Hairpin Lengthening and Shortening of Regular Languages*

Florin Manea[1] **Robert Mercas**[2] **Victor Mitrana**[3]

[1]Faculty of Mathematics and Computer Science,
University of Bucharest,
Str. Academiei 14, RO-010014 Bucharest, Romania,
E-mail: flmanea@fmi.unibuc.ro

[2]Facultät für Informatik,
Otto-von-Guericke-Universität Magdeburg,
PSF 4120, D-39016 Magdeburg, Germany
E-mail: robertmercas@gmail.com

[3]Department of Organization and Structure of Information,
University School of Informatics,
Polytechnic University of Madrid,
Crta. de Valencia km. 7 - 28031 Madrid, Spain.
E-mail: victor.mitrana@upm.es

Abstract. We consider here two formal operations on words inspired by the DNA biochemistry: hairpin lengthening introduced in [10] and its inverse called hairpin shortening. We study the closure of the class of regular languages under the non-iterated and iterated variants of the two operations. The main results are: although any finite number of applications of the hairpin lengthening to a regular language may lead to non-regular languages, the iterated hairpin lengthening of a regular language is always regular. As far as the hairpin shortening operation is concerned, the class of regular languages is closed under bounded and unbounded iterated hairpin shortening.

1 Introduction

This paper is a continuation of a series of works started with [3] (based on some ideas from [1]), where a new, bio-inspired, formal operation on words, called hairpin completion, was introduced. The initial work was followed by a series of related papers ([4, 5, 7, 9, 11, 12, 8]), where both the hairpin completion, as well as its inverse operation, the hairpin reduction, were further investigated both from the algorithmic and the language theoretic points of view.

We briefly recall the biological motivation of this operation. Polymerase chain reaction (PCR) is an automated process which enables researchers to produce a huge number of copies of a specific DNA sequence. Although PCR starts with

* Work supported by the Alexander von Humboldt Foundation

a test tube containing double-stranded DNA molecules (called template and primer), we consider here a pretty similar phenomenon involving single-stranded DNA (ssDNA) following the second and third step of PCR, namely annealing and extension, respectively. It is well known that ssDNA are composed by nucleotides which differ from each other by their bases: A (adenine), G (guanine), C (cytosine), and T (thymine). Two single strands can bind to each other, forming the secondary structure of DNA, if they are pairwise *Watson-Crick complementary*: A is complementary to T, and C to G. The binding of two strands is usually called annealing. This process which appears at about 54^0C is due to the Hydrogen-bonds that are constantly formed and broken between the two ssDNA. More bonds last longer and allow the polymerase to attach new nucleotides [**?**]. Once a few bases are built in, the ionic bond between the two ssDNA becomes so strong, that it does not break anymore. The polymerase attaches the bases (complementary to the template) to the primer on the 3' side.

We now imagine the following situation in which the role of the two ssDNA (template and primer) in the PCR is played by only one ssDNA. An intramolecular base pairing, known as *hairpin*, is a pattern that can occur in single-stranded DNA or RNA molecules. In this case, the single-stranded molecule bends, and one part of the strand bonds to another part of the same strand. In this way, the role of template and primer is played by the prefix and suffix, or vice-versa, of the ssDNA. In our case the phenomenon produces a new molecule as follows (see Figure 1): one starts with a ssDNA molecule, such that one of its ends (a prefix or, respectively, a suffix) is annealed to another part of itself by Watson-Crick complementarity forming a hairpin, and a *polymerization buffer* with many copies of the four basic nucleotides. Then, the initial hairpin is lengthened by polymerases (thus adding a suffix or, respectively, a prefix), until a complete hairpin structure is obtained (the beginning of the strand is annealed to the end of the strand). Of course, all these phenomena are considered here in an idealized way. For instance, we allow polymerase to extend the strand at either end (usually denoted in biology with 3' and 5') despite that, due to the greater stability of 3' when attaching new nucleotides, DNA polymerase can act continuously only in the 5'$\longrightarrow$ 3' direction. However, polymerase can also act in the opposite direction, but in short "spurts" (Okazaki fragments). This is the source of inspiration for the hairpin completion operation introduced in [3]. The situation is schematically illustrated in Picture 1.

Hairpin or hairpin-free structures have numerous applications to molecular genetics and DNA-computing. In some DNA-based algorithms, these DNA molecules cannot be used in the subsequent computations. Therefore, it is important to design methods for constructing sets of DNA sequences which are unlikely to lead to such "bad" hybridizations. This problem was considered in a series of papers, see e.g. [**?**,**?**,**?**] and the references therein. On the other hand, molecules which may form a hairpin structure have been used as the basic feature of a computational model reported in [**?**], where an instance of the 3-SAT problem has been solved by a DNA algorithm whose second phase is mainly based on the elimination of hairpin structured molecules. Different types of hairpin and

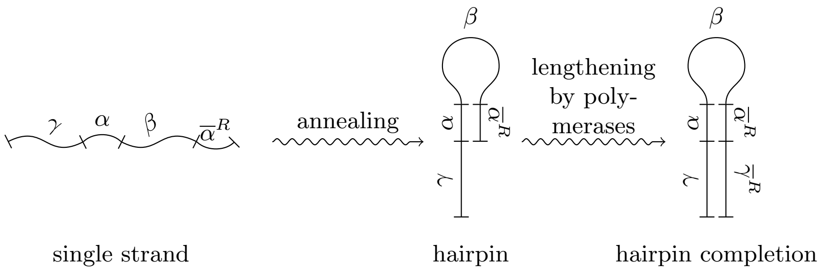

Fig. 1. Hairpin Completion

hairpin-free languages are defined in [2, 13] and more recently in [**?**,6], where they are studied from a language theoretical point of view.

In [10] and later on in [**?**], a new variant of the hairpin completion, called hairpin lengthening, which seems more appropriate for possible bio-lab implementation, is considered. Informally, it seems more natural to consider that the prefix/suffix added by the hairpin completion cannot be arbitrarily long, since every step of a computation in a laboratory has to use a finite amount of resources and finite time. This variant concerns the prolongation of a strand which forms a hairpin, similarly to the process described for hairpin completion, but not necessarily until a complete hairpin structure is obtained. The main motivation in introducing this operation is that, in practice, it may be a difficult task to control the completion of a hairpin structure, and it seems easier to model only the case when such a structure is extended.

Both the hairpin completion and hairpin lengthening can be seen as formal operations by which one can generate a set of words, starting from a single word: for each possible pairing between a prefix and a complementary factor, or a suffix and a complementary factor, we can obtain a word by hairpin completion and several words by hairpin lengthening. As most of the unary operations on words, the hairpin completion and lengthening defined above can be extended canonically to operations on languages, and, then, their iterated version can be defined. We consider here two formal operations on words inspired by the DNA biochemistry: hairpin lengthening discussed above and a new operation which is its inverse, namely hairpin shortening. We study the closure of the class of regular languages under the non-iterated and iterated variants of the two operations.

In this paper we show that although any finite number of applications of the hairpin lengthening to a regular language may lead to non-regular languages, the iterated hairpin lengthening preserves the regularity of a language. As far as the hairpin shortening operation is concerned, we prove that the class of regular languages is closed both under finitely-iterated and freely iterated hairpin shortening.

2 Preliminaries

We assume the reader to be familiar with the fundamental concepts of formal languages and automata theory, particularly with the notions of regular languages and finite automata; for all the related notions see [14].

We denote by V^* and V^+ the set of all words over V including the empty word ε, and the set of all non-empty words over V, respectively. Given a word w over an alphabet V, we denote by $|w|$ its length, while $w[i..j]$ denotes the factor of w starting at position i and ending at position j, $1 \le i \le j \le |w|$. If $i = j$, then $w[i..j]$ is the i-th letter of w, which is simply denoted by $w[i]$.

Let Ω be a "superalphabet", that is an infinite set such that any alphabet considered in this paper is a subset of Ω. In other words, Ω is the *universe* of the languages in this paper, i.e., all words and languages are over alphabets that are subsets of Ω. An *involution* over a set S is a bijective mapping $\sigma : S \longrightarrow S$ such that $\sigma = \sigma^{-1}$. Any involution σ on Ω such that $\sigma(a) \neq a$ for all $a \in \Omega$ is said to be, in this paper's context, a *Watson-Crick involution*. Despite that this is nothing more than a fixed point-free involution, we prefer this terminology since the hairpin lengthening defined later is inspired by the DNA lengthening by polymerases, where the Watson-Crick complementarity plays an important role. Let $\bar{\cdot}$ be a Watson-Crick involution fixed for the rest of the paper. The Watson-Crick involution is extended to a morphism from Ω^* to Ω^* in the usual way. We say that the letters a and $\overline{a}$ are complementary to each other. For an alphabet V, we set $\overline{V} = \{\overline{a} \mid a \in V\}$. Note that V and $\overline{V}$ could be disjoint or intersect or be equal. We denote by $(\cdot)^R$ the mapping defined by ${}^R : V^* \longrightarrow V^*$, $(a_1a_2\ldots a_n)^R = a_n \ldots a_2a_1$. Note that ^{R} is an involution and an *anti-morphism* ($(xy)^R = y^Rx^R$ for all $x, y \in V^*$). Note also that the two mappings $\bar{\cdot}$ and $\cdot^R$ commute, namely, for any word x the equality $(\overline{x})^R = \overline{x^R}$ holds.

Let V be an alphabet, for any $w \in V^+$ we define the *k-hairpin lengthening* of w, denoted by $HL_k(w)$, for some $k \ge 1$, as follows:

- $HLP_k(w) = \{\overline{\delta^R}w | w = \alpha\beta\overline{\alpha^R}\gamma, |\alpha| = k,\ \alpha, \beta, \gamma \in V^+$ and δ is a prefix of $\gamma\}$,
- $HLS_k(w) = \{w\overline{\delta^R} | w = \gamma\alpha\beta\overline{\alpha^R}, |\alpha| = k,\ \alpha, \beta, \gamma \in V^+$ and δ is a suffix of $\gamma\}$,
- $HL_k(w) = HLP_k(w) \cup HLS_k(w)$.

The *hairpin lengthening* of w is defined by $HL(w) = \bigcup\limits_{k \ge 1} HL_k(w)$. Clearly, $HL_{k+1}(w) \subseteq HL_k(w)$ for any $w \in V^+$ and $k \ge 1$. Therefore, one can easily note that $HL(w) = HL_1(w)$.

The k-hairpin lengthening is naturally extended to languages by $HL_k(L) = \bigcup\limits_{w \in L} HL_k(w)$ for $k \ge 1$.

This operation is schematically illustrated in Figure 2.

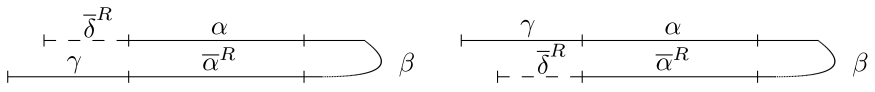

Figure 2: Hairpin lengthening

The iterated version of the k-hairpin lengthening is defined as usual by:

$$HL_k^0(w) = \{w\},\ HL_k^{n+1}(w) = HL_k(HL_k^n(w)),\ HL_k^*(w) = \bigcup_{n\geq 0} HL_k^n(w),$$

and $HL_k^*(L) = \bigcup_{w\in L} HL_k^*(w)$.

The iterated version of the hairpin lengthening is $HL^*(L) = \bigcup_{w\in L} HL_1^*(w)$.

Let us define the inversion operation of the hairpin lengthening, namely the hairpin shortening. Let V be an alphabet, and define for any $w \in V^+$ the k-*hairpin shortening* of w, denoted by $HS_k(w)$, for some $k \geq 1$, as follows:

- $HSP_k(w) = \{x|$ exists δ such that $\overline{\delta^R}x = \overline{\delta^R}\alpha\beta\overline{\alpha^R}\gamma = w, |\alpha| = k,\ \alpha, \beta, \gamma \in V^+$ and δ is a prefix of $\gamma\}$,
- $HSS_k(w) = \{x|$ exists δ such that $x\overline{\delta^R} = \gamma\alpha\beta\overline{\alpha^R\delta^R} = w, |\alpha| = k,\ \alpha, \beta, \gamma \in V^+$ and δ is a suffix of $\gamma\}$,
- $HS_k(w) = HSP_k(w) \cup HSS_k(w)$.

The *hairpin shortening* of w is defined by $HS(w) = \bigcup_{k\geq 1} HS_k(w)$. The k-hairpin shortening is naturally extended to languages by $HS_k(L) = \bigcup_{w\in L} HS_k(w)$ for $k \geq 1$.

The iterated version of the k-hairpin shortening is defined as usual by:

$$HS_k^0(w) = \{w\},\ HS_k^{n+1}(w) = HS_k(HS_k^n(w)),\ HS_k^*(w) = \bigcup_{n\geq 0} HS_k^n(w),$$

and $HS_k^*(L) = \bigcup_{w\in L} HS_k^*(w)$.

The iterated version of the hairpin lengthening is $HS^*(L) = \bigcup_{w\in L} HS_1^*(w)$.

3 Hairpin Lengthening of Regular Languages

The following result was reported in [**?**]:

Theorem 1 *A language is linear context-free if and only if it is the gsm image of the k-hairpin lengthening of a regular language for $k \geq 1$.*

Consequently,

Proposition 1 *The class of regular languages is not closed under k-hairpin lengthening for $k \geq 1$.*

A first natural question concerns the closure of the class of regular languages under the application for a finite number of times of the k-hairpin lengthening operation. In [**?**], we show that if we apply for a finite number of times the k-hairpin lengthening operation to a regular language we can obtain non-regular languages. The regular language $L = \{\diamond a^n b^k c\overline{b^k} \mid n \geq k\}$ proves this. Indeed, it is not hard to see that, for $m \geq 1$, we have

$$\begin{aligned} HL_k^m(L) = \{&\diamond a^n b^k c\overline{b^k a^s} \mid n \geq k, m \leq s \leq n + (m-1)(n-k)\} \cup \\ &\{\diamond a^n b^k c\overline{b^k a^s}\bar{\diamond} \mid n \geq 1, k \leq s \leq n + (m-1)(n-k)\}. \end{aligned}$$

Thus, $HL_k^m(L) \cap \diamond V^* \bar{\diamond} = \{\diamond a^n b^k c\overline{b^k a^s}\bar{\diamond} \mid n \geq 1, k \leq s \leq (m-1)(n-k)\}$ and one can easily show that this language is not regular.

We now came to the second natural question: Is the class of regular languages closed under iterated k-hairpin lengthening? Somehow surprising (in the view of the above discussion), the answer is affirmative.

First, let us note that every iterated k-hairpin lengthening can be simulated by an iterated k-hairpin lengthening, where only one symbols is added in the right-hand or left-hand end of the current word. Let V be an alphabet and $k \geq 1$. For each letter $a \in V$ we define the two languages:

$$\begin{aligned} R(k,a) &= \{x\overline{a}yz\overline{y}^R \mid x \in V^*, y, z \in V^+, |y| = k\}, \\ L(k,a) &= \{yz\overline{y}^R\overline{a}x \mid x \in V^*, y, z \in V^+, |y| = k\}. \end{aligned}$$

Obviously, the two languages are regular.

We now define the mapping $\phi_k : V^* \longrightarrow V^*$ by:

$$\phi_k(w) = \{w\} \cup \{wa \mid w \in R(k,a), a \in V\} \cup \{aw \mid w \in L(k,a), a \in V\}.$$

This mapping is naturally extended to languages by $\phi_k(L) = \bigcup\limits_{w \in L} \phi_k(w)$.

The iterated version of this mapping is defined as usual by:

$$\phi_k^0(w) = \{w\}, \qquad \phi_k^{n+1}(w) = \phi_k(\phi_k^n(w)), \qquad \phi_k^*(w) = \bigcup_{n \geq 0} \phi_k^n(w),$$

and $\phi_k^*(L) = \bigcup\limits_{w \in L} \phi_k^*(w)$.

It is plain that for any language L and any $k \geq 1$, $HL_k^*(L) = \phi_k^*(L)$. The main result of this section is based on this simple remark.

Theorem 2 *The class of regular languages is closed under iterated k-hairpin lengthening for $k \geq 1$.*

Proof. We begin our proof with a simple remark. Let w and w' be two words over an alphabet V such that $w \in HL_k^*(w')$. According to the remarks made before this proof, there exist the words $w_0, w_1, \ldots, w_t$ such that $w_0 = w'$, $w_t = w$, $w_i \in HL_k(w_{i-1})$ and $|w_i| = |w_{i-1}| + 1$ for $1 \leq i \leq t$. Then, for $1 \leq i \leq t$:

- If $w_i \in HLS_k(w_{i-1})$ has the suffix αa with $a \in V$ and $\alpha \in V^k$, and w ends with α, then $wa \in HL_k^*(w')$.

- If $w_i \in HLP_k(w_{i-1})$ has the prefix $a\alpha$ with $a \in V$ and $\alpha \in V^k$, and w starts with α, then $aw \in HL_k^*(w')$.

Assume now that we want to determine whether a word w of length n can be obtained by iterated k-hairpin lengthening from one of its factors $w[i..j]$. More precisely, we determine whether w can be obtained from $w[i..j]$ by iterated k-hairpin lengthening such that in each step the current word is lengthened with exactly one symbol (which is equivalent to the general iterated k-hairpin lengthening, as we have already seen before in this proof).

Let $S_1 = (x_1, x_2, \ldots, x_t)$ be a list of the distinct words of length $k+1$ that occur as factors of the word $w[1..i+k-1]$, ordered increasingly according to the rightmost position where they appear in $w[1..i+k-1]$ (i.e., the rightmost occurrence of x_ℓ in $w[1..i+k-1]$ is to the left of the rightmost position where $x_{\ell+1}$ occurs in the same factor). Let p_ℓ be the maximum number such that $p_\ell \leq i-1$ and $w[p_\ell..p_\ell+k] = x_\ell$, i.e., p_ℓ is the starting position of the rightmost occurrence of x_ℓ in $w[1..i+k-1]$; clearly, $p_t = i-1$. Finally, we define the list of numbers $S_1' = (n_1, \ldots, n_t)$ by $n_1 = \min(k, p_1 - 1)$ and $n_\ell = \min(k, p_\ell - p_{\ell-1} - 1)$ for $2 \leq \ell \leq t$.

It is not hard to see that all the factors $w[i'..i'+k]$ with $p_{\ell-1}+1 \leq i' \leq i-1$ are from the set $\{x_\ell, x_{\ell+1}, \ldots, x_t\}$. By the remark stated at the beginning of this proof, it follows that if $j' \geq j$ and $w[p_\ell..j']$ can be obtained by iterated k-hairpin lengthening from $w[i..j]$, then $w[p_{\ell-1}+1..j']$ can be also obtained by iterated k-hairpin lengthening from $w[i..j]$. Moreover, $w[p_\ell..j']$ can be obtained from $w[p_\ell+1..j]$ by k-hairpin lengthening if and only if $\overline{x}_\ell{}^R$ appears as a factor in $w[p_\ell+k+1..j']$. Finally, $w[1..j']$ can be obtained from $w[p_1+1..j]$ by k-hairpin lengthening if and only if $\overline{x}_1{}^R$ appears as a factor in $w[p_1+k+1..j']$.

Analogously, let $S_2 = (y_1, y_2, \ldots, y_s)$ be a list of all the different words of length $k+1$ that occur as factors of the word $w[j-k+1..n]$, ordered increasingly according to the leftmost position where they appear in $w[j-k+1..n]$ (i.e., the leftmost occurrence of y_ℓ in $w[j-k+1..n]$ is to the left of the leftmost position where $y_{\ell+1}$ occurs in the same factor). Let q_ℓ be the minimum number such that $q_\ell \geq j+1$ and $w[q_\ell - k..q_\ell] = y_\ell$, i.e., q_ℓ is the ending position of the leftmost occurrence of y_ℓ in $w[j-k+1..n]$; clearly, $q_1 = j+1$. We also define the list of numbers $S_2' = (m_1, \ldots, m_s)$ by $m_s = \min(k, n - q_s)$ and $m_\ell = \min(k, q_{\ell+1} - q_\ell + 1)$ for $1 \leq \ell \leq s-1$.

It is immediate that all the factors $w[i'-k..i']$ with $j+1 \leq i' \leq q_{\ell+1}-1$ are from the set $\{y_1, y_2, \ldots, y_\ell\}$. Thus, if $i' \leq i$ and $w[i'..q_\ell]$ can be obtained by iterated k-hairpin lengthening from $w[i..j]$, then $w[i'..q_{\ell+1}-1]$ can be also obtained by iterated k-hairpin lengthening from $w[i..j]$. Moreover, $w[i'..q_\ell]$ can be obtained from $w[i'..q_\ell - 1]$ by k-hairpin lengthening if and only if $\overline{y}_\ell{}^R$ appears as a factor in $w[i'..q_\ell - k - 1]$. Also, $w[i'..n]$ can be obtained from $w[i'..q_s - 1]$ by k-hairpin lengthening if and only if $\overline{y}_s{}^R$ appears as a factor in $w[i'..q_s - k - 1]$.

Further, assume that S is a list of the distinct factors of length $k+1$ of $w[i..j-k]$ and S' a list of the distinct factors of length $k+1$ of $w[i+k..j]$; also, let T and T' be two lists containing initially k times the word $\perp^{k+1}$, where $\perp \notin V$. It follows that $w[p_{t-1}+1..j]$ can be obtained from $w[i..j]$ if and only if

$\overline{x}_t^R$ appears in S'. Similarly, $w[i..q_2 - 1]$ can be obtained from $w[i..j]$ if and only if $\overline{y}_1^R$ appears in S. Now we are in the position of extending these words even more, in order to see if the whole word w can be generated. We update the lists $S_1, S_1', S_2, S_2', S, S'$ as well as T and T' as follows:

- If $w[p_{t-1} + 1..j]$ can be obtained from $w[i..j]$, we delete from S_1 the element x_t and from S_1' the element n_t. If $n_t = k$ we add x_t to S' and set T' to be the list with k elements $\perp^{k+1}$; otherwise, if $n_t = \ell < k$, we update the list T' as follows: we delete the last $\ell + 1$ elements of T and put in S' those that do not contain $\perp$, and add at the beginning of T' the word x_t and, then, ℓ times the word $\perp^{k+1}$.
- If $w[i..q_1 - 1]$ can be obtained from $w[i..j]$, we delete from S_2 the element y_1 and from S_2' the element m_1. If $m_1 = k$ we add y_1 to S' and set T to be the list with k elements $\perp^{k+1}$; otherwise, if $m_1 = \ell < k$, we update the list T as follows: we delete the last $\ell + 1$ elements of T and put in S those that do not contain $\perp$, and add at the beginning of T the word y_1 and, then, ℓ times the word $\perp^{k+1}$.

At this point we can repeat the process just described above and try to extend the newly obtained word ($w[p_{t-1} + 1..j]$ or $w[i..q_1 - 1]$) even more, to the left or to the right. We stop iterating this process either when both S_1 and S_2 are empty or when the current word cannot be extended (to the left or to the right) anymore. In the first case, when the two lists become empty, we decide that w can be obtained from $w[i..j]$ by iterated k-hairpin lengthening, while in the second case we decide that w can not be obtained from $w[i..j]$ by iterated k-hairpin lengthening. It is worth noting that our decision is taken by looking only at the lists S_1, S_1', S_2, S_2', S and S', and the total number of possible assignments for these lists depends only on k, not on w or on the values of i and j. Therefore, we call a 6-tuples of lists $(S_1, S_1', S_2, S_2', S, S')$ acceptable if the process described above ends with S_1 and S_2 empty; otherwise, they are called unacceptable. One clearly needs a constant amount of time (depending on k) to decide which 6-tuples of lists are acceptable, and which are unacceptable; this can be done by a preprocessing phase before designing an algorithm (or, equivalently, an automaton, as we will see in the end of this proof) recognizing $HL_k^*(L)$.

According to these remarks we present a non-deterministic algorithm that accepts the iterated k-hairpin lengthening of a regular language L over V. The main idea is quite simple: we choose non-deterministically a factor of the input word, and compute the sets S_1, S_1', S_2, S_2', S, and S' determined by this factor. Then, we just check if these sets form an acceptable 6-tuple or not, and if the chosen factor is in L; if both checks return positive answers, we accept the input word, otherwise we reject it.

This algorithm is made of several different phases, which we present separately. Assume that L is specified by a deterministic automaton accepting it $M = (Q, V, q_o, F, \delta)$. Also, for the first seven phases of the algorithm we assume that the input word w has at least length $2k + 2$, otherwise it belongs to $HL_k^*(L)$ if and only if it lies in L. For simplicity, if S is an ordered list with m elements $(x_1, x_2, \ldots, x_m)$, we denote by $S[\ell]$ the ℓ-th element of that list, namely x_ℓ. By deleting an element $S[\ell]$ from a list as the above we obtain a new list with $m-1$ elements, $(x_1, x_2, \ldots, x_{\ell-1}, x_{\ell+1}, \ldots, x_m)$. By adding a new element x to the end of the list S above we obtain a new list with $m+1$ elements $(x_1, x_2, \ldots, x_m, x)$.

In the first phase we initialize the lists we use in the rest of the computation.

Algorithm 3.1 Phase 1: Initialization

```
Initialize the ordered lists $S_1, S_2, S'_1, S'_2, S, S'$ as the void lists;
Initialize the words $last_k = \lambda$ and $last_{k+1} = \lambda$;
Set the variables $phase = 2$, $i = 1$, $f = 1$, and $count = 0$; set the state $q = q_0$;
```

In the second phase of the algorithm we just read the first k symbols of the input word, and store them in the word $last_k$.

Algorithm 3.2 Phase 2

```
while $phase = 2$ do
  Add the symbol $w[i]$ at the end of $last_k$;
  Increase $i$ by 1, increase $t$ by 1;
  if $t = k + 1$ then
    Set $phase = 3$;
  end if
end while
```

The third phase of the algorithm is used to compute the lists S_1 and S'_1. We also decide non-deterministically when we have read the first k symbols of a factor of w, that is tested if it is in L, and used to test whether w is in $HL^*_k(L)$ or not.

At the beginning of the fourth phase of the algorithm we have already analyzed the first k symbols of the chosen factor. In this phase we start computing the set S. We also count the number of symbols read after the first k symbols of the factor, in order to detect the precise moment when we should start adding words to S' as well. As soon as we reach this moment or when we decide non-deterministically that we reached the last k symbols of the factor, we move on to one of the following phases. In the fifth phase we finish computing the set S' and start computing the set S. In this phase, as in the previous one, we can also decide non-deterministically that we have reached the last k symbols of the chosen factor, case in which we move to Phase 6.

In the sixth phase we finish reading the chosen factor as well as we finish computing S. After this phase q is in F if and only if the non-deterministically chosen factor is in L.

In the seventh phase we compute the lists S_2 and S'_2. The computation described in this phase stops at the moment we have reached the end of the input word. Then we move along to phase 8, where the decision is made.

Algorithm 3.3 Phase 3

```
while $phase = 3$ do
    Add the symbol $w[i]$ at the end of $last_k$, set $last_{k+1} = last_k$, and delete the first symbol of $last_k$; Let $t$ be the number of elements in $S_1$;
    if $count < k$ then
        Increase $count$ by 1;
    else
        $count = k$;
    end if
    if there exists $\ell < t$ such that $S_1[\ell] = last_{k+1}$ then
        Delete $S_1[\ell]$ from $S_1$, set $S_1'[\ell + 1] = \min(k, S_1'[\ell + 1] + S_1'[\ell])$, delete $S_1'[\ell]$ from $S_1'$, set $count = 1$;
    else
        if $S_1[t] = last_{k+1}$ then
            Delete $S_1[t]$ from $S_1$, delete $S_1'[t]$ from $S_1'$;
        end if
    end if
    Insert $last_{k+1}$ at the end of $S_1$, insert $count$ at the end of $S_1'$;
    Increase $i$ by 1.
    Choose non-deterministically between $phase = 3$ or setting $phase = 4$ ;
end while
```

Algorithm 3.4 Phase 4

```
Set $t = 1$; let $q = \delta(q_0, last_k)$;
while $phase = 4$ do
    Set $q = \delta(q, w[i])$;
    Add the symbol $w[i]$ at the end of $last_k$, set $last_{k+1} = last_k$, and delete the first symbol of $last_k$;
    Insert $last_{k+1}$ in $S$, if it was not in this list already; Increase $t$ by 1;
    Increase $i$ by 1;
    if $t = k + 1$ then
        Set $phase = 5$;
    else
        Choose non-deterministically between $phase = 4$ or setting $phase = 6$ ;
    end if
end while
```

Algorithm 3.5 Phase 5

```
while $phase = 5$ do
    Set $q = \delta(q, w[i])$;
    Add the symbol $w[i]$ at the end of $last_k$, set $last_{k+1} = last_k$, and delete the first symbol of $last_k$;
    Insert $last_{k+1}$ in $S$, if it is not already in this list; Insert $last_{k+1}$ in $S'$, if it was not in this list already;
    Increase $i$ by 1;
    Choose non-deterministically between $phase = 5$ or setting $phase = 6$;
end while
```

Algorithm 3.6 Phase 6

```
Set $t = 1$;
while $phase = 6$ do
    Set $q = \delta(q, w[i])$;
    Add the symbol $w[i]$ at the end of $last_k$, set $last_{k+1} = last_k$, and delete the first symbol of $last_k$;
    Insert $last_{k+1}$ in $S'$, if it was not in this list already;
    Increase $i$ by 1;
    Increase $t$ by 1;
    if $t = k + 1$ then
        Set $phase = 7$;
    end if
end while
```

Finally, once we have computed all the needed lists, we use the eighth phase to decide whether the input word was obtained by hairpin lengthening from the non-deterministically chosen factor. The decision taken in this phase relies on the preprocessing that we have already mentioned: we assume that we know which 6-tuples of lists are acceptable and which are not.

Algorithm 3.7 Phase 7

```
Set $count = 0$.
while $phase = 7$ do
    Add the symbol $w[i]$ at the end of $last_k$, set $last_{k+1} = last_k$, and delete the first symbol of $last_k$;
    if $last_{k+1}$ appears in $S_2$ then
        if $count < k$ then
            Increase $count$ by 1;
        else
            $count = k$;
        end if
    else
        Insert $last_{k+1}$ at the end of $S_2$;
        Add $count$ at the end of $S'_2$ and reset $count = 1$ when $S_2$ contains at least two elements;
    end if
    Increase $i$ by 1.
    if $i = n + 1$ then
        Add $count$ at the end of $S'_2$; set $phase = 8$;
    end if
end while
```

By the remarks made prior to the detailed description of the eight phases it is clear that our algorithm decides exactly $HL_k^*(L)$.

Algorithm 3.8 Phase 8: Decision

```
if $|w| < 2k + 2$ and $w \in L$ then
  ACCEPT; HALT;
else
  if $|w| < 2k + 2$ then
    REJECT; HALT;
  end if
end if
if $|w| \geq 2k + 2$ and $(S_1, S_1', S_2, S_2', S, S')$ is acceptable and $q \in F$ then
  ACCEPT; HALT;
else
  REJECT;
end if
```

Further, one can easily see that this approach can be implemented on a finite automaton. The states of this automaton store the six lists we use, and all the other variables used in the algorithm, except for i (clearly, only a constant memory is needed to do this). The transitions are defined according to the processing described in the phases above. Clearly, the variable i should not be memorized, since this variable is used only to read the symbols of the input word one by one, and this can be done by a finite automaton. Thus, if L is regular, then $HL_k^*(L)$ is also regular.

In conclusion, the class of regular languages is closed under iterated k-hairpin lengthening. □

4 Hairpin Shortening of Regular Languages

We get the following result:

Theorem 3 *The class of regular languages is closed under k-hairpin shortening for any $k \geq 1$.*

Proof. Let us take a regular language L, and denote by $A = (Q, V, q_0, F, \delta)$ the deterministic finite automaton accepting L. We assume that there exists no state $q \in Q$ such that $q_0 = \delta(q, a)$, for some letter $a \in V$. For a fixed integer $k \geq 1$, we show that HSS_k is regular. We define the non-deterministic automaton $A' = (Q', V, q_0, F', \delta')$, as follows:

$$Q' = Q \cup Q \times Q \cup Q \times Q \times \{x \cup [x] \cup (x) | x \in V^i, \text{ where } 0 \leq i \leq k\},$$
$$F = \{(q, q, (\lambda)) | q \in Q\},$$

$$\begin{aligned}
\delta'(q_1, a) &= \delta(q_1, a) \cup \{(\delta(q_1, a), q_2) | f = \delta(q_2, \overline{a}) \text{ and } f \in F\},\\
\delta'((q_1, q_2), a) &= \{(\delta(q_1, a), q_2') | q_2 = \delta(q_2', \overline{a})\} \cup\\
&\quad \{(\delta(q_1, a), q_2', \lambda) | q_2 = \delta(q_2', \overline{a})\},\\
\delta'((q_1, q_2, x), a) &= \{(\delta(q_1, a), q_2, ax) || x| < k\} \cup \{(\delta(q_1, a), q_2, [x]) || x| = k\},\\
\delta'((q_1, q_2, [x]), a) &= \{(\delta(q_1, a), q_2, [x])\} \cup \{(\delta(q_1, a), q_2, (x))\},\\
\delta'((q_1, q_2, (xa)), \overline{a}) &= \{(q_1, q_2', (x)) | q_2 = \delta(q_2', \overline{a})\}.
\end{aligned}$$

Our automaton accepts all words $\gamma\delta\alpha\beta\overline{\alpha}$ where $\gamma\delta\alpha\beta\overline{\alpha\gamma} \in L$. In order to see how it works, please note that at the beginning the automaton just reads the prefix of γ, possibly empty that precedes δ. After that, with a non-deterministic choice we ensure that reading δ, we would have a word that has $\overline{\delta^R}$ as a suffix in L. Next we again non-deterministically start to read α, and remember the letters for a later comparison with the end of the word. Please note that since the number of factors of length k is finite, the automaton remains finite. Finally, we non-deterministically start to read β and guess when β ends; then we compare the rest of the word with the complement image of the reverse of α, and accept in the case we end up in the same state.

It is easy to see that A' accepts exactly $HSS_k(L)$. Due to the closure of regular languages on the reverse operation, the result follows. □

A direct consequence is:

Corollary 1 *The class of regular languages is closed under finitely iterated k-hairpin shortening for any $k \geq 1$.*

Let us finally look at the iterated case of hairpin shortening of a regular language. Clearly, given a language $L \subseteq V^*$ and a word $w \in V^*$, if $w \in HS_k^*(L)$, then $HL_k^*(w) \cap L \neq \emptyset$ must hold.

Theorem 4 *The class of regular languages is closed under iterated k-hairpin shortening for all $k \geq 1$.*

Proof. The proof is to some extend similar to that of Theorem 2. Let $A = (Q, V, q_0, F, \delta)$ be a deterministic finite automaton with the transition function δ totally defined. We now consider the following nondeterministic algorithm consisting of three phases: *initialization, update, decision.*

The input of this algorithm is the automaton A, an integer $k \geq 1$, and a word $w \in V^*$ with $|w| \geq 2k + 2$. Clearly, any word shorter than $2k + 2$ is in $HS_k^*(L)$ if and only if it belongs to L. The first phase can be accomplished by scanning once from left to right the input word. The second phase is devoted to the update of all variables initialized in the first phase.

Phases 2 and 3 are executed alternatively until a decision is made in Phase 3.

It is not hard to see that the algorithm decides any input, as in the first phase we make a number of steps proportional to the length of the input, while

Algorithm 4.1 Phase 1: Initialization

```
Choose a pair of states $(q, r) \in Q \times Q$ such that $\delta(q, w) = r$;
$P = w[1..k]$;
$S = w[|w| - k + 1..|w|]$;
$L = \{x \in V^* \mid w = uxv, |x| = k + 1, u, v \in V^*, |v| \geq k + 1\}$;
$R = \{x \in V^* \mid w = uxv, |x| = k + 1, u, v \in V^*, |u| \geq k + 1\}$;
Set $phase := 2$;
```

Algorithm 4.2 Phase 2: Update

```
Choose only one of the two situations:
if $\overline{P^R a} \in R$ for some $a \in V$ then
    $P = aP[1..k - 1]$;
    $L = L \cup \{aP\}$;
    $q = q'$ such that $\delta(q', a) = q$;
end if
if $\overline{aS^R} \in L$ for some $a \in V$ then
    $S = S[2..k]a$;
    $R = R \cup \{Sa\}$;
    $r = \delta(r, a)$;
end if
$phase := 3$;
```

Algorithm 4.3 Phase 3: Decision

```
if $q = q_0$ and $r \in F$ then
    ACCEPT;
else
    if $(q, r, L, R, P, S)$ has been considered in Phase 2 then
        REJECT;
    else
        $phase := 2$;
    end if
end if
```

in the second and third phase we make only a finite number of steps, depending only on k. Note that in these latter phases the algorithm does not read any input symbol, so, basically, the only part of the algorithm where we need to have access to the input is Phase 1. The proof is complete as soon as we note that this algorithm can be implemented on a finite automaton. □

References

1. Bottoni, P., Labella, A., Manca, V., Mitrana, V.: Superposition based on watson-crick-like complementarity. Theory of Computing Systems 39(4), 503–524 (Jul 2006)
2. Castellanos, J., Mitrana, V.: Some remarks on hairpin and loop languages. In: Words, Semigroups, and Transductions '01. pp. 47–58 (2001)
3. Cheptea, D., Martín-Vide, C., Mitrana, V.: A new operation on words suggested by dna biochemistry: Hairpin completion. Transgressive Computing p. 216228 (2006)
4. Diekert, V., Kopecki, S.: Complexity results and the growths of hairpin completions of regular languages. In: Proceedings of the 15th international conference on Implementation and application of automata. Lecture Notes in Computer Science, vol. 6482, pp. 105–114. Springer-Verlag, Berlin, Heidelberg (2011)
5. Diekert, V., Kopecki, S., Mitrana, V.: On the hairpin completion of regular languages. In: Leucker, M., Morgan, C. (eds.) Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5684, pp. 170–184. Springer (2009)
6. Ito, M., Leupold, P., Manea, F., Mitrana, V.: Bounded hairpin completion. Information and Computation 209(3), 471–485 (2011)
7. Kopecki, S.: On iterated hairpin completion. Theoretical Computer Science 412(29), 3629–3638 (2011)
8. Manea, F.: A series of algorithmic results related to the iterated hairpin completion. Theoretical Computer Science 411(48), 4162–4178 (2010)
9. Manea, F., C, M.V., Mitrana, V.: On some algorithmic problems regarding the hairpin completion. Discrete Applied Mathematics 157(9), 2143–2152 (2009)
10. Manea, F., Martín-Vide, C., Mitrana, V.: Hairpin lengthening. In: Ferreira, F., Löwe, B., Mayordomo, E., Gomes, L.M. (eds.) CiE. Lecture Notes in Computer Science, vol. 6158, pp. 296–306. Springer (2010)
11. Manea, F., Mitrana, V.: Hairpin completion versus hairpin reduction. In: Cooper, S.B., Löwe, B., Sorbi, A. (eds.) CiE. Lecture Notes in Computer Science, vol. 4497, pp. 532–541. Springer (2007)
12. Manea, F., Mitrana, V., Yokomori, T.: Two complementary operations inspired by the DNA hairpin formation: Completion and reduction. Theoretical Computer Science 410(4-5), 417–425 (2009)
13. Păun, G., Rozenberg, G., Yokomori, T.: Hairpin languages. International Journal of Foundations of Computer Science 12, 837–847
14. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages. Springer-Verlag, Berlin, Heidelberg (1997)