, a]). \tag{16}$$

In the last case, we check whether a word w with $a \in \text{alph}(w)$ can be derived from B in at most $i - 1$ steps and accordingly we make sure that C can be terminated in exactly $i - 1$ steps. We obtain the following equation

$$c_3 := \min(M[i-1, C, \#], \max_{j \in [1,i-1]} M[j, B, a]). \tag{17}$$

For $M[i, A, a] = 1$, at least one of the four cases must be fulfilled, therefore $M[i, A, a]$ results in

$$M[i, A, a] = \max(c_1, c_2, c_3, c_4). \tag{18}$$

If there exists an $a \in \Sigma$ such that $M[i, A, a] = 1$, then $M[i, A, \#] = 1$. For the runtime we get the following result.

▶ **Lemma 39.** *The matrix M can be calculated in $\mathcal{O}(|\Sigma| \cdot |V|^3)$.*

Proof. Let $i \in [1, |V|]$ be fixed. For a fixed i we get $\mathcal{O}(|\Sigma \cdot |V|^2)$, since we iterate over all combinations of $A \in V$ and $a \in \Sigma$. Therefore, we calculate $\mathcal{O}(|\Sigma| \cdot |V|)$ entries. In addition, we need $\mathcal{O}(|V|)$ time per entry, since we take a maximum over $j \in [1, i-1]$, which is limited upwards by $|V|$. Therefore, we get a total of $\mathcal{O}(|\Sigma| \cdot |V|^2)$. Because $i \in [1, |V|]$ applies, a total runtime of $\mathcal{O}(|\Sigma| \cdot |V|^3)$ is achieved. This is therefore polynomial in our input size. ◀

In the next step, we use M to construct matrices R and L that store whether we can derive sentential forms wB or Bw from a variable $A \in V$ so that w contains a certain letter. Since the grammar is in CNF, we only consider productions of the form $(A, BC) \in P$ for R and L. In the following, we consider the procedure for calculating R. The construction of L is analogous. The following applies to R

$$R[i, A, B, a] = k$$

with $i \in [1, |V|]$, $A, B \in V$, $a \in \Sigma \cup \{\#\}$ and $k \in \{0, 1\}$. Here $R[i, A, B, a] = 1$ if $a \neq \#$ and a parse tree of A with depth at most i exists, so that we obtain the sentence $w \cdot B$ with $w \in \Sigma^*$ and $a \in \text{alph}(w)$. Similar to M, $R[i, A, B, \#] = 1$ if there is an $a \in \Sigma$ so that $R[i, A, B, a] = 1$. Similar to M, i is again limited upwards by $|V|$. Since G is in CNF, we have the base case for $i = 2$. The following applies

$$R[2, A, B, a] = 1,$$

if a production (A, CB) exists with $M[1, C, a] = 1$. Correspondingly, $R[2, A, B, \#] = 1$ if there is an $a \in \Sigma$ such that $R[2, A, B, a] = 1$.

In the induction step we consider parse trees of height i and productions (A, CD) with $A, C, D \in V$ for the entry $R[i, A, B, a]$. We again obtain four cases. Since the variable D is on the right in the production (A, CD), we only consider the matrix R for D and use the entries from M for C. In the first case, we check whether a sentential form wB with $a \in \text{alph}(w)$ can be derived from D in $i - 1$ steps. In addition, we make sure that C is terminable in at most $i - 1$ steps. We obtain

$$c_1 := \min(R[i-1, D, B, a], \max_{j \in [1, \dots i-1]} M[j, C, \#]). \tag{19}$$

In the second case, we require that a sentential form wB can be derived from D in $i - 1$ steps, but w does not have to contain a. Instead, we check whether a word w with $a \in \text{alph}(w)$ can be derived from C in a maximum of $i - 1$ steps. This results in

$$c_2 := \min(R[i-1, D, B, \#], \max_{j \in [1, \dots i-1]} M[j, C, a]). \tag{20}$$

In the third case, we make sure that C is terminable in $i - 1$ steps and that a $j \leq i - 1$ exists for D, so that we obtain a sentential form wB with $a \in \text{alph}(w)$ from D in j steps. We obtain the following equation

$$c_3 := \min(M[i-1, C, \#], \max_{j \in [1, \dots i-1]} R[j, D, B, a]). \tag{21}$$

In the last case, we check whether a word w with $a \in \text{alph}(w)$ can be derived from C in $i - 1$ steps. Accordingly, we ensure that a sentential form wB can be derived from D in a maximum of $i - 1$ steps. This results in

$$c_4 := \min(M[i-1, C, a], \max_{j \in [1, \dots i-1]} R[j, D, B, \#]). \tag{22}$$

Since at least one of the conditions must apply, we obtain the following expression for $R[i, A, B, a]$

$$R[i, A, B, a] = \max(c_1, c_2, c_3, c_4). \tag{23}$$

Analogue to R we construct a matrix

$$L[i, A, B, a] = k$$

with $i \in [1, |V|]$, $A, B \in V$, $a \in \Sigma \cup \{\#\}$ and $k \in \{0, 1\}$. Here $L[i, A, B, a] = 1$ if $a \neq \#$ and a parse tree of A with depth at most i exists, so that we obtain the sentential form $B \cdot w$ with $w \in \Sigma^*$ and $a \in \text{alph}(w)$. Similar to R and M, $L[i, A, B, \#] = 1$ if there is an $a \in \Sigma$ so that $L[i, A, B, a] = 1$.

▶ **Lemma 40.** *The matrices R and L can be calculated in $\mathcal{O}(|\Sigma| \cdot |V|^4)$.*

Proof. Let $i \in [1, |V|]$ be fixed. For a fixed i we get $\mathcal{O}(|\Sigma \cdot |V|^3)$, since we iterate over all combinations of $A, B \in V$ and $a \in \Sigma$. Therefore, we calculate $\mathcal{O}(|\Sigma| \cdot |V|^2)$ entries. In addition, we need $\mathcal{O}(|V|)$ time per entry, since we take a maximum over $j \in [1, i-1]$, which is limited upwards by $|V|$. Therefore, we get a total of $\mathcal{O}(|\Sigma| \cdot |V|^3)$. Because $i \in [1, |V|]$ applies, this results in a total runtime of $\mathcal{O}(|\Sigma| \cdot |V|^4)$. This is therefore polynomial in our input size. ◀

Since we obtain a runtime of $\mathcal{O}(|\Sigma| \cdot |V|^3)$ for M and a runtime of $\mathcal{O}(|\Sigma| \cdot |V|^4)$ for R and L resp., we need a total of $\mathcal{O}(|\Sigma| \cdot |V|^4)$ time to construct the matrices.

▶ **Theorem 41.** *Let L be a CFL. We can decide whether $\iota_\exists(L)$ is finite in $\mathcal{O}(|\Sigma| \cdot |V|^4)$ time.*

Proof. First we can construct R and L in $\mathcal{O}(|\Sigma| \cdot |V|^4)$ time. Then we check whether there is a variable $A \in V$ such that $R[i, A, A, a] = 1$ or $L[i, A, A, a] = 1$ for all $a \in \Sigma$ and any $i \in [1, |V|]$. If this is the case, we find a variable with which we always generate an arch by inserting the different parse trees and thus $\iota_\exists(L)$ is infinite. This we do ub $\mathcal{O}(|V|^2 \cdot |\Sigma|)$ time, since we iterate over all $A \in V$ and $a \in \Sigma$ and also consider all parse trees of heights $i \in [1, |V|]$. Overall we get $\mathcal{O}(|\Sigma| \cdot |V|^4)$ because of the construction of the matrices. ◀

8 The inbetweeners: DFAwtl

8.1 More concise, for paper

One of the first questions that arise from this work is whether the gap in between polynomial decidability for CFL and undecidability for CSL is filled by other automata model.

Imagine a production line in which there is a shortage of raw materials, for some of the components. Using classic automata models to simulate this leads to a stop in production until all materials become available. However, in situations as such the production usually proceeds with focus on different components, and goes back, as soon as materials are available, in order to complete the whole production mechanism.

While classic finite automata have good algorithmic properties their computational power is rather limited. One version of automata that read words in a non-sequential manner are finite automata with translucent letters, introduced by [47]. In terms of the above scenario, they represent a good simulation. These models are strictly more powerful than classical finite automata, with their extra capabilities stemming from a *translucent transition* that is present in their description, but not as powerful as other models that are allowed to jump symbols in their processing, e.g., see [45] or even [15].

In what follows, we discuss some problems from Section 1 in relation to a slightly different version of the DFAwtl model, presented in [46]. This latter version is slightly more restrictive than the original in the sense that it drops the end marker from the original definition and requires that the whole input is processed before acceptance. In other words, we define an automata with translucent letters as a quintuple $A = (Q, \Sigma, S, F, \delta)$, where Q is a set of states, Σ is the input alphabet, δ is a mapping from $Q \times \Sigma$ to 2^Q, $S \in Q$ is the set of initial states, and $F \in Q$ is the set of final states. As in the classical model, a word is accepted whenever the reading tape is empty and we end up in a final state. However, in the current setting, we define a transition relation on the set of configurations of A as follows $\delta(q, xay) \to (p, xy)$ if and only if $\delta(q, a) = p$ and $\delta(q, b)$ is not defined, for any $b \in \text{alph}(x)$, $a, b \in \Sigma$, and $q, p \in Q$. We call an operation through which the current read symbol is ignored in the current state, i.e., $\delta(q, b)$ is not defined due to a lack of transitions with b from q, a *translucent transition*. The class of languages accepted by the more restrictive deterministic finite automata with translucent letters, for short DFAwtl, strictly includes the class REG and is still incomparable with CFL and the above mentioned class of rational trace languages. The recent survey [49] overviews the extensive literature regarding variations of these types of machines.

A first observation is that in terms of execution, in each step a DFAwtl reads the leftmost symbol possible, i.e., that has not been previously read and there is a transition from the current state labeled with it. Therefore, for every letter, individually, the processing in DFAwtl is done sequentially.

▶ **Lemma 42.** *Consider a DFAwtl A and word $w \in L(A)$, such that* $\text{alph}(w) = \{a_1, \dots, a_\sigma\}$. *After k steps of the computation of A on w such that the number of read occurrences of the symbol a_i is k_i, with $\sum_{i=1}^{\sigma} k_i = k$, the vector $(k_1, \dots, k_\sigma)$ determines the remaining input uniquely.*

As a consequence of the above we establish the following equivalence in the binary alphabet case.

▶ **Lemma 43.** *For a given DFAwtl A with input alphabet $\{a, b\}$, we can construct a pushdown automata B such that $L(A) = L(B)$, in time $\mathcal{O}(|A|)$.*

Therefore, the class of languages accepted by DFAwtl goes outside that of CFLs and becomes incomparable with it, see [48], only starting from the ternary alphabet case. As an application of the above lemma, we get the next result for free.

▶ **Corollary 44.** *The languages accepted by DFAwtl over binary alphabets are CF.*

As a consequence of the corollary we get the following.

▶ **Corollary 45.** *Over binary alphabets, all problems of Section 1 are decidable in polynomial time for a DFAwtl A.*

Thus our interest now shifts to alphabets Σ of size 3 or greater, in the context of DFAwtls. For Problem 1 we first note that we cannot apply the straightforward solution of constructing the finite automaton accepting the upward closure of w, and check whether it has an empty intersection with any given language accepted by DFAwtl, since one can encode the solution set of any Post correspondence problem (PCP) as the intersection of a regular language and a DFAwtl language.

▶ **Proposition 46.** *For any PCP instance there exist $L \in REG$ and a DFAwtl M such that the PCP instance has a solution if and only if $L \cap L(M)$ is not empty.*

Building on Lemma 42, the decidability of Problem 1 for larger alphabets in the case of DFAwtl is settled by remembering at each position all possible subsequnces still needed to be processed.

▶ **Theorem 47.** *Problem 1 is decidable in time $\mathcal{O}(nm^{\sigma})$ for a DFAwtl$A$, where n is the number of states of A, m is the length of w and σ is the size of the alphabet of w.*

By a reduction from the Hamiltonian cycle problem we are able to show that in this case no polynomial time solution is possible.

▶ **Theorem 48.** *Problem 1 is NP-hard over unbounded alphabets.*

Talk about downward closure and why doesn't it work, or might not work, and what we would have gotten from it for free?

8.2 Longer to Appendix

While classic finite automata have nice algorithmic properties their computational power is rather limited. Imagine a production line in which there is a shortage of raw materials. In such situations, the production proceeds with different components, and goes back, as soon as materials are available in order to complete whole production mechanism.

One variation of automata that read words in a non-sequential manner are finite automata with translucent letters, introduced by [47]. In terms of the above scenario, they represent a good simulation. These models are strictly more powerful than classical finite automata, with their extra capabilities stemming from a *translucent transition* that is present in their description, but not as powerful as other models that are allowed to jump symbols in their processing, e.g., see [45] or even [15].

Add full definition of DFAwtl

In terms of execution, in each step the machine reads the leftmost symbol possible, i.e., that has not been previously read and there is a transition from the current state labeled with it. The nondeterministic version of this automata model accepts all rational trace languages, and all accepted languages have semi-linear Parikh images. Moreover, the class of languages accepted by this model is incomparable to the class of CFL, while still being CS. The class of languages accepted by the more restrictive deterministic finite automata with translucent letters, for short DFAwtl, strictly includes the class REG and is still incomparable with CFL and the above mentioned class of rational trace languages. The recent survey [49] overviews the extensive literature regarding variations of these types of machines.

In what follows, we discuss the proposed problems of Section 1 in relation to a slightly different version of the DFAwtl model which was presented in [46]. This latter version of the model is slightly more restrictive than the original in the sense that it drops the acceptance of words for which not symbols have been read.

For Problem 1 we first note that we cannot apply the straightforward solution of constructing the finite automaton accepting the upward closure of w, and check whether it has an empty intersection with any given language accepted by DFAwtl, because one can encode the solution set of any Post correspondence problem (PCP) as the intersection of a regular language and a DFAwtl language.

Proposition 46. For any PCP instance there exist $L \in \mathrm{REG}$ and a DFAwtl M such that the PCP instance has a solution if and only if $L \cap L(M)$ is not empty. ◀

Proof. Let the PCP instance be given as the word vectors $(u_1, \ldots, u_k)$ and $(v_1, \ldots, v_k)$ over an alphabet Σ. We construct the DFAwtl M as follows. The input alphabet of M is $\{1, \ldots, k\} \cup \bigcup_{i=1}^{4} \Sigma_i$, where $\Sigma_i = \{a_i \mid a \in \Sigma\}$. For each $i \in [1,4]$ let pr_i be the morphism that turns words over Σ into words over Σ_i by simply attaching a subscript i to each letter ($pr_i(a) = a_i$). For each $j \in [1,k]$ there are states p_j such that $\delta(q_0, j) = p_j$, where q_0 is the initial state. For each j, we define paths in M such that $\delta(p_j, pr_1(u_j)) = r_j$, and all words $x \in \Sigma_1^*$ that are not lexicographically comparable with $pr_1(u_j)$ lead to a sink state from p_j. From r_j we define a path such that $\delta(r_j, pr_2(v_j)) = s_j$ and all words $x \in \Sigma_2^*$ lexicographically incomparable with $pr_2(v_j)$ lead to a sink state from r_j. From s_j we have a single transition $\delta(s_j, \#) = q_0$. The construction so far would accept shuffles $i_1 \cdots i_\ell \sqcup pr_1(u_{i_1} \cdots u_{i_\ell}) \sqcup pr_2(v_{i_1} \cdots v_{i_\ell}) \sqcup \#^\ell$, that is, makes sure that the words chosen from both word vectors correspond to the same indices. The next part is for checking whether the concatenation of the words is actually equal, that is, $u_{i_1} \cdots u_{i_\ell} = v_{i_1} \cdots v_{i_\ell}$. We add a transition $\delta(q_0, \#) = q_1$. From q_1 we add transitions $\delta(q_1, a_3) = t_a$ for each $a_3 \in \Sigma_3$ and from t_a we add $\delta(t_a, a_4) = z_a$. From z_a we return to q_1 by reading #. We set q_1 to be the only final state. Now the machine accepts shuffles

$$i_1 \cdots i_\ell \sqcup pr_1(u_{i_1} \cdots u_{i_\ell}) \sqcup pr_2(v_{i_1} \cdots v_{i_\ell}) \sqcup pr_3(w) \sqcup pr_4(w) \sqcup \#^m$$

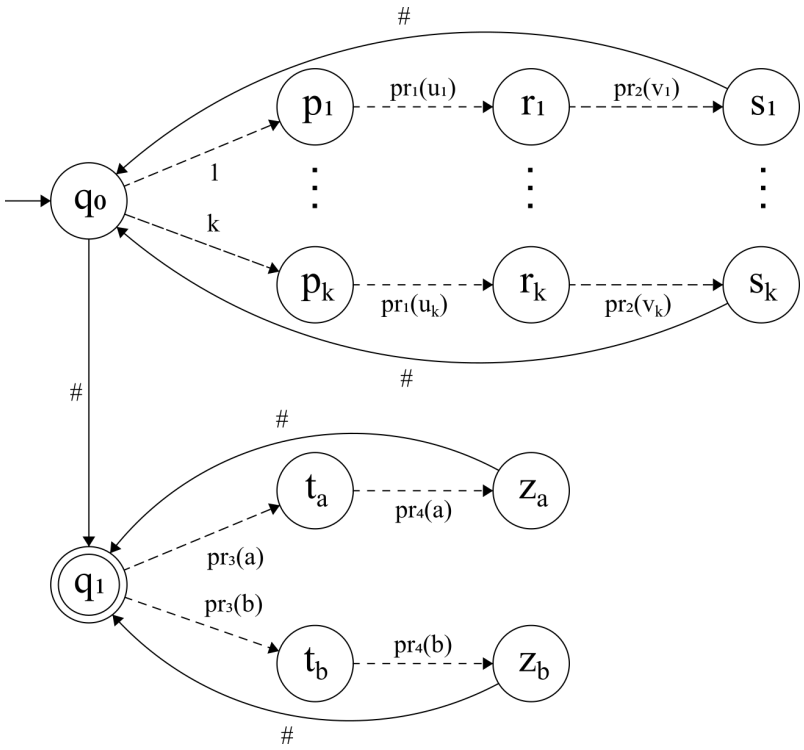

Figure 1 The DFAwtl accepting $i_1 \cdots i_\ell \sqcup pr_1(u_{i_1} \cdots u_{i_\ell}) \sqcup pr_2(v_{i_1} \cdots v_{i_\ell}) \sqcup pr_3(w) \sqcup pr_4(w) \sqcup \#^m$.

where $w \in \Sigma^*$ and $m = \ell + |w|$, but it has no way of checking whether $w = u_{i_1} \cdots u_{i_\ell}$ (see Fig. 1). For that and for making sure that the input is formatted the correct way for M to process, we intersect $L(M)$ with the regular language

$$(\bigcup_{i=1}^{k})^*(\bigcup_{a\in\Sigma} a_1 a_3)^*(\bigcup_{a\in\Sigma} a_2 a_4)^*\#^*.$$

The resulting language consists exactly of the words encoding solutions of the PCP instance, which means that the language is empty if and only if the PCP instance has a solution. ◀

▶ **Corollary 49.** *Emptiness of intersection of DFAwtl with regular languages is undecidable.*

Proof. The construction in Proposition 46 reducing PCP instances to emptiness of intersection between a regular language and a DFAwtl language is effective, so decidability of the emptiness problem would mean decidability of PCP, a well-known undecidable problem. ◀

Next we observe that for every letter, individually, the processing in DFAwtl is done sequentially.

Lemma 42. Consider a DFAwtl A and word $w \in L(A)$, such that $\text{alph}(w) = \{a_1, \ldots, a_\sigma\}$. On the acceptance path of w, after k steps of the computation of A on w, we can associate to the remaining input uniquely the vector $(k_1, \ldots, k_\sigma)$ such that the number of read occurrences of the symbol a_i is k_i, with $\sum_{i=1}^{\sigma} k_i = k$. ◀

Proof. Observe that for any symbol a_i, where $i \in [1, \sigma]$, for a transition with a_i from some current state q, we process the first occurrence of a_i from the unread part of w, i.e., for step $k+1$ we increase k_i by 1. That is, the part of w left to read is missing the first k_j occurrences of the symbol a_j, for $j \in [1, \sigma]$, i.e., the description of this subsequence is unique. ◀

▶ **Theorem 50** (47). *Problem 1 is decidable in time $\sigma nm^{\mathcal{O}(\sigma)}$ for a DFAwtl A, where n is the number of states of A, m is the length of w and σ is the size of the alphabet of w.*

Proof. Recall that the input of our problem is a DFAwtl $A = (Q, \Sigma, q_0, F, \delta)$ with $|Q| = n$ and $|\Sigma| = \sigma \geq 2$ and the word w.

To solve Problem 1, we must find if a word u that has w as subsequence is accepted by A. It is straightforward that the Parikh vector of u is at least as large, component wise, as that of w, while there are at most $\prod_{i=1}^{\sigma}(|w|_i + 1) = m' \in \mathcal{O}(m^\sigma)$, vectors that are at most as large as the Parikh vector corresponding to w. Each of these vectors has the form $\mathbf{v} = (k_1, \ldots, k_\sigma)$ with $0 \leq k_i \leq |w|_i$ for $i \in [\sigma]$.

We also note that among all words accepted by the DFAwtl which have w as a subsequence, if any such words exist, there must exist a word u that has length at most mn. Indeed, when considering the acceptance path for u, we claim that we must always read at least one symbol of w for every factor of u processed. If this is not the case and a factor u' of u, with $|u'| > n$, is read in between two consecutive symbols of w, then along this path we must have passed through the same state twice, which means that a shorter such suitable factor could replace u' in the construction of u.

We now go ahead and give our dynamic programming approach. We construct a 3-dimensional matrix $M[\cdot, \cdot, \cdot]$, with elements $M[i, p, \mathbf{v}]$ with $i \in [m]$, $p \in Q$ and $\mathbf{v}$ being one of the vectors smaller than the Parikh vector of w. By definition, $M[i, p, \mathbf{v}] = 1$ if upon reaching state p in our automaton, starting from state q_0, we have travelled a path of length i and have read from w symbols according to the vector $\mathbf{v} = (k_1, \ldots, k_\sigma)$, i.e., after i steps we have read k_ℓ occurrences of the symbol ℓ with $\ell \in [\sigma]$. Otherwise, we set $M[i, p, \mathbf{v}] = 0$. Observe that, following Lemma 42, if an $a \in \Sigma$ from w was matched, it must have been the first occurrence of a as the automaton cannot make a translucent transition over an earlier occurrence of a to read a later such occurrence of a, i.e., in every state p the vector $\mathbf{v} = (k_1, \ldots, k_\sigma)$ uniquely identifies the part of w still needed to be processed. In other words, for every state p and vector $\mathbf{v}$, we can identify in $\mathcal{O}(n)$ the subsequence w' of w that the automaton still needs to process, continuing from that state p.

In the base case, for $i = 0$, since we have not matched any symbol of w yet, for any state $p \in Q$, we initialize all values $M[0, p, (0, 0, \ldots, 0)] = 1$. All other values are set to 0.

In the inductive step, we consider we have traveled a path of length $i > 1$ and that we are in a state p of the automaton, i.e., we computed all of the values $M[i, p, \mathbf{v}]$ for the m' different vectors $\mathbf{v}$. We compute $M[i + 1, \bullet, \bullet]$ from the values $M[i, \bullet, \bullet]$, as follows. For each pair of states $p, q \in Q$, letter $\ell \in \Sigma$ and vector $\mathbf{v} = (k_1, \ldots, k_\sigma)$ with $M[i, p, \mathbf{v}] = 1$, if a transition $\delta(p, \ell) = q$ exists, and the subsequence of w left to be processed from p for the vector $\mathbf{v}$ has a prefix $b_1 \cdots b_j$ such that $b_j = \ell$ and the automaton has no transition $\delta(p, \ell')$, for any $\ell' \in \text{alph}(b_1 \cdots b_{j-1})$, then we set $M[i+1, q, \mathbf{v}'] = 1$, where $\mathbf{v}' = (k_1, \ldots, k_\ell + 1, \ldots, k_\sigma)$, i.e., while transitioning from p to q we read another occurrence of ℓ from w. We call this operation an *update*. Moreover, for all transitions from p to q, we set the value of $M[i + 1, q, \mathbb{v}] = 1$, for any $M[i, p, \mathbb{v}] = 1$. The latter is possible since, in each of these cases, we could have reached state q, while reading yes another symbol of the supersequence u which does not correspond to w, i.e., for Problem 1 we are interested in the existence of a supersequence of the subsequence w which is accepted by the machine. For every update, if $\mathbf{v}'$ equals the Parikh vector of w, we stop and answer yes.

Since while computing M, there cannot be more than n steps between two updates, i.e., $i \leq mn$, and each update corresponds to matching a symbol of w, we can stop the computation of matrix M after nm steps. If we have not yet reported a match to the Parikh vector of w, then we can decide that no u accepted by the automaton exists such that w is a subsequence of u.

The correctness of the construction follows from the explanations given alongside it, while the complexity of the dynamic programming algorithm described is $\mathcal{O}(nm^\sigma)$ (as we have mn iterations, for each of these we have to iterate over σm^σ elements, and in each iteration we perform at most σ

updates each of complexity m). ◀

As a consequence of the above we establish the following equivalence in the binary alphabet case.

Lemma 43. For a given DFAwtl A with input alphabet $\{a, b\}$, we can construct a pushdown automata B such that $L(A) = L(B)$, in time $\mathcal{O}(|A|)$. ◀

Proof. Since we deal with only two letters and a deterministic machine, at any point in the computation we must read at least one of the symbols, and can only jump over unary factors. Moreover, as a consequence of Lemma 42, whenever we are in a state, we cannot apply a translucent transition to a letter unless all other symbols waiting to be processed (we made translucent transitions on) are identical, i.e., we can only make a translucent step with b only if there is no a waiting to be processed, and vice-versa.

Let $A = (Q, \{a, b\}, q_0, F, \delta)$ be a DFAwtl. Construct the pushdown automata $B = (Q \cup Q_a \cup Q_b, \{a, b\}, \{a, b\}, q_0, F, \delta')$, where Q_a and Q_b are copies of Q indexed by the two letters of the alphabet. For each $x \in \{a, b\}$ and each state $p \in Q$ such that $\delta(p, y) = q$ but $\delta(p, x)$ is not defined, where $y \neq x$, we define the following transitions in the PDA:

I think this is not a correct simulation, so I add another

$$\delta'(p, \bot, x) = (p_x, x), \qquad \delta'(p, x, x) = (p, xx), \qquad \delta'(p, x, y) = (q, x), \qquad \delta'(p, y, \varepsilon) = (q, \varepsilon)$$

For all states p such that $\delta(p, x) = q_1$ and $\delta(p, y) = q_2$ we define the transitions:

$$\delta'(p, x, x) = (q, x) \qquad \text{and} \qquad \delta'(p, \bot, x) = (q, \varepsilon).$$

Let $A = (Q, \{a, b\}, q_0, F, \delta)$ be a DFAwtl. Construct the PDA $B = (Q, \{a, b\}, \{a, b\}, q_0, \{f_B\}, \delta')$. For each $x \in \{a, b\}$ and each state $p \in Q$ such that $\delta(p, y) = q$ but $\delta(p, x)$ is not defined, where $y \neq x$, in the PDA we define the transitions $\delta'(p, \bot, \varepsilon) = (q, y)$, $\delta'(p, y, \varepsilon) = (q, yy)$ and $\delta'(p, x, y) = (q, x)$. For each state p and letter x, we add the transitions $\delta'(p, x, x) = (p, \varepsilon)$. For each state p such that $\delta(p, a)$ and $\delta(p, b)$ are both defined in A we add $\delta'(p, \bot, x) = (q, \varepsilon)$ if $\delta(p, x) = q$.

Let $A = (Q, \{a, b\}, q_0, F, \delta)$ be a DFAwtl. Construct the PDA $B = (Q, \{a, b\}, \{a, b\}, q_0, \{f_B\}, \delta')$. For each $x \in \{a, b\}$ and each state $p \in Q$ such that $\delta(p, y) = q$ but $\delta(p, x)$ is not defined, where $y \neq x$, in the PDA we define the transitions $\delta'(p, \bot, \varepsilon) = (q, y)$, $\delta'(p, y, \varepsilon) = (q, yy)$ and $\delta'(p, x, y) = (q, x)$. For each state p and letter x, we add the transitions $\delta'(p, x, x) = (p, \varepsilon)$. For each state p such that $\delta(p, a)$ and $\delta(p, b)$ are both defined in A we add $\delta'(p, \bot, x) = (q, \varepsilon)$ if $\delta(p, x) = q$.

For each state p of DFAwtl A such that p cannot read x, when the PDA is in a state p, we simulate A's possible translucent transition in the PDA by pushing y on the stack to be matched later with the first remaining y in the input. Whenever the PDA has x on the top of the stack, it means that earlier A would have preformed some transitions on reading x from states to which y was translucent, and during the simulation we have not matched the x yet. In that case if it reads x from the tape, it does not change its state, but removes an x from the stack to simulate that A "already read that symbol earlier". If the PDA has an empty stack, we matched all the symbols read by A by jumping over prefixes, and the PDA can perform identical moves as A. Finally, for each $f \in F$, we add a transition $\delta'(f, \bot, \varepsilon) = (f_B, \varepsilon)$. This makes sure that the input is only accepted with an empty stack, i.e., when all jumps have been accounted for.

The result easily follows with the pushdown counting the translucent transitions. ◀

As an application of the above lemma, we get the next result for free.

Corollary 44. The languages accepted by DFAwtl over binary alphabets are CF. ◀

As a consequence of the corollary and of Theorem 18 we get the following.

Corollary 45. Over binary alphabets, all problems of Section 1 are decidable in polynomial time for a DFAwtl A. ◀

Building on Lemma 42, we can show decidability of Problem 1 for arbitrarily large alphabets.

Theorem 47. Problem 1 is decidable in time $\mathcal{O}(nm^\sigma)$ for a DFAwtl$A$, where n is the number of states of A, m is the length of w and σ is the size of the alphabet of w. ◀

Proof. Assume that we are want to check whether $w = a_1 \cdots a_m$ can be a subsequence of some word accepted by A. Following Lemma 42 if an $a \in \Sigma$ from w was matched, it must have been the first occurrence of a as the automaton cannot jump over an earlier occurrence of a to read a later a, i.e., in any state the subsequence still in need of reading is uniquely determined by the number of occurrences read for each symbol.

I think we have to work on this, as lots of things are handwaved away.

Our algorithm starts from the initial state and tries to match the first letter of w, i.e., a_1. If no transition exists, we check all paths from this state which lead to an a_1 transition. If any translucent transitions are done on the path, which can read, say, a_2 and a_4, but not a_3, then, when we reach the a_1 transition on the path, we already match $a_1a_2a_4$ from the prefix $a_1a_2a_3a_4$ of w, and need to consider now the first transition labeled with a_3. Thus, for each state we remember what letters from w we read so far, and perform another step trying to match the first remaining letter from w. Since we only care about the letter counts, the number of possible tuples to remember for each state is $\mathcal{O}(m^\sigma)$, i.e., the number of all possible subsequences. In each step of the algorithm, we increase the tuples, reducing the remaining subsequence to match, so the number of such 'macro' steps is $\mathcal{O}(n)$.

Note that in this greedy approach we match within w with the first possible transition of that label, for each path. If the computation on the machine got stuck, i.e, we reach a state from which we do not have a transition with any of the remaining letters, we can add to our computation extra symbols that allow us to proceed further from such states. This is possible since for Problem 1 we are interested in the existence of a supersequence of the subsequence w which is accepted by the machine. ◀

Theorem 48. Problem 1 is NP-hard over unbounded alphabets. ◀

Proof. We reduce the Hamiltonian cycle problem to Problem 1. Let $G = (\{v_1, \ldots, v_n\}, E)$ be an undirected simple graph. We define a DFAwtl $A = (Q, \{v_1, \ldots, v_n, a, b\}, q_0, F, \delta)$ that accepts some word that is a supersequence of $v_1 \cdots v_n$ if and only if G is a Hamiltonian graph. First we construct a gadget A_1 that tracks the first step of the Hamiltonian path and then create n copies of it such that the automaton moves from gadget A_i to A_{i+1} if and only if the corresponding current path in G can be extended by another edge. Thus the automaton accepts words that correspond to paths of length n in G, so $v_1 \cdots v_n$ is a subsequence of an accepted word if and only if there is a closed path of length n starting from v_1 reaching all vertices.

Within each gadget A_i with $i \in [1, n]$ we have states $v_{i,j}$ and $w_{i,j}$, for each $j \in [1, n]$. We also have additional states in the gadget that allow, for each $j, k \in [1, n]$, to transition from $v_{i,j}$ to $w_{i,k}$ by reading some word u_k if $v_j v_k$ is an edge of G. A simple choice for $u_1, \ldots, u_n$ is the first n words when we lexicographically order the words of length $\lceil \log_2 n \rceil$ over alphabet $\{a, b\}$. All other binary paths of length $\lceil \log_2 n \rceil$ from $v_{i,j}$ go to the sink state. For each $w_{i,k}$ we have a single transition, which is labeled v_k and goes to state $v_{i+1,k}$, i.e., to the next gadget. For gadget A_n only $w_{n,1}$ has a transition going out of the gadget, to state $v_{n+1,1}$.

The initial state is $v_{1,1}$. There is a single final state, i.e., $v_{n+1,1}$. Fig. 2 illustrates the construction for a simple input graph.

The automaton accepts exactly the set of words

$$v_{i_1} \sqcup \cdots \sqcup v_{i_n} \sqcup u_{i_1} \cdots u_{i_n},$$

where $v_{i_1} \cdots v_{i_n}$ is a walk on G. Therefore, G is Hamiltonian if and only if $v_1 \cdots v_n$ is a subsequence of some word in the language.

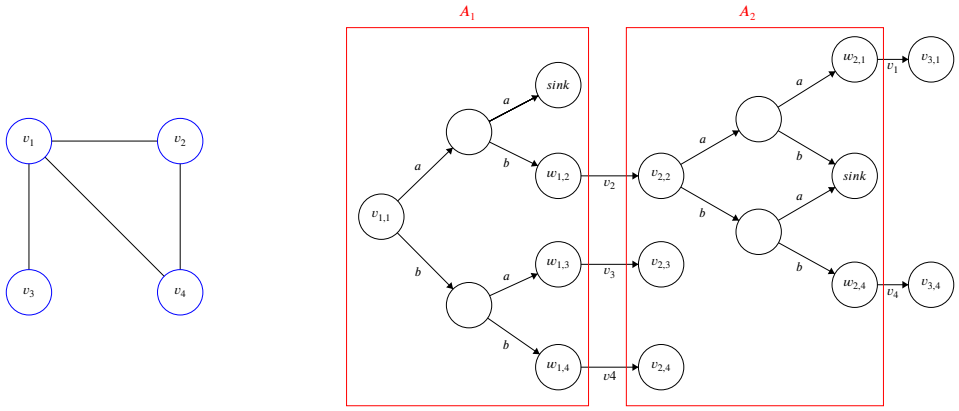

Figure 2 A simple graph (without a Hamiltonian cycle) and part of the corresponding DFAwtl as constructed in the proof of Theorem 48. Gadgets A_1 and A_2 are marked with rectangles.

The number of states and transitions in each gadget A_i is $\mathcal{O}(n^2)$, since we have n states $v_{i,j}$, further n states $w_{i,j}$ and for each $v_{i,j}$ an additional $\mathcal{O}(n)$ intermediate states forming the complete binary tree that starts from $v_{i,j}$ and leads to each $w_{i,k}$, in turn. As the number of gadgets is n, we obtain a DFAwtl with $\mathcal{O}(n^3)$ states. ◀

Talk about downward closure and why doesn't it work, or might not work, and what we would have gotten from it for free?

The next lemma shows that a cycle of a DFAwtl is used arbitrarily many times (in particular, more than once) in order to increase the universality index of words accepted by the machine only when leaving the cycle (without return) is done via transitions with symbols that we have already processed arbitrarily many times beforehand, i.e., were part of a cycle processed already.

▶ **Lemma 51.** *For a DFAwtl $A = (Q, \Sigma, q_0, F, \delta)$, if there exist states $q_1, q \in Q$, symbol $a \in \Sigma$ and integer $\ell > 0$ such that $\delta(q_1, a) = q$ and, for every $u \in L_{q_1}$, we have $|u_a| < \ell$, then every minimal length $v \in L$ with $\iota(v) \geq k$, for some k, goes through cycles transversing q_1, provided such cycles exist, at most once.*

Proof. We prove this by a counterpositive argument. Consider a word w that is of minimal length and has universality index arbitrarily large. Moreover, assume that processing this word using A leads to the computation going twice through some cycle $(q_1, q_2, \dots, q_k)$ and then reading symbol $a \in \Sigma$ from q_1, and that every prefix of w from L_{q_1} has a bounded number of a's. Consider the shortest prefix $w'_p \in L_{q_1}$ of w, i.e., w'_p contains a fixed number of a's, denote by w''_p the shortest prefix of w in L_q, where $\delta(q_1, a) = q$. Hence there exist words w'_s, w''_s such that

$$w = w'_p w'_s = w''_p w''_p.$$

We first note that no transition of the cycle $(q_1, q_2, \dots, q_k)$ is labelled with a. This is because of our restriction on the number of possible occurrences of this letter in any word of L_{q_1}. Now, if while processing w'_s using A we go through the cycle $(q_1, q_2, \dots, q_k)$ twice, we read in fact each of the labels of the transitions in this cycle twice, while symbol a only once. Therefore, there exists another word x which is a strict subsequence of w'_s and has w''_s as suffix, such that $\iota(w'_p x) = \iota(w)$ and $|w'_p x| < |w|$. ◀

▶ **Theorem 52.** *Problem 5 is decidable.*

Proof. For a given automaton A, in order to have $\iota_\forall(L(A))$ to be infinite, for any arbitrarily large k we have to have $w \in L(A)$ with $\iota(w) = k$. If the automaton contains a loop with all the symbols of the alphabet, then from the REG case [4], we are done and answer yes. We now consider this is not the case, and look for necessary and sufficient conditions when such a word exists.

Obviously the first of the necessary conditions is for each symbol of the alphabet to be read an infinite number of times. Thus we conclude that it is necessary for the union of all symbols read on loops on some accepting path to amount to the whole alphabet. Let us now fix some notation for the rest of this proof. We consider w to be the shortest word with $\iota(w) = k$. It must be that all loops on the accepting path of w are strongly connected components, denoted $Q_1, Q_2, \dots, Q_p$, such that there is a path $Q_1 \to Q_2 \to \cdots \to Q_p$, but there is no path $Q_{i+1} \to^* Q_i$, for any $i \in [1..n-1]$. Observe that any two such components are disjoint, while each consists of at least one state. Denote by $q_i \in Q_i$ the state used to enter cycle Q_i along the considered path, by Q'_i the set of symbols used to exit the component Q_i, and by q'_i the state used to exit Q_i along the considered path. For each Q_i we denote by Q_i^+ the set of all symbols which are read within Q_i (are part of the considered cycle on the path) and by Q_i^- the set of symbols for which we do not have any transitions

- for any state in Q_i (that is $Q'_i \cup Q_i^- = \emptyset$)
- within the considered cycle in Q_i (thus there could exist $q \in Q'_i \cup Q_i^-$)
- for any state in $\bigcup_{j \in [i-1]} Q_j$ (symbols we do not encounter in previously visited connected component)

Following Lemma 51 in addition to the union of these cycles to render the whole alphabet, i.e., $\bigcup_{i \in [p]} Q_i = \Sigma$, we now also need that exiting each Q_i is done only via symbols already processed, i.e., $Q'_i \subseteq \bigcup_{j \in [i]} Q_j^+$.

◀

References

1. Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14.
2. Amir Abboud and Aviad Rubinstein. Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, pages 35:1–35:14, 2018. doi:10.4230/LIPIcs.ITCS.2018.35.
3. Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, pages 39–51, 2014. doi:10.1007/978-3-662-43948-7_4.
4. Duncan Adamson, Pamela Fleischmann, Annika Huch, Tore Koß, Florin Manea, and Dirk Nowotka. k-universality of regular languages, 2023. arXiv:2311.10658.
5. Ashwani Anand and Georg Zetzsche. Priority downward closures. In Guillermo A. Pérez and Jean-François Raskin, editors, *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*, volume 279 of *LIPIcs*, pages 39:1–39:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2023.39, doi:10.4230/LIPICS.CONCUR.2023.39.
6. Alexander Artikis, Alessandro Margara, Martín Ugarte, Stijn Vansummeren, and Matthias Weidlich. Complex event recognition languages: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 7–10, 2017. doi:10.1145/3093742.3095106.
7. Ricardo A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.
8. L. Barker, P. Fleischmann, K. Harwardt, F. Manea, and D. Nowotka. Scattered factor-universality of words. In *DLT*, pages 14–28. Springer, 2020.
9. Laura Barker, Pamela Fleischmann, Katharina Harwardt, Florin Manea, and Dirk Nowotka. Scattered factor-universality of words. In *Proc. DLT 2020*, volume 12086 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2020.
10. Laura Barker, Pamela Fleischmann, Katharina Harwardt, Florin Manea, and Dirk Nowotka. Scattered factor-universality of words. *CoRR*, abs/2003.04629, 2020.
11. Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. *Theor. Comput. Sci.*, 443:25–34, 2012. doi:10.1016/j.tcs.2012.03.029.
12. Karl Bringmann and Bhaskar Ray Chaudhury. Sketching, streaming, and fine-grained complexity of (weighted) LCS. In *Proc. FSTTCS 2018*, volume 122 of *LIPIcs*, pages 40:1–40:16, 2018.
13. Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proc. SODA 2018*, pages 1216–1235, 2018.
14. Sam Buss and Michael Soltys. Unshuffling a square is NP-hard. *J. Comput. Syst. Sci.*, 80(4):766–776, 2014. doi:10.1016/j.jcss.2013.11.002.
15. Hiroyuki Chigahara, Szilárd Zsolt Fazekas, and Akihiro Yamamura. One-way jumping finite automata. *International Journal of Foundations of Computer Science*, 27(3):391–405, 2016.
16. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
17. Maxime Crochemore, Borivoj Melichar, and Zdenek Tronícek. Directed acyclic subsequence graph — overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003.
18. J.D. Day, P. Fleischmann, M. Kosche, T. Koß, F. Manea, and S. Siemer. The edit distance to k-subsequence universality. In *STACS*, volume 187, pages 25:1–25:19, 2021.
19. L. Fleischer and M. Kufleitner. Testing Simon's congruence. In *Proc. MFCS 2018*, volume 117 of *LIPIcs*, pages 62:1–62:13, 2018.
20. F. V. Fomin, D. Kratsch, I. Todinca, and Y. Villanger. Exact algorithms for treewidth and minimum fill-in. *SIAM J. Comput.*, 38(3):1058–1079, 2008. doi:10.1137/050643350.

21 Dominik D. Freydenberger, Pawel Gawrychowski, Juhani Karhumäki, Florin Manea, and Wojciech Rytter. Testing k-binomial equivalence. In Multidisciplinary Creativity*, a collection of papers dedicated to G. Păun 65th birthday*, pages 239–248, 2015. available in CoRR abs/1509.00622.

22 Emmanuelle Garel. Minimal separators of two words. In *Proc. CPM 1993*, volume 684 of *Lecture Notes in Computer Science*, pages 35–53, 1993.

23 Pawel Gawrychowski, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. Efficiently testing Simon's congruence. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, pages 34:1–34:18, 2021. doi:10.4230/LIPIcs.STACS.2021.34.

24 Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. Complex event recognition in the big data era: a survey. *VLDB J.*, 29(1):313–352, 2020. doi:10.1007/s00778-019-00557-w.

25 Simon Halfon, Philippe Schnoebelen, and Georg Zetzsche. Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In *Proc. LICS 2017*, pages 1–12, 2017.

26 Jean-Jacques Hebrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theor. Comput. Sci.*, 82(1):35–49, 22 May 1991.

27 Jean-Jacques Hébrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theor. Comput. Sci.*, 82:35–49, 1991.

28 G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336, 1952.

29 J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

30 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

31 P. Karandikar, M. Kufleitner, and P. Schnoebelen. On the index of Simon's congruence for piecewise testability. *Inf. Process. Lett.*, 115(4):515–519, 2015.

32 P. Karandikar and P. Schnoebelen. The height of piecewise-testable languages with applications in logical complexity. In *Proc. CSL*, volume 62 of *LIPIcs*, pages 37:1–37:22, 2016.

33 P. Karandikar and P. Schnoebelen. The height of piecewise-testable languages with applications in logical complexity. In *CSL*, 2016.

34 P. Karandikar and P. Schnoebelen. The height of piecewise-testable languages and the complexity of the logic of subwords. *LICS*, 15(2), 2019.

35 Dietrich Kuske. The subtrace order and counting first-order logic. In *Proc. CSR 2020*, volume 12159 of *Lecture Notes in Computer Science*, pages 289–302, 2020.

36 Dietrich Kuske and Georg Zetzsche. Languages ordered by the subword order. In *Proc. FOSSACS 2019*, volume 11425 of *Lecture Notes in Computer Science*, pages 348–364, 2019.

37 Martin Lange and Hans Leiß. To CNF or not to cnf? an efficient yet presentable version of the CYK algorithm. *Informatica Didact.*, 8, 2009. URL: http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009.

38 Marie Lejeune, Julien Leroy, and Michel Rigo. Computing the k-binomial complexity of the Thue-Morse word. In *Proc. DLT 2019*, volume 11647 of *Lecture Notes in Computer Science*, pages 278–291, 2019.

39 Julien Leroy, Michel Rigo, and Manon Stipulanti. Generalized Pascal triangle for binomial coefficients of words. *Electron. J. Combin.*, 24(1.44):36 pp., 2017.

40 Chun Li and Jianyong Wang. Efficiently mining closed subsequences with gap constraints. In *SDM*, pages 313–322. SIAM, 2008.

41 Chun Li, Qingyan Yang, Jianyong Wang, and Ming Li. Efficient mining of gap-constrained subsequences and its various applications. *ACM Trans. Knowl. Discov. Data*, 6(1):2:1–2:39, 2012.

42 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Lower bounds based on the exponential time hypothesis. *Bull. EATCS*, 105:41–72, 2011. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/92.

43 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978.

44 Alexandru Mateescu, Arto Salomaa, and Sheng Yu. Subword histories and Parikh matrices. *J. Comput. Syst. Sci.*, 68(1):1–21, 2004.

45 Alexander Meduna and Petr Zemek. Jumping finite automata. *International Journal of Foundations of Computer Science*, 23(7):1555–1578, 2012.

46 Victor Mitrana, Andrei Păun, Mihaela Păun, and José-Ramón Sánchez-Couso. Jump complexity of finite automata with translucent letters. *Theoretical Computer Science*, 992:114450, 2024.

47 Benedek Nagy and Friedrich Otto. Finite-state acceptors with translucent letters. In A.O. De La Puente G. Bel-Enguix, V. Dahl, editor, *BILC 2011: AI Methods for Interdisciplinary Research in Language and Biology*, pages 3–13, 2011.

48 Benedek Nagy and Friedrich Otto. Globally deterministic cd-systems of stateless r-automata with window size 1. *International Journal of Computer Mathematics*, 90(6):1254–1277, 2013.

49 Friedrich Otto. A survey on automata with translucent letters. In Benedek Nagy, editor, *Implementation and Application of Automata*, pages 21–50, Cham, 2023. Springer Nature Switzerland.

50 Rohit Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966. doi:10.1145/321356.321364.

51 William E. Riddle. An approach to software system modelling and analysis. *Comput. Lang.*, 4(1):49–66, 1979. doi:10.1016/0096-0551(79)90009-2.

52 Michel Rigo and Pavel Salimov. Another generalization of abelian equivalence: Binomial complexity of infinite words. *Theor. Comput. Sci.*, 601:47–57, 2015.

53 Arto Salomaa. Connections between subwords and certain matrix mappings. *Theoret. Comput. Sci.*, 340(2):188–203, 2005.

54 P. Schnoebelen and P. Karandikar. The height of piecewise-testable languages and the complexity of the logic of subwords. *Logical Methods in Computer Science*, 15, 2019.

55 P. Schnoebelen and J. Veron. On arch factorization and subword universality for words and compressed words. In *WORDS 2023, Proceedings*, volume 13899 of *Lecture Notes in Computer Science*, pages 274–287. Springer, 2023.

56 Shinnosuke Seki. Absoluteness of subword inequality is undecidable. *Theor. Comput. Sci.*, 418:116–120, 2012.

57 Alan C. Shaw. Software descriptions with flow expressions. *IEEE Trans. Software Eng.*, 4(3):242–254, 1978. doi:10.1109/TSE.1978.231501.

58 I. Simon. Piecewise testable events. In *Autom. Theor. Form. Lang., 2nd GI Conf.*, volume 33 of *LNCS*, pages 214–222. Springer, 1975.

59 Imre Simon. *Hierarchies of events with dot-depth one - Ph.D. thesis*. University of Waterloo, 1972.

60 Imre Simon. Words distinguished by their subwords (extended abstract). In *Proc. WORDS 2003*, volume 27 of *TUCS General Publication*, pages 6–13, 2003.

61 Zdenek Tronícek. Common subsequence automaton. In *Proc. CIAA 2002 (Revised Papers)*, volume 2608 of *Lecture Notes in Computer Science*, pages 270–275, 2002.

62 Georg Zetzsche. The complexity of downward closure comparisons. In *Proc. ICALP 2016*, volume 55 of *LIPIcs*, pages 123:1–123:14, 2016.

63 Georg Zetzsche. Separability by piecewise testable languages and downward closures beyond subwords. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 929–938. ACM, 2018. doi:10.1145/3209108.3209201.

64 Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 217–228, 2014. doi:10.1145/2588555.2593671.