Bounded Prefix-Suffix Duplication

Marius Dumitran , Javier Gil , Florin Manea , and Victor Mitrana

Abstract. We consider a restricted variant of the prefix-suffix duplication operation, called bounded prefix-suffix duplication. It consists in the iterative duplication of a prefix or suffix, whose length is bounded by a constant, of a given word. We give a sufficient condition for the closure under bounded prefix-suffix duplication of a class of languages. Consequently, the class of regular languages is closed under bounded prefix-suffix duplication; furthermore, we propose an algorithm deciding whether a regular language is a finite k-prefix-suffix duplication language. An efficient algorithm solving the membership problem for the k-prefix-suffix duplication of a language is also presented. Finally, we define the k-prefix-suffix duplication distance between two words, extend it to languages and show how it can be computed for regular languages.

1 Introduction

Treating sets of chromosomes and genomes as languages raises the possibility that the structural information contained in biological sequences can be generalized and investigated by formal language theory methods [13]. Thus, the interpretation of duplication as a formal operation on words has inspired a series of works in the area of formal languages opened by [3,14] and continued by several other papers, e.g., [10] and the references therein. In [6] one considers duplications that appear at the both ends of the words only, called prefix-suffix duplications, inspired by the case of telomeric DNA. In this context, one investigates the class of languages that can be defined by the iterative application of the prefix-suffix duplication to a word and tries to compare it to other well studied classes of languages. It is shown that the languages of this class have a rather complicated structure even if the initial word is rather simple.

Several problems remained unsolved in the aforementioned paper. This is the mathematical motivation for the work presented here. By considering a weaker

variant of the prefix-suffix duplication, called bounded prefix-suffix duplication, we are able to solve, in this new setting, some of the problems that remained unsolved in [6]. Another motivation is related to the biochemical reality that inspired the definition of this operation. It seems more practical and closer to the biological reality to consider that the factor added by the prefix-suffix duplication cannot be arbitrarily long. One should note that the investigation we pursue here is not aimed to tackle real biological facts and provide solutions for them. In fact, its aim is to provide a better understanding of the structural properties of strings obtained by prefix-suffix duplication as well as specific tools for the manipulation of such strings.

We give a brief description of the contents of this work. We first define a restricted variant of the prefix-suffix duplication called bounded prefix-suffix duplication. It consists in the duplication of a prefix or suffix whose length is bounded by a constant of a given word. We give sufficient conditions for a family of languages to be closed under bounded prefix-suffix duplication. Consequently, we show that every language generated by applying iteratively the bounded prefix-suffix duplication to a word is regular. We also propose an algorithm deciding whether there exists a finite set of words generating a given regular language w.r.t. bounded-prefix-suffix duplication.

We show that the membership problem for the language obtained by applying iteratively k-prefix-suffix duplications from a language recognizable in $\mathcal{O}(f(n))$ time can be solved in $\mathcal{O}(nk \log k + n^2 f(n))$ time. In particular, when considering the k-prefix-suffix duplication language generated by a word x, this problem can be solved in $\mathcal{O}(n \log k)$ time, if $|x| \geq k$, and $\mathcal{O}(nk \log k)$ time in the general case.

We then define the k-prefix-suffix duplication distance between two given words as the minimal number of k-prefix-suffix duplications applied to one of them in order to get the other one and show how it can be efficiently computed. This distance is extended to languages and we propose an algorithm for efficiently computing the k-prefix-suffix duplication distance between two regular languages.

2 Preliminaries

We assume the reader to be familiar with fundamental concepts of formal language theory and complexity theory which can be found in many textbooks, e.g., [12] and [11], respectively.

We start by summarizing the notions used throughout this work. An *alphabet* is a finite and nonempty set of symbols. The cardinality of a finite set A is written $|A|$. Any finite sequence of symbols from an alphabet V is called a *word* over V. The set of all words over V is denoted by V^* and the empty word is denoted by ε; also V^+ is the set of non-empty words over V, V^k is the set of all words over V of length k, while $V^{\leq k}$ is the set of all words over V of length at most k. Given a word w over an alphabet V, we denote by $|w|$ its length, If $w = xyz$ for some $x, y, z \in V^*$, then x, y, z are called prefix, subword, suffix, respectively, of w. For a word w, $w[i..j]$ denotes the subword of w starting at position i and ending at

position j, $1 \leq i \leq j \leq |w|$; by convention, $w[i..j] = \varepsilon$ if $i > j$. If $i = j$, then $w[i..j]$ is the i-th letter of w which is simply denoted by $w[i]$. A *period* of a word w over V is a positive integer p such that $w[i] = w[j]$ for all i and j with $i \equiv j \pmod{p}$. By $per(w)$ (called *the period of* w) we denote the smallest period of w. If $per(w) < |w|$ and $per(w)$ divides $|w|$, then w is a repetition; otherwise, w is called primitive. A primitively rooted square is a word w that has the form xx for some primitive word x.

We say that the pair ${}_w(i,p)$ is a *duplication* (*repetition*) in w starting at position i in w if $w[i..i+p-1] = w[i+p..i+2p-1]$. Analogously, the pair $(i,p)_w$ is a duplication in w ending at position i in w if $w[i-2p+1..i-p] = w[i-p+1..i]$. In both cases, p is called the length of the duplication. Furthermore, the pair ${}_w(i,p)_w$ is a duplication in w having the middle at position i in w if $w[i-p+1..i] = w[i+1...i+p]$.

Despite that the prefix-suffix operation introduced in [6] is a purely mathematical one and the biological reality is just a source of inspiration, it seems rather unrealistic to impose no restriction on the length of the prefix or suffix which is duplicated. The restriction considered in this paper concerns the length of all prefixes and suffixes that are duplicated to the current word. They cannot be longer than a given constant. This restricted variant of prefix-suffix duplication is called *bounded prefix-suffix duplication*. Formally, given a word $x \in V^*$ and a positive integer k, we define:

– *k-prefix duplication*, namely $PD_k(x) = \{ux \mid x = uy \text{ for some } u \in V^+, |u| \leq k\}$. The *k-suffix duplication* is defined analogously, that is $SD_k(x) = \{xu \mid x = yu \text{ for some } u \in V^+, |u| \leq k\}$.

– *k-prefix-suffix duplication*, namely $PSD_k(x) = PD_k(x) \cup SD_k(x)$.

These operations are naturally extended to languages L by

$$PD_k(L) = \bigcup_{x \in L} PD_k(x), \quad SD_k(L) = \bigcup_{x \in L} SD_k(x), \quad PSD_k(L) = \bigcup_{x \in L} PSD_k(x).$$

We further define, for each $\Theta \in \{PD, SD, PSD\}$:

$$\Theta_k^0(x) = \{x\}, \Theta_k^{n+1}(x) = \Theta_k^n(x) \cup \Theta_k(\Theta_k^n(x)), \text{ for } n \geq 0, \ \Theta_k^*(x) = \bigcup_{n \geq 0} \Theta_k^n(x).$$

Furthermore, $PSD_k^*(L) = \bigcup_{x \in L} PSD_k^*(x)$. A language $L \subseteq V^*$ is called a bounded prefix-suffix duplication language if $L = PSD_k^*(x)$ for some $x \in V^*$ and $k > 0$. A prefix-suffix duplication language is defined analogously, see [6]. When duplications of arbitrary factors within the word are permitted, we obtain an (arbitrary) duplication language, see, e.g., [3].

In this paper, we show a series of results of algorithmic nature. All the time complexity bounds we obtain in this context hold for the RAM with logarithmic memory-word size. In the algorithmic problems we approach, we are usually given as input one or more words. These words are assumed to be over an integer alphabet; that is, if w is the input word, and has length n, then we assume that its letters are integers from the set $\{1, \ldots, n\}$. See a discussion about this assumption in [9]. If the input to our problems is a language, then

we assume that this language is specified by a procedure deciding it (e.g., if the language is regular, then we assume that we are given a DFA accepting it).

We recall basic facts about the data structures we use. For a word u, with $|u| = n$, over $V \subseteq \{1, \ldots, n\}$ we can build in linear time a suffix array structure as well as data structures allowing us to retrieve in constant time the length of the longest common prefix of any two suffixes $u[i..n]$ and $u[j..n]$ of u, denoted $LCP(i, j)$. These structures are called LCP data structures in the following. For details, see, e.g., [8,9]. Similarly, one can construct in linear time data structures allowing us to retrieve in constant time the length of the longest common suffix of any two prefixes $u[1..i]$ and $u[1..j]$ of u, denoted $LCS(i, j)$.

We also use a linear data structure, called deque (double-ended queue, see [15]). This is a doubly linked list for which elements can be added to or removed from either the front or back. Finally, tries are complete trees whose edges are labeled with letters of an alphabet V, and ordered according to an (existing) order of the letters of this alphabet; each path of a trie corresponds to a word over V.

3 Bounded Prefix-Suffix Duplication as a Formal Operation on Languages

We start with some language theoretical properties of the class of duplication languages. By combining the results from [1] and [4] (rediscovered in [3] and [14] for arbitrary duplication languages), and [6] we recall the following result.

Theorem 1.
1. An arbitrary duplication language is regular if and only if it is a language over an alphabet with at most two symbols.
2. A prefix-suffix duplication language is context-free if and only if it is a language over the unary alphabet.

Whether or not every arbitrary duplication language is recognizable in polynomial time is open while every prefix-suffix duplication language is in **NL**.

We say that a class $\mathcal{L}$ of languages is closed under bounded prefix-suffix duplication if $PSD_k^*(L) \in \mathcal{L}$ for any $L \in \mathcal{L}$ and $k \geq 1$.

Theorem 2. *Every nonempty class of languages closed under union with regular languages, intersection with regular languages, and substitution with regular languages, is closed under bounded prefix-suffix duplication.*

Proof. Let $\mathcal{L}$ be a family of languages having all the required closure properties. By [7], $\mathcal{L}$ is closed under inverse morphism. Let $L \subseteq V^*$, with $|V| = m$, be a language from $\mathcal{L}$, and k be a positive integer. We define the alphabet

$$U = V \cup \{p_1, p_2, \ldots, p_{m^k}\} \cup \{s_1, s_2, \ldots, s_{m^k}\},$$

and the morphism $h : U^* \longrightarrow V^*$ defined by $h(a) = a$ for any $a \in V$ and $h(p_i) = h(s_i) =$ the i^{th} word of length k over V in the lexicographic order, for all $1 \leq i \leq m^k$. Further, let F be the finite language defined by $F = \{x \in L \mid |x| \leq 2k - 1\}$ and

$$E = (L \cup PSD_k^{2k}(F)) \cap \{x \in V^+ \mid |x| \geq 2k\}.$$

As $PSD_k^{2k}(F)$ is a finite language and $\mathcal{L}$ is closed under union with regular languages and intersection with regular languages, it follows that E is still in $\mathcal{L}$. The following relation is immediate:

$$PSD_k^*(L) = PSD_k^*(E) \cup PSD_k^{2k}(F).$$

It is rather easy to prove that

$$PSD_k^*(E) = \sigma(h^{-1}(E) \cap \{p_1, p_2, \ldots, p_{m^k}\}V^*\{s_1, s_2, \ldots, s_{m^k}\}),$$

where σ is a substitution defined by $\sigma(p_i) = PD_k^*(x_i)$ and $\sigma(s_i) = SD_k^*(x_i)$, where x_i is the i^{th} word of length k over V in the lexicographic order.

Each language $PD_k^*(x_i)$ can be generated by a prefix grammar [5], hence it is regular. Analogously, each language $SD_k^*(x_i)$ is regular. Consequently, σ is a substitution with regular languages. By the closure properties of $\mathcal{L}$, $PSD_k^*(E)$ belongs to $\mathcal{L}$, hence $PSD_k^*(L)$ is also in $\mathcal{L}$. □

Much differently from the statements of Theorem 1 we have:

Corollary 1. *Every bounded prefix-suffix duplication language is regular.*

A language L is said to be a multiple k-prefix-suffix duplication language if there exists a language E such that $L = PSD_k^*(E)$. If E is finite, then L is said to be a finite k-prefix-suffix duplication language. Note that given a regular language L and a positive integer k, a necessary condition such that $L = PSD_k^*(E)$ holds, for some set E, is $L = PSD_k^*(L)$. By Theorem 2 a finite automaton accepting $PSD_k^*(L)$ can effectively be constructed and so the above equality can be algorithmically checked. However, if the equality holds, we cannot infer anything about the finiteness of E. The problem is completely solved by the next theorem.

Theorem 3. *Let L be a regular language which is a multiple k-prefix-suffix duplication language for some positive integer k. There exists a unique minimal (with respect to inclusion) regular language E, which can be algorithmically computed, such that $L = PSD_k^*(E)$. In particular, one can algorithmically decide whether L is a finite k-prefix-suffix duplication language.*

Proof. Let $L \subseteq V^*$ be a multiple k-prefix-suffix duplication language accepted by the deterministic finite automaton $A = (Q, V, f, q, F)$. We define the language

$$M_k(L) = \{x \in L \mid \text{ there is no } y \in L \text{ such that } x \in PSD_k(y)\}.$$

As $L = PSD_k^*(L)$, it follows that

$$M_k(L) = \{x \in L \mid \text{ there is no } y \in L, y \neq x \text{ such that } x \in PSD_k^*(y)\}.$$

Claim. *If $PSD_k^*(E) = L$ for some $E \subseteq L$, then the following statements hold:*
(i) $M_k(L) \subseteq E$, and
(ii) $PSD_k^*(M_k(L)) = L$.

Proof of the claim. (i) Let $x \in M_k(L) \subseteq L$; there exists $y \in E$ such that $x \in PSD_k^*(y)$. By the definition of $M_k(L)$, it follows that $x = y$.

(ii) Clearly, $PSD_k^*(M_k(L)) \subseteq L$. Let $y \in L$; there exists $x \in L$ such that $y \in PSD_k^*(x)$. We may choose x such that $x \in PSD_k(z)$ for no $z \in L$. Thus, $x \in M_k(L)$, and $y \in PSD_k^*(M_k(L))$, which concludes the proof of the claim.

Clearly, $M_k(L) = L \setminus PSD_k(L)$; hence $M_k(L)$ is regular and can effectively be constructed.

In order to check whether L is a finite k-prefix-suffix duplication language we first compute $M_k(L)$. Then we check whether $M_k(L)$ is finite. Finally, by Theorem 2, the language $PSD_k^*(M_k(L))$ is regular and can be effectively computed, therefore the equality $PSD_k^*(M_k(L)) = L$ can be algorithmically checked. □

3.1 Membership Problem

In the sequel, we will make use of the following classical result from [2]. It is known that the number of primitively rooted square factors of length at most $2k$ that occur in a word w at a position is $\mathcal{O}(\log k)$. Moreover, one can construct the list of primitively rooted squares of length at most $2k$ occurring in w in $\mathcal{O}(n \log k)$ time. Each square is represented in the list by the starting position and the length of their root, and the list is ordered increasingly by the starting position of the squares; when more squares share the same starting position they are ordered by the length of the root. Moreover, one can store an array of n pointers, where the i^{th} such pointer gives the memory location of the list of the primitively rooted squares occurring at position i. A similar list, where the squares are ordered by their ending position, can be computed in the same time. Further, we develop our main algorithmic tools.

Lemma 1. *Given $w \in V^*$, of length n, and an integer $k \leq n$, we can identify all prefixes $w[1..i]$ of w such that $w \in SD_k^*(w[1..i])$ in $\mathcal{O}(n \log k)$ time.*

Proof. We propose an algorithm that computes an array $S[\cdot]$, defined by $S[i] = 1$ if $w \in SD_k^*(w[1..i])$, and $S[i] = 0$, otherwise. This algorithm has a preprocessing phase, in which all the primitively rooted squares with root of length at most k occurring in w are computed. This preprocessing takes $\mathcal{O}(n \log k)$ time.

Now, we describe the computation of the array S. Initially, all the positions of this array are initialized to 0, except $S[n]$, which is set to 1. Clearly, this is correct, as $w \in SD_k^*(w[1..n]) = SD_k^*(w)$. Further, we update the values in the array S using a dynamic programming approach. That is, for i from n to 1, if $S[i] = 1$, then we go through all the primitively rooted squares $(w[j+1..i])^2$, $|i - j| \leq k$, that end at position i in w. For each such factor $w[j+1..i]$ we set $S[j] = 1$. Indeed, $w[1..i]$ can be obtained from $w[1..j]$ by appending $w[j+1..i]$ (which is known to be a suffix of $w[1..j]$); as we already know that w can be obtained by suffix duplication from $w[1..i]$, it follows that w can be obtained by suffix duplication from $w[1..j]$. The processing for each i takes $\mathcal{O}(\log k)$ time.

It is not hard to see that our algorithm works correctly. Assume that $w \in SD_k^*(w[1..j])$ for some $j < n$. Let us consider the longest sequence of suffix duplication steps (or, for short, derivation) that produces w starting from $w[1..j]$. Say that this derivation has $s \geq 2$ steps, so it can be described by a sequence of indices $j_1 = j < j_2 < \ldots < j_s = n$ such that $w[1..j_{i+1}] \in SD_k(w[1..j_i]$ for $1 \leq i \leq s-1$. We can show that $w[j_i + 1..j_{i+1}]$ is primitive for all i. Otherwise, $w[j_i + 1..j_{i+1}] = t^\ell$ for some word t and $\ell \geq 2$, so we can replace in the original

derivation the duplication that produces $w[1..j_{i+1}]$ from $w[1..j_i]$ by other ℓ duplication steps in which t factors are added to $w[1..j_i]$. This leads to a sequence with more than s duplications steps producing w from $w[1..j]$, a contradiction. Now, it is immediate that, in our algorithm, $S[j_s]$ is set to 1 in the first step. Assuming that for some i we already have $S[j_{i+1}] = 1$, when considering the value j_{i+1} in the main loop of our algorithm, as $w[j_i + 1..j_{i+1}]^2$ is a primitively rooted square ending on position j_{i+1}, we will set $S[j_i] = 1$. In the end, we will also have $S[j] = S[j_1] = 1$, so our algorithm works properly. □

Lemma 2. *Given $w \in V^*$, of length n, we can identify all suffixes $w[j..n]$ of w such that $w \in PD_k^*(w[j..n])$ in $\mathcal{O}(n \log k)$ time.*

The proof is similar to the one of Lemma 1, and it is left to the reader. The output of the algorithm will be an array $P[\cdot]$, defined by $P[j] = 1$ if $w \in PD_k^*(w[j..n])$, and $P[j] = 0$, otherwise.

The next lemma shows a way to compute the factors of length at least k, from which w can be obtained by iterated prefix or suffix duplication.

Lemma 3. *Given $w \in V^*$ of length n and a list F of factors of w of length greater than or equal to k, given by their starting and ending position, ordered by their starting position, and in case of equality by their ending position, we can check whether there exists $x \in F$ such that $w \in PSD_k^*(x)$ in time $\mathcal{O}(n \log k + |F|)$.*

Proof. The main remark of this lemma is that, if $w[i..j]$ is longer than k, then $w \in PSD_k^*(w[i..j])$ if and only if $w[1..j] \in PD_k^*(w[i..j])$ and $w = w[1..n] \in SD_k^*(w[1..j])$. Equivalently, we have $w \in PSD_k^*(w[i..j])$ if and only if $w[1..n] \in PD_k^*(w[i..n])$ and $w[1..n] \in SD_k^*(w[1..j])$.

This remark suggests the following approach: we first identify all the suffixes $w[j..n]$ of w such that $w \in PD_k^*(w[j..n])$ and all the prefixes $w[1..i]$ of w such that $w \in SD_k^*(w[1..i])$; this takes $\mathcal{O}(n \log k)$, by Lemmas 1 and 2. Now, for every factor $w[i..j]$ in list F, we just check whether $S[i] = P[j] = 1$ (that is, $w \in PD_k^*(w[i..n]) \cap SD_k^*(w[1..j])$); if so, we decide that $w \in PSD_k^*(w[i..j])$. □

Building on the previous lemmas, we can now solve the membership problem for $PSD_k^*(L)$ languages, provided that we know how to solve the membership problem for L on the RAM with logarithmic word size model.

Theorem 4. *If the membership problem for the language L can be decided in $\mathcal{O}(f(n))$ time, then the membership problem for $PSD_k^*(L)$ can be decided in $\mathcal{O}(nk \log k + n^2 f(n))$.*

Proof. Assume that we are given a word w, of length n; we want to test whether $w \in PSD_k^*(L)$ or not. For simplicity, we assume that L is constant (i.e., its description, given as a procedure deciding L in $\mathcal{O}(f(n))$ time, is not part of the input). If L was given as part of the input, then we can use exactly the same algorithm, but one should add to the final time complexity the time needed to read the description of L and effectively construct a procedure deciding L in $\mathcal{O}(f(n))$ time.

First, let us note that we can identify trivially in $\mathcal{O}(n^2 f(n))$ the factors of w that are in L. More precisely, we can produce a list F of factors of w that are contained in L, specified by their starting and ending position, ordered by their starting position, and, in case of equality by their ending position. The list F can be easily split, in $\mathcal{O}(|F|)$ time, into two lists: F_1, containing the factors of length at least k, and F_2, the list of factors of length less than k. It is worth noting that $|F| \in \mathcal{O}(n^2)$. By Lemma 3 it follows that we can decide in time $\mathcal{O}(n \log k + |F_1|)$ whether $w \in PSD_k^*(x)$ for some $x \in F_1$.

It remains to test whether $w \in PSD_k^*(x)$ for some $x \in F_2$. The main remark we make in this case is that there exists $x \in F_2$ such that $w \in PSD_k^*(x)$ if and only if there exists $y \in PSD_k^*(x)$ such that $k \leq |y| \leq 2k$ and $w \in PSD_k^*(y)$. Therefore, we will produce the list F_3 of words $z \in \cup_{x \in F_2} PSD_k^*(x)$ such that z is a factor of w and $k \leq |z| \leq 2k$.

In order to compute F_3 we can use the $\mathcal{O}(|u|^2 \log |u|)$ algorithm proposed in [6] to decide whether a word u is contained in $PSD^*(v)$. In that algorithm, one first marks the factors of u that are equal to v. Further, for each possible length ℓ of the factors of u, from 1 to $|u|$, and for each $i \leq n$ where a factor of length ℓ of u may start, one checks whether $u[i..i+\ell-1]$ can be obtained by prefix (respectively, suffix duplication) from a shorter suffix (respectively, prefix), that was already known (i.e., marked) to be in $PSD^*(v)$, such that in the last step of duplication a primitive root x of a primitively rooted square prefix x^2 of $u[i..i+\ell-1]$ was appended to the shorter suffix (respectively, a primitive root x of a primitively rooted square suffix x^2 of $u[i..i+\ell-1]$ was appended to the shorter prefix). Each time we found a factor of w that can be obtained in this way from one of its marked prefixes or suffixes, we marked it as part of $PSD_k^*(v)$ and continued the search with the next factor of w.

In our case, we can pursue the same strategy: taking w in the role of u, and having already marked the words of F_2 (which are factors of w) just like we did with the occurrences of v, we run the algorithm described above, but only for $\ell \leq 2k$. Note that the primitive roots of primitively rooted square suffixes or prefixes of factors $w[i..i+\ell-1]$ with $\ell \leq 2k$ have length at most k; hence, each duplication that is made towards obtaining such a factor is, in fact, a k-prefix-suffix duplication. In this manner we obtain the factors of w of length at most $2k$ that are from $PSD_k^*(F_2)$. The time needed to obtain these factors is $\mathcal{O}(nk \log k)$. We store this set of factors in F_3 just like before: the factors are specified by their starting and ending position, ordered by their starting position, and, in case of equality by their ending position. The set F_3 may have up to $\mathcal{O}(nk)$ factors, as each of them has length at most $2k$.

By Lemma 3, we can decide in time $\mathcal{O}(n \log k + |F_3|) = \mathcal{O}(nk)$ whether $w \in PSD_k^*(F_3)$. Accordingly, adding the time needed to compute F_3 from F_2, it follows that we can decide in time $\mathcal{O}(nk \log k)$ whether $w \in PSD_k^*(F_2)$. Hence, we can decide whether $w \in PSD_k^*(L)$ in $\mathcal{O}(nk \log k + n^2 f(n))$ time. □

In fact, there are classes of languages for which a better bound than the one in Theorem 4 can be obtained. If L is context-free (respectively, regular) the time needed to decide whether $w \in PSD_k^*(L)$ is $\mathcal{O}(n^3)$ (respectively, $\mathcal{O}(nk \log k + n^2)$),

where $|w| = n$. Indeed, F has always at most n^2 elements, and in the case of context-free (or regular) languages it can be obtained in $\mathcal{O}(n^3)$ time (respectively, $\mathcal{O}(n^2)$) by the Cocke-Younger-Kasami algorithm (respectively, by running a DFA accepting L on all suffixes of w, and storing the factors accepted by the DFA). When L is a singleton, the procedure is even more efficient.

Corollary 2. *Given two words w and x, with $|w| \geq |x|$, we can decide whether $w \in PSD_k^*(x)$ in time $\mathcal{O}(|w|k \log k)$. If $|x| \geq k$, then we can decide whether $w \in PSD_k^*(x)$ in time $\mathcal{O}(|w| \log k)$.*

Proof. Assume that $|w| = n$ and $|x| = m$. First, note that the list F of all occurrences of x in w can be obtained in linear time $\mathcal{O}(n + m)$, using, e.g., the Knuth-Morris-Pratt algorithm [16], and $|F| \in \mathcal{O}(n)$.

For the first part, we follow the same general approach as in Theorem 4. If $|x| < k$, we produce the list of all the factors longer than k, but of length at most $2k$, that can be derived from x. This list is produced in $\mathcal{O}(nk \log k)$ time. Therefore, the total complexity of the algorithm is $\mathcal{O}(nk \log k)$, in this case.

The second result follows now immediately from Lemma 3, as F contains only words of length at least k. □

4 Bounded Prefix-Suffix Duplication Distances

Given two words x, w and $k \geq 1$, the k-prefix-suffix duplication distance between x and w is defined by

$$\delta_k(x, w) = \inf\{\ell \mid x \in PSD_k^\ell(w) \text{ or } w \in PSD_k^\ell(x)\}.$$

By definition, the k-prefix-suffix duplication distance between two words is equal to ∞ if the longer word cannot be derived from the shorter. In a similar fashion, we can define k-suffix duplication distance or k-prefix duplication distance between x and w as the minimum number of k-suffix duplication, respectively, k-prefix duplication steps, needed to transform x into w or w into x.

Theorem 5. *Given $k \geq 1$, let x and w be two words of respective length m and n, $n > m$. If $m \geq k$, then $\delta_k(x, w)$ can be computed in $\mathcal{O}(n \log k)$. If $m < k$, then $\delta_k(x, w)$ can be computed in $\mathcal{O}(nk \log k)$.*

The k-prefix-suffix duplication distance between two words can be extended to the k-prefix-suffix duplication distance between a word x and a language L by $\delta_k(x, L) = \min\{\delta_k(x, y) \mid y \in L\}$. Moreover, one can canonically define the distance between languages: for two languages L_1, L_2 and a positive integer k, we set $\delta_k(L_1, L_2) = \min\{\delta_k(x, y) \mid x \in L_1, y \in L_2\}$.

Theorem 6. *Given two regular languages L_1 and L_2 over an alphabet V, recognised by deterministic finite automata with sets of states Q and S, respectively, and a positive integer $k \geq 1$, one can algorithmically compute $\delta_k(L_1, L_2)$ in $\mathcal{O}((k + N)M^2|V|^{2k})$, where $M = \max\{|Q|, |S|\}$ and $N = \min\{|Q|, |S|\}$.*

Proof. Let us assume that both L_1 and L_2 are given by the minimal deterministic finite automata accepting them, namely A_1 and, respectively, A_2. Let $A_1 = (Q, V, \delta', q_0, Q_f)$ and $A_2 = (S, V, \delta'', s_0, S_f)$. As a rule, we denote the states of Q and S by q and s, respectively, with or without indices.

Before starting the main proof, let us briefly explain a series of implementation details.We work with 5-tuples (q, s_1, s_2, w_1, w_2) and 4-tuples (s_1, s_2, w_1, w_2), where $w_1, w_2 \in V^*, |w_1| = |w_2| \leq k$; moreover, whenever $|w_1| < k$ then $w_1 = w_2$.

A set T of 5-tuples as above is implemented as a 3-dimensional array M_T, where $M_T[q][s_1][s_2]$ contains a representation of the set $\{(w_1, w_2) \in V^* \times V^* \mid (q, s_1, s_2, w_1, w_2) \in T\}$ which is implemented using a trie data structure essentially storing all possible words of length k, augmented with suffix links. Using this representation we can check in constant time whether or not a certain pair of words (given as pair of nodes of the trie we construct) is in the set. The same strategy may be used for implementing a set R of 4-tuples.

For a word $w \in V^*$, we denote by $pref_k(w)$ the longest prefix of length at most k of w; similarly, let $suf_k(w)$ be the longest suffix of length at most k of w.

The algorithm that computes $\delta_k(L_1, L_2)$ has two similar main parts. In the first one, we compute the minimum value d_1 such that there exists a word $x \in L_2$ with $x \in PSD_k^{d_1}(L_1)$. In the second part, we compute, using exactly the same procedure, the minimum value d_2 such that there exists a word $y \in L_1$ with $y \in PSD_k^{d_2}(L_2)$. Then, we conclude that $\delta_k(L_1, L_2) = \min\{d_1, d_2\}$. Hence, it suffices to describe how the minimum value d_1 such that there exists a word $x \in L_2$ with $x \in PSD_k^{d_1}(L_1)$ is computed.

As a preprocessing phase of our algorithm, we compute in $\mathcal{O}(k|Q|^2|V|^k)$ time (in a naive manner), for each $q_1 \in Q$ and $w \in V^{\leq k}$ all states q_2 such that $\delta'(q_2, w) = q_1$ and the state $q_3 = \delta(q_1, w)$. Provided that we use the same idea of storing words as labels of nodes from the trie (the label of w being denoted $\#(w)$), we can store this information in space $\mathcal{O}(|Q|^2|V|^k)$, so that we can obtain in constant time, for q_1 and $\#(w)$, the states q_2 and q_3 defined above. We then process the automaton A_2 in a similar manner, in time $\mathcal{O}(|S|^2|V|^k)$.

We present now the main part of our algorithm. First, we compute the set

$$R_0 = \{(s_1, s_2, w_1, w_2) \mid \text{ there exists } w \in L_1 \text{ such that } \delta''(s_1, w) = s_2, \\ pref_k(w) = w_1, suf_k(w) = w_2\}.$$

This computation is done as follows. We compute iteratively the sets T_s^i, $i \geq 1$, each one containing the tuples (q, s, s_1, w_1, w_2) for which there exists a word w of length i, with $pref_k(w) = w_1$, $suf_k(w) = w_2$, $\delta'(q_0, w) = q$ and $\delta''(s, w) = s_1$, but there exists no word w' shorter than w with the same properties. Clearly, in such a 5-tuple, $|w_1| = |w_2|$ and if $|w_1| < k$ then $w_1 = w_2$. We can implement the union (over all values of i) of the sets T_s^i by marking in a trie storing all the words of length k over V the nodes corresponding to the words of this set. The sets T_s^i are computed as long as they are non-empty; clearly, if T_s^i is empty, then the sets T_s^j are empty, for all $j \geq i$. On the other hand, as the number of all the tuples (q, s, s_1, w_1, w_2) as above is upper bounded by $2|Q||S||V|^{2k}$, there exists i_0 such that $T_s^i = \emptyset$ when $i \geq i_0$ and $T_s^{i_0-1} \neq \emptyset$. It is not hard to see that T_s^{i+1} can be computed in time $\mathcal{O}(k|T_s^i|)$, given the elements of T_s^i.

Indeed, for each 5-tuple $(q, s, s_1, w_1, w_2) \in T_s^i$ and letter $a \in V$, we compute the 5-tuple $(\delta'(q,a), s, \delta''(s_1,a), pref_k(w_1a), suf_k(w_2a))$; note that the nodes of the trie corresponding to the words $pref_k(w_1a)$ and $suf_k(w_2a)$ can be obtained in $\mathcal{O}(1)$ time, by knowing the nodes corresponding to w_1 and w_2 and using their suffix links. Then, if the new tuple does not belong to $\bigcup_{i=1}^{i} T_s^i$, we add it to T_s^{i+1}; by maintaining another trie-structure for $\bigcup_{i=1}^{i} T_s^i$, we obtain that checking whether an element is in this set or adding an element to it is done in $\mathcal{O}(1)$ time. To efficiently go through the elements of T_s^i, we store them in a linked list.

We now set $\hat{T}_s = \bigcup_{i=1}^{i_0} T_s^i$. It follows that $\hat{T}_s$ is computed in $\mathcal{O}(|Q||S||V|^{2k})$ time. Therefore, $R_0 = \{(s_1, s_2, w_1, w_2) \mid (q, s_1, s_2, w_1, w_2) \in \bigcup_{s \in S} \hat{T}_s, q \in Q_f\}$. Clearly, it takes $\mathcal{O}(|Q||S|^2|V|^{2k})$ time to compute R_0. We now set $\hat{R}_j = \bigcup_{i=0}^{j} R_i$ and iteratively compute the sets R_j, $j = 1, 2, \ldots$ as follows:

- $R_{j+1} = (R_{j+1}^1 \cup R_{j+1}^2) \setminus \hat{R}_j$,
- $R_{j+1}^1 = \{(s_1, s', w_1', w_2') \mid$ there exist $(s_1, s_2, w_1, w_2) \in R_j$, and $w' \in V^*$ a suffix of w_2, such that $\delta''(s_2, w') = s', pref_k(w_1w') = w_1', suf_k(w_2w') = w_2'\}$,
- $R_{j+1}^2 = \{(s', s_2, w_1', w_2') \mid$ there exist $(s_1, s_2, w_1, w_2) \in R_j$, and $w' \in V^*$ a prefix of w_1, such that $\delta''(s', w') = s_1, pref_k(w'w_1) = w_1', suf_k(w'w_2) = w_2'\}$.

Actually, $(s_1, s_2, w_1, w_2) \in R_j$ if and only if there exists a word w which can be obtained by applying j times the k-prefix-suffix duplication to a word from L_1 such that $pref_k(w) = w_1$, $suf_k(w) = w_2$, and $\delta''(s_1, w) = s_2$; furthermore, there is no word w' that fulfils the same conditions and can be obtained by applying less than j times the k-prefix-suffix duplication to the words of L_1. Clearly, all the elements of these sets fulfil the conditions allowing us to use again a trie implementation for the union of the sets. Using this implementation, and additionally storing each R_j as a list, the time needed to compute the set R_{j+1} is upper bounded by $\mathcal{O}(k|R_j|)$. Indeed, first we construct R_{j+1}^2: for each tuple $(s_1, s_2, w_1, w_2) \in R_j$ and prefix x of w_1, we use the precomputed data structures to obtain the state s such that $\delta'(s, x) = s_1$ and decide that $(s, s_2, pref_k(xw_1), suf_k(xw_2))$ should be added to R_{j+1} (but only if it is not already in other $R_{j'}$ with $j' < j+1$). To implement this efficiently, we consider the prefixes of x in increasing order with respect to the length, and so we will get the node corresponding to xa in the trie in $\mathcal{O}(1)$ time when we know the node corresponding to x. Then we construct R_{j+1}^1: for each tuple (s_1, s_2, w_1, w_2) and for each suffix x of w_2, we use the precomputed data structures to obtain $s = \delta'(s_2, x)$ and decide that $(s_1, s, pref_k(w_1x), suf_k(w_2x))$ should be added to R_{j+1} (again, only if it is not in other $R_{j'}$ with $j' < j+1$). This time we consider the suffixes x of w_2 in decreasing order with respect to their length; in this way, we get the node corresponding to x from the node corresponding to ax in $\mathcal{O}(1)$ time using the suffix links. The sets R_j are computed until either one meets a value j_0 such that $(s_0, s, w_1, w_2) \in R_{j_0}$ for some $s \in S_f$ and $w_1, w_2 \in V^{\leq k}$, or $R_j = \emptyset$. As the number of all 4-tuples that may appear in all the sets R_j is bounded by $\mathcal{O}(|S|^2|V|^{2k})$, the computation of the sets R_j ends after at most $\mathcal{O}(k|S|^2|V|^{2k})$ steps. It is clear that if the process of computing the sets R_j

ends by reaching the value j_0 mentioned above, then we conclude that $d_1 = j_0$. Otherwise, $d_1 = \infty$ holds. The correctness of the computation of d_1 follows immediately from the discussions above.

Consequently, the total time needed to compute d_1 is $\mathcal{O}(|V|^k + k|Q|^2|V|^k + 2k|S|^2|V|^{2k} + |Q||S|^2|V|^{2k}) = \mathcal{O}(k|Q|^2|V|^k + |Q||S|^2|V|^{2k})$. We can use the same procedure to compute d_2, just by changing the roles of L_1 and L_2. Then, we return as $\delta_k(L_1, L_2) = \min\{d_1, d_2\}$. The time needed to compute this distance is $\mathcal{O}((k+N)M^2|V|^{2k})$, where $M = \max\{|Q|, |S|\}$ and $N = \min\{|Q|, |S|\}$. □

Note that if V is a constant size alphabet, then the previous result provides a cubic algorithm computing the distance between two regular languages. The following corollary follows from Theorem 6, for $L_1 = \{x\}$ and $L_2 = L$.

Corollary 3. *Given a word x, a regular language L accepted by a DFA with q states, and a positive integer $k \geq 1$, one can algorithmically compute $\delta_k(x, L)$ in $\mathcal{O}((k + |N|)|M|^2|V|^{2k})$ time, where $M = \max\{q, |x|\}$ and $N = \min\{q, |x|\}$.*

References

1. Bovet, D.P., Varricchio, S.: On the regularity of languages on a binary alphabet generated by copying systems. Inform. Proc. Letters 44(3), 119–123 (1992)
2. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Inf. Process. Lett. 12(5), 244–250 (1981)
3. Dassow, J., Mitrana, V., Păun, G.: On the regularity of duplication closure. Bull. European Assoc. Theor. Comput. Sci. 68, 133–136 (1999)
4. Ehrenfeucht, A., Rozenberg, G.: On the separating power of EOL systems. RAIRO Inform. Theor. 17(1), 13–22 (1983)
5. Frazier, M., David Page Jr., C.: Prefix grammars: an alternative characterization of the regular languages. Inf. Process. Lett. 2, 67–71 (1994)
6. Garcia Lopez, J., Manea, F., Mitrana, V.: Prefix-suffix duplication. J. Comput. Syst. Sci. (in press) doi:10.1016/j.jcss.2014.02.011
7. Ginsburg, S.: Algebraic and automata-theoretic properties of formal languages. North-Holland Pub. Co. (1975)
8. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York (1997)
9. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
10. Leupold, P.: Reducing repetitions. In: Prague Stringology Conf., pp. 225–236 (2009)
11. Papadimitriou, C.: Computational Complexity. Addison-Wesley (1994)
12. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, vol. I–III. Springer, Berlin (1997)
13. Searls, D.B.: The computational linguistics of biological sequences. In: Artificial Intelligence and Molecular Biology, pp. 47–120. MIT Press, Cambridge (1993)
14. Wang, M.-W.: On the irregularity of the duplication closure. Bull. European Assoc. Theor. Comput. Sci. 70, 162–163 (2000)
15. Knuth, D.: The Art of Computer Programming, 3rd edn. Fundamental Algorithms, vol. 1, pp. 238–243. Addison-Wesley (1997), Section 2.2.1: Stacks, Queues, and Deques, ISBN 0-201-89683-4
16. Knuth, D., Morris, J.H., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)