Hairpin Lengthening*

Florin Manea[1,3] **Carlos Martín-Vide**[2] **Victor Mitrana**[3,4]

[1]Otto-von-Guericke-University Magdeburg, Faculty of Computer Science
PSF 4120, D-39016 Magdeburg, Germany.
E-mail: flmanea@fmi.unibuc.ro

[2]Research Group in Mathematical Linguistics, Rovira i Virgili University
Avinguda Catalunya, 35, 43002 Tarragona, Spain
E-mail: carlos.martin@urv.cat

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

[4]Departamento de Organización y Estructura de la Información
Escuela Universitaria de Informática, Universidad Politécnica de Madrid
Crta. de Valencia Km. 7, 28031, Madrid, Spain

Abstract. The hairpin completion is a natural operation of formal languages which has been inspired by molecular phenomena in biology and by DNA-computing. We consider here a new variant of the hairpin completion, called hairpin lengthening, which seems more appropriate for practical implementation. The variant considered here concerns the lengthening of the word that forms a hairpin structure, such that this structure is preserved, without necessarily completing the hairpin. Although our motivation is based on biological phenomena, the present paper is more about some algorithmic properties of this operation. We prove that the iterated hairpin lengthening of a language recognizable in $\mathcal{O}(f(n))$ time is recognizable in $\mathcal{O}(n^2 f(n))$ time, while the one-step hairpin lengthening of such a language is recognizable in $\mathcal{O}(nf(n))$ time. Finally, we propose an algorithm for computing the hairpin lengthening distance between two words in quadratic time.

Keywords: DNA computing, hairpin structure, hairpin completion, hairpin lengthening, formal languages.

1 Introduction

This paper is a continuation of a series of works started with [2] (based on some ideas from [1]), where, inspired by the DNA manipulation, a new formal operation on words, called hairpin completion, was introduced. The initial work was followed by a series of related papers ([8–11]), where both the hairpin completion, as well as its inverse operation, the hairpin reduction, were further investigated.

* Florin Manea's work was supported by the *Alexander von Humboldt Foundation.* Victor Mitrana's work was supported by the PIV Program of the *Agency for Management of University and Research Grants.*

Single-stranded DNA molecules (ssDNA) are composed by nucleotides which differ from each other by their bases: A (adenine), G (guanine), C (cytosine), and T (thymine). Therefore each ssDNA may be viewed as a finite string over the four-letter alphabet $\{A, C, G, T\}$. 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 also called *annealing*. Similarly, RNA molecules are chains of nucleotides having the bases A, G, C and U (uracil), with A complementary to U, and C to G. An intramolecular base pairing, known as *hairpin*, is a pattern that can occur in single-stranded DNA or RNA molecules. Hairpin or hairpin-free structures have numerous applications to DNA-computing and molecular genetics. In many 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. [12, 4, 5, 7] and the references therein.

In [2] a new formal operation on words is introduced, namely the *hairpin completion*. It consists of three biological principles. Besides the Watson-Crick complementarity and annealing, the third biological phenomenon is that of *lengthening DNA by polymerases*. In our case the phenomenon produces a new molecule as follows: one starts with a hairpin - which is, here, a single-stranded molecule, such that one of its ends (a prefix or, respectively, a suffix) is annealed to another part of itself by Watson-Crick complementarity -, and a *polymerization buffer* with many copies of the four basic nucleotides. Then, the initial hairpin is prolonged 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).

In this paper we consider a new variant of the hairpin completion, called hairpin lengthening, which seems more appropriate for practical implementation. This variant concerns the prolongation of a strand which forms a hairpin, similarly to the process described above, 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.

Nevertheless, it seems interesting to consider the iterated versions of the hairpin completion or lengthening. Since these operations can be seen as phenomena by which a single-stranded molecule evolves into a new single-stranded molecule, it is natural to consider the situation when multiple evolution steps occur, thus the initial word is transformed by multiple hairpin completion/lengthening steps.

In this context a natural algorithmic question occurs: "given two words, can we decide if the smaller one evolved (in one-step or by iterated hairpin completion/ lengthening) into the longer one?". Moreover, one can be also interested in finding what is the minimum number of steps needed to transform a word into another by iterated application of hairpin completion/lengthening. In the case of the hairpin completion, these problems were approached in [8, 11]. In this paper, we prove that the iterated hairpin lengthening of a language recognizable in $\mathcal{O}(f(n))$ time is recognizable, in its turn, in $\mathcal{O}(n^2 f(n))$ time; by customizing the proof of this result, one can show that the one-step hairpin lengthening of a language recognizable in $\mathcal{O}(f(n))$ time is recognizable in $\mathcal{O}(nf(n))$ time. Then we define the hairpin lengthening distance between two words and propose an algorithm for computing it in quadratic time. Note that all the time complexity bounds we show here hold on the unit cost RAM model.

2 Preliminaries

Given a word w over an alphabet V, we denote by $|w|$ its length, while $w[i..j]$ denotes the subword of w starting at position i and ending at position j, $1 \leq i \leq j \leq |w|$. 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 ${}^R : V^* \longrightarrow V^*$, $(a_1 a_2 \dots a_n)^R = a_n \dots a_2 a_1$. Note that ^{R} is an involution and an *anti-morphism* ($(xy)^R = y^R x^R$ for all $x, y \in V^*$). Note also that the two mappings $\bar{\cdot}$ and $\cdot^R$ commute, namely, for any word x, $(\overline{x})^R = \overline{x^R}$ 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 $k \geq 1$, 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 $HL(w) = \bigcup_{k \geq 1} HL_k(w)$. Clearly, $HL_{k+1}(w) \subseteq HL_k(w)$ for any $w \in V^+$ and $k \geq 1$. The hairpin lengthening is naturally extended to languages by $HL_k(L) = \bigcup_{w \in L} HL_k(w)$.

This operation is schematically illustrated in Figure 1.

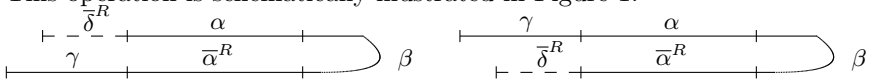

Figure 1: Hairpin lengthening

The iterated version of the 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 $HL_k^*(L) = \bigcup_{w\in L} HL_k^*(w)$.

3 Complexity of the hairpin lengthening

A key means in this section is the rather well-known Knuth-Morris-Pratt algorithm (KMP for short, [6]), a classical algorithm used to locate all the occurrences of a given word x, usually called pattern, in another given word w, usually called text, with linear time-complexity $\mathcal{O}(|x|+|w|)$ and linear working-space. Note that while running the main procedure of this algorithm (see [3] for a detailed presentation) we can compute, without changing the overall time or space complexity, an array of $|x|$ natural numbers $LO_{x,w}$ (Leftmost Occurrence) defined by $LO_{x,w}[i] = \begin{cases} t, \text{ if the leftmost occurrence of } x[1..i] \text{ in } w \text{ is } w[t-i+1..t], \\ 0, \text{ if } x[1..i] \text{ does not appear in } w. \end{cases}$
Clearly, if $LO_{x,w}[i] = 0$, then $LO_{x,w}[j] = 0$ for all $j > i$; moreover, for all $i \geq 1$ $LO_{x,w}[i] \leq LO_{x,w}[i+1]$, provided that $LO_{x,w}[i+1] \neq 0$.

The main result of this section is:

Theorem 1 *For every $k \geq 1$ and every language L recognizable in $\mathcal{O}(f(n))$ time, the iterated k-hairpin lengthening of L is recognizable in $\mathcal{O}(n^2 f(n))$ time.*

Proof. It is worth mentioning that the argument used for proving a similar complexity result in [8] for hairpin completion does not work here. The algorithm proposed in [8] is based on the fact that every word obtained by iterated hairpin completion in more than one step has a non-trivial suffix that equals the reverse of the complement of its prefix, and this property can be efficiently tested. In the case of the hairpin lengthening this property does not hold, thus we have to develop another approach. However, in both cases we rely on a dynamic programming strategy (which may make the two algorithms seem similar).

Let w be a word of length n. We define a function, $Member_{L,k}(w)$, which decides whether or not $w \in HL_k^*(L)$. The algorithm implemented by this function computes, as the main data structure, a $n \times n$ matrix M with binary entries defined by $M[i][j] = (w[i..j] \in HL_k^*(L))$, that is $M[i][j]$ has the same truth value as $w[i..j] \in HL_k^*(L)$. The computation of M is based on a dynamic programming approach. Initially, for all $i, j \in \{1, \ldots, n\}$, we set $M[i][j] = 1$, provided that $w[i..j] \in L$, and $M[i][j] = 0$, otherwise. Further on, we analyze all the subwords of w, in increasing order of their length; in order to decide whether $w[i..j]$ can be obtained by iterated k-hairpin lengthening from a word in L we simply have to check whether one of the following conditions is satisfied:

– there exists an index $i + 2k + 2 \leq s < j$ such that $w[i..s] \in HL_k^*(L)$ and $w[i..j] \in HLS_k(w[i..s])$,
– there exists an index $i < t \leq j - 2k - 2$ such that $w[t..j] \in HL_k^*(L)$ and $w[i..j] \in HLP_k(w[t..j])$.
Note that the search for the indices s and t can be carried out because of the dynamic programming strategy.

However, the approach described above cannot be implemented efficiently in a direct way. To this aim we need some additional data structures. We define two $n \times n$ upper triangular matrices P_s and P_p, having natural number entries, with the following meaning:
– $P_s[i][j]$ stores the position on which the rightmost occurrence of $\overline{w[i..j]}^R$ starts in $w[1..i-1]$. By default, we set $P_s[1][j] = 0$ for all $j \leq n$ and $P_s[i][j] = 0$ for all $j < i \leq n$.
– $P_p[i][j]$ stores the position on which the leftmost occurrence of $\overline{w[i..j]}^R$ ends in $w[j+1..n]$. By default, we set $P_p[i][n] = 0$ for all $i \leq n$ and $P_p[i][j] = 0$ for all $j < i \leq n$.

We claim that the nontrivial elements of the two matrices can be computed as follows:
– $P_s[i][j] = i - LO_{w[i..n],\overline{w[1..i-1]}^R}[j - i + 1]$ for all i and j such that $n \geq j \geq i > 1$.
– $P_p[i][j] = j + LO_{\overline{w[1..j]}^R,w[j+1..n]}[j - i + 1]$ for all i and j such that $n > j \geq i \geq 1$.

Algorithm 1 *ComputeMat*(w): returns the values of the two matrices

for $i = 2$ to n **do**
  Compute $LO_{w[i..n],\overline{w[1..i-1]}^R}$;
end for
for $j = 1$ to $n - 1$ **do**
  Compute $LO_{\overline{w[1..j]}^R,w[j+1..n]}$;
end for
for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    if $i = 1$ or $j < i$ **then**
      $P_s[i][j] = 0$
    else
      $P_s[i][j] = i - LO_{w[i..n],\overline{w[1..i-1]}^R}[j - i + 1]$
    end if
    if $j = n$ or $j < i$ **then**
      $P_p[i][j] = 0$
    else
      $P_p[i][j] = j + LO_{\overline{w[1..j]}^R,w[j+1..n]}[j - i + 1]$
    end if
  end for
end for
Return the pair of matrices (P_s, P_p)

Indeed, $P_s[i][j] = t$, $t < i$, implies $i - LO_{w[i..n],\overline{w[1..i-1]}^R}[j - i + 1] = t$, hence $w[i..j] = \overline{w[t..t + j - i]}^R$ and t is the greatest number, with $t + j - i < i$, that

verifies this relation. On the other hand, $P_s[i][j] = i$ implies that $w[i..j]$ does not occur as a factor of $\overline{w[1..i-1]}^R$.

Analogously, $P_p[i][j] = t$, $t > j$, implies $\overline{w[i..j]}^R = w[t-j+i..t]$ and t is the smallest number, with $t - j + i > j$, that verifies this relation. On the other hand, $P_p[i][j] = j$ implies that $\overline{w[i..j]}^R$ does not occur as a factor of $w[j+1..n]$.

By these considerations, we may easily conclude that the two matrices can be computed in quadratic time and space by Algorithm 1.

As far as the computation of M defined in the beginning of this proof is concerned, we conclude that $M[i][j] = 1$ if and only if one of the following conditions holds:
– $w[i..j] \in L$.
– There exists an index s such that $i \leq s \leq j$, $M[i][s] = 1$ and $w[s-k+1..j]$ is a subword of $\overline{w[i..s-k]}^R$, hence $P_s[s-k+1][j] \geq i$.
– There exists an index t such that $i \leq t \leq j$, $M[t][j] = 1$ and $w[i..t+k-1]$ is a subword of $\overline{w[i..t+k]}^R$, hence $P_p[i][t+k-1] \leq j$.

On the other hand, it is rather plain that $P_s[i][j] > P_s[i'][j]$ for $i' < i$, and $P_p[i][j] < P_p[i][j']$ for $j < j'$. From these considerations we deduce:
– If there exists the index s such that $w[i..j] \in HLS_k(w[i..s])$, then $w[i..j] \in HLS_k(w[i..s'])$ for every index s' with $j > s' > s$. Indeed, from $w[i..j] \in HLS_k(w[i..s])$ it follows that $P_s[s-k+1][j] \geq i$, thus $P_s[s'-k+1][j] \geq i$, for all $s' > s$, which is equivalent to $w[i..j] \in HLS_k(w[i..s'])$.
– If there exists the index t such that $w[i..j] \in HLP_k(w[t..j])$, then $w[i..j] \in HLP_k(w[t'..j])$ for every index t' with $i < t' < t$. Indeed, from $w[i..j] \in HLP_k(w[t..j])$ it follows that $P_s[i][t+k-1] \leq j$, thus $P_s[i][t'+k-1] \leq j$, for all $t' < t$, which is equivalent to $w[i..j] \in HLP_k(w[t'..j])$.

Algorithm 2 $Member_{L,k}(w)$: returns the truth value of $w \in HL_k^*(L)$

```
1: Initialize matrix M: if w[i..j] ∈ L set M[i][j] = 1, otherwise set M[i][j] = 0
2: (P_s, P_p) = ComputeMat(w);
3: Initialize arrays r and l;
4: for len = 1 to n do
5:    for i = 1 to n − len + 1 do
6:       j = i + len − 1;
7:       if M[i][j] = 0 and r[i] ≠ 0 and P_s[r[i] − k + 1][i] ≥ i then
8:          M[i][j] = 1;
9:       end if
10:      if M[i][j] = 0 and l[i] ≠ 0 and P_p[i][l[j] + k − 1] ≤ j then
11:         M[i][j] = 1;
12:      end if
13:      if M[i][j] = 1 then
14:         r[i] = j and l[j] = i;
15:      end if
16:   end for
17: end for
18: Return true if M[1][n] = 1 or false otherwise.
```

This shows that $M[i][j] = 1$ if and only if one of the following holds:
– $w[i..j] \in HLS_k(w[i..s])$ where s is the greatest index such that $M[i][s] = 1$ and $s < j$;
– $w[i..j] \in HLP_k(w[t..j])$ where t is the smallest index such that $M[t][j] = 1$ and $i < t$.

In conclusion, M can be computed as shown in Algorithm 2 which makes use of two further arrays, l and r with n positions each, where $r[i]$ is the greatest s found so far such that $M[i][s] = 1$, and $l[j]$ is the smallest t found so far such that $M[t][j] = 1$.

The soundness of the algorithm follows from the aforementioned consideration. It is easy to note that the most time consuming part of the algorithm is that formed by the step 1 which requires $\mathcal{O}(n^2 f(n))$ time. All the other parts require quadratic time. □

A closer look to the proof reveals that the overall complexity of the algorithm is $\mathcal{O}(\max(f(n), n^2))$, provided that all the subwords of a word w (with $|w| = n$) contained in L can be found also in $\mathcal{O}(f(n))$ time. This is the case of context-free languages ($f(n) = n^3$) and regular languages ($f(n) = n^2$).

Using techniques inspired by the above algorithms we can prove that:

Theorem 2 *For every $k \geq 1$ and every language L recognizable in $\mathcal{O}(f(n))$ time, the k-hairpin lengthening of L is recognizable in $\mathcal{O}(nf(n))$ time. If L is regular (context-free), $HL_k(L)$ is recognizable in $\mathcal{O}(n)$ (respectively, $\mathcal{O}(n^3)$) time.*

4 Hairpin Lengthening Distance

The *k-hairpin lengthening distance* between two words x and y is defined as the minimal number of hairpin lengthening operations which can be applied either to x in order to obtain y or to y in order to obtain x. If none of them can be obtained from the other by iterated hairpin lengthening, then the distance is ∞. Formally, the k-hairpin lengthening distance between x and y, denoted by $HLD_k(x, y)$, is defined by: $HLD_k(x, y) = \begin{cases} \min\{p \mid x \in HL_k^p(y) \text{ or } y \in HL_k^p(x)\}, \\ \infty, \text{ if neither } x \in HL_k^p(y) \text{ nor } y \in HL_k^p(x) \end{cases}$

We stress from the very beginning that the function HLD_k defined above is not a distance function in the mathematical sense, since it does not necessarily verify the triangle inequality. However, we call it distance as similar measures (based on different operations on words) are called distances in the literature.

In our view, it is rather surprising that the hairpin lengthening distance can be computed in quadratic time, using a greedy strategy. We recall from [11] that the best known algorithm for computing the hairpin completion distance requires $\mathcal{O}(n^2 \log n)$ time, where n is the length of the longest input word.

Theorem 3 *The k-hairpin lengthening distance between two words x and w can be computed in $\mathcal{O}(max(|x|, |w|)^2)$.*

Proof. First, let us define the notion of derivation, in the context of hairpin lengthening. We say that the word x derives the word w, and denote it $x \to w$, if and only if $w \in HL_k(x)$. If x is a subword of w, $w \in HL_k(x)$, and w has length n, we define the *maximal derivation of w from x* as the sequence of p derivation steps $w_0 \to w_1 \to \ldots \to w_p$, where:
– $w_0 = x$ and $w_p = w$;

– for any i, with $p > i \geq 0$, we either have $w_i = w[s_i..t_i]$ and $w_{i+1} = w[s_i..t_{i+1}]$, where t_{i+1} is the maximum value t such that $w[s_i..t] \in HLS_k(w[s_i..t_i])$, or we have $w_i = w[s_i..t_i]$ and $w_{i+1} = w[s_{i+1}..t_i]$, where s_{i+1} is the minimum value s such that $w[s..t_i] \in HLS_k(w[s_i..t_i])$.

In the following we show that if $w \in HL_k^p(x)$ then there exists a maximal derivation of w from x consisting of at most p derivation steps. Clearly, if $p = 1$ then the derivation $x \rightarrow w$ is already a maximal derivation of w from x. Let us assume that $p > 1$.

Since $w \in HL_k^p(x)$ there exists a sequence of p derivation steps $x = w_0 \rightarrow w_1 \rightarrow \ldots \rightarrow w_p = w$. Assume by contradiction that this derivation is not maximal. Therefore in this derivation we have, for some $p - 1 > i \geq 0$, one of the following cases:
– $w_i \rightarrow w_{i+1}$, $w_i = w[s_i..t_i]$, $w_{i+1} = w[s_{i+1}..t_{i+1}]$ and there exists $t'_{i+1} > t_{i+1}$ such that $w[s_i..t'_{i+1}] \in HLS_k(w_i)$.
– $w_i \rightarrow w_{i+1}$, $w_i = w[s_i..t_i]$, $w_{i+1} = w[s_{i+1}..t_{i+1}]$ and there exists $s'_{i+1} < s_{i+1}$ such that $w[s'_{i+1}..t_i] \in HLS_k(w_i)$.

We show how this derivation can be transformed into a maximal derivation of w from x. We analyze only the first case, since the other can be treated in a similar fashion.

Let $w'_{i+1} = w[s_i..t'_{i+1}]$. If $w_{i+2} = w[s_{i+2}..t_{i+1}]$ with $s_{i+2} < s_i$ (i.e. it was obtained from w_{i+1} by hairpin lengthening with a prefix), then we can derive $w'_{i+2} = w[s_{i+2}..t'_{i+1}]$ from w'_{i+1}. This process can continue until we reach a derivation step where a suffix is added to the derived string. Without loss of generality, we may assume that w_{i+2} is actually obtained in this manner from w_{i+1}. That is $w_{i+2} = w[s_i..t_{i+2}]$ with $t_{i+2} > t_{i+1}$. There are two cases to be discussed: if $t_{i+2} < t'_{i+1}$, then we simply skip this derivation step; if $t_{i+2} > t'_{i+1}$, then we still can obtain w_{i+2} from w'_{i+1} (by arguments similar to those used in the proof of Theorem 1). It follows that we can still continue the derivation and obtain w in at most as many steps as in the original derivation by replacing the derivation $w_i \rightarrow w_{i+1}$ with the derivation $w_i \rightarrow w'_{i+1}$.

Consequently, any sequence of p derivation steps leading from x to w can be transformed into a maximal derivation of w from x, with at most p steps (note that such a property does not hold for the hairpin completion operation). Now we can deduce that $HLD_k(x, w)$ equals the minimum number of derivation steps performed in a maximal derivation of w from x. Moreover, one can easily note that if $x = w_0 \rightarrow w_1 \rightarrow \ldots \rightarrow w_p = w$ is a maximal derivation of w from x, then $w_0 \rightarrow \ldots \rightarrow w_i$ is a maximal derivation of w_i from x, for all $i \leq p$.

Now, let us return to the algorithm for computing the hairpin lengthening distance. Assume that x and w are two words of length m and n, respectively. We are interested in computing the distance $HLD_k(x, w)$. We can assume, without loss of generality, that $m < n$. As we have seen, $HLD_k(x, w)$ equals the minimum number of derivation steps performed in a maximal derivation of w from x, provided that such a derivation exists, or ∞, otherwise; it is clear that w can not be transformed into x.

The first step in computing the minimum number of derivation steps performed in a maximal derivation of w from x is to compute the $n \times n$ matrices C_s and C_p, defined by:
– $C_s[i][j] = t$ if and only if $w[i..t] \in HLS_k(w[i..j])$ and $w[i..t'] \notin HLS_k(w[i..j])$ for all the indices t' such that $n \geq t' > t$.
– $C_p[i][j] = s$ if and only if $w[s..j] \in HLP_k(w[i..j])$ and $w[s'..j] \notin HLP_k(w[i..j])$ for all the indices s' such that $1 \leq s' < s$.
To compute these matrices we will need the auxiliary $n \times n$ matrices P'_s and P'_p: $P'_s[i][j] = \max\{t \mid j \leq t \leq n, P_s[j][t] = i\}$ and $P'_p[i][j] = \min\{s \mid 1 \leq s \leq i, P_p[s][i] = j\}$.

Clearly, matrices P'_s and P'_p can be computed using the $ComputeMat(w)$ function, within the same time: basically we initialize all the elements of these matrices with 0, and, then, we update an element ($P'_s[i][j]$, for instance) each time we need to, according to their definition (in our example, when we identify a new s such that $P_s[j][s] = i$ we set $P'_s[i][j]$ to the maximum value from its former value and s).

Algorithm 3 $HLD_k(x, w)$: returns $HLD_k(x, w)$ $(2k + 1 < |x| < |w|)$

```
1: Initialize array H;
2: Compute matrices C_s and C_p;
3: for l = 2k to n do
4:    for i = 1 to n − l + 1 do
5:       j = i + l − 1;
6:       if C_s[i][j − k + 1] ≠ 0 then
7:          H[i][C_s[i][j − k + 1]] = min{H[i][C_s[i][j − k + 1]], 1 + H[i][j]}
8:       end if
9:       if C_p[i + k − 1][j] ≠ 0 then
10:         H[C_p[i + k − 1][j]][j] = min{H[C_p[i + k − 1][j]][j], 1 + H[i][j]}
11:      end if
12:   end for
13: end for
14: Return H[1][n]
```

Now we can show how the matrices C_s and C_p are computed. It is not hard to see that the following recurrence relations hold:
– $C_s[i][j] = 0$ for $i > j$ or $j = n$, and $C_s[i][j] = \max\{P'_s[i][j], C_s[i + 1][j]\}$ otherwise.
– $C_p[i][j] = 0$ for $i > j$ or $i = 1$, and $C_p[i][j] = \min\{P'_p[i][j], C_p[i][j - 1]\}$ otherwise.

Finally we can compute the hairpin lengthening distance between the two words x and w. The strategy that we use is a mixture of dynamic programming and greedy: we analyze, in increasing order of their length (by dynamic programming), all the subwords of w and construct for each of them the subwords of w that can be derived from it by extending it as much as possible using hairpin lengthening - as in each step of a maximal derivation of w from x (greedy strategy); at the same time we count for each of the constructed words the minimum number of derivation steps needed to obtain that subword from x in a maximal derivation of w from x, and store these values in a $n \times n$ matrix H. In this manner $H[i][j]$ will store, at the end of the computation, the value $HLD_k(x, w[i..j])$.

In more detail, when we analyze a subword $w[i..j]$ we proceed as follows:
– Initially we set $H[i][j] = \infty$ for all $i, j \in \{1, \ldots, n\}$; then we set $H[i][j] = 0$ if

$w[i..j] = x$.
– Further, for a pair of indices i, j, with $i < j$ and $j - i + 1 > 2k$, we set:
$H[i][C_s[i][j-k+1]] = \min\{H[i][C_s[i][j-k+1]], 1 + H[i][j]\}$, if $C_s[i][j-k+1] \neq 0$,
$H[C_p[i+k-1][j]][j] = \min\{H[C_p[i+k-1][j]][j], 1 + H[i][j]\}$, if $C_p[i+k-1][j] \neq 0$.

It is not hard to see that for each subword $w[s..t]$ of w we verify all the possible ways in which it can be derived from another subword of w using the rules of a maximal derivation of w from x. When we find such a derivation $w[i..j] \rightarrow w[s..t]$, we have already computed $H[i][j]$, thus we can update, if necessary, the value $H[s][t]$. Therefore, the relations given above will lead to the correct computation of the elements of the matrix H. To find the distance $HLD_k(x, w)$ we simply have to return $H[1][n]$.

The implementation of this strategy is given in the Algorithm 3. The time complexity of the above algorithm is $\mathcal{O}(n^2)$, where n is length of the longest input word. Indeed, steps 1 and 2 can be executed in quadratic time each. The part formed by steps $3 - 13$ requires quadratic time, as well. In conclusion, the overall running time of the algorithm is $\mathcal{O}(\max(|x|, |w|)^2)$. □

References

1. P. Bottoni, A. Labella, V. Manca, V. Mitrana. Superposition based on Watson-Crick-like complementarity. *Theory of Computing Systems* 39(4):503–524, 2006.
2. D. Cheptea, C. Martin-Vide, V. Mitrana. A new operation on words suggested by DNA biochemistry: hairpin completion, *Proc. Transgressive Computing*, 216–228, 2006.
3. T.H. Cormen, C.E. Leiserson, R.R Rivest. *Introduction to Algorithms*, MIT Press, 1990.
4. R. Deaton, R. Murphy, M. Garzon, D.R. Franceschetti, S.E. Stevens. Good encodings for DNA-based solutions to combinatorial problems, *Proc. of DNA-based computers II*, DIMACS Series, vol. 44:247–258, 1998.
5. M. Garzon, R. Deaton, L.F. Nino, S.E. Stevens Jr., M. Wittner. Genome encoding for DNA computing, *Proc. Third Genetic Programming Conference*, Madison, MI, 684–690, 1998.
6. D.E. Knuth, J.H. Morris, V.R. Pratt. Fast pattern matching in strings, *SIAM Journal of Computing* 6:323–350, 1977.
7. L. Kari, S. Konstantinidis, P. Sosik, G. Thierrin. On hairpin-free words and languages, *Proc. Developments in Language Theory 2005*, LNCS 3572:296-307, 2005.
8. F. Manea, C. Martín-Vide, V. Mitrana. On some algorithmic problems regarding the hairpin completion, *Discr. App. Math.* 157(9):2143-2152, 2009.
9. F. Manea, V. Mitrana. Hairpin completion versus hairpin reduction, *Computation in Europe CiE 2007*, LNCS 4497:532-541, 2007.
10. F. Manea, V. Mitrana, T. Yokomori. Two complementary operations inspired by the DNA hairpin formation: completion and reduction, *Theor. Comput. Sci.* 410(4-5):417-425, 2009.
11. F. Manea. A series of algorithmic results related to the iterated hairpin completion. Submitted.
12. K. Sakamoto, H. Gouzu, K. Komiya, D. Kiga, S. Yokoyama, T. Yokomori, and M. Hagiya. Molecular computation by DNA hairpin formation, *Science* 288:1223–1226, 2000.