Elsevier Editorial System(tm) for Theoretical Computer Science
Manuscript Draft

Manuscript Number:

Title: Testing DNA code words properties of regular languages

Article Type: SI : CS2Bio 2014

Section/Category: C - Theory of natural computing

Keywords: DNA Computing; Formal Languages; Automata Theory

Corresponding Author: Dr. Rosalba Zizza,

Corresponding Author's Institution: University of Salerno

First Author: Rocco Zaccagnino

Order of Authors: Rocco Zaccagnino; Rosalba Zizza; Carlo Zottoli

Testing DNA code words properties of regular languages ☆

Rocco Zaccagnino[a], Rosalba Zizza[a,*], Carlo Zottoli

[a] *Dipartimento di Informatica, Università degli Studi di Salerno, via Giovanni Paolo II 132, 84084 Fisciano (SA), Italy*

Abstract

The *DNA codeword design problem* regards the following position in DNA Computing: given a problem encoded in DNA strands and biochemical processes, the final computation is a concatenation of the input DNA strands that must allow us to recover the solution of the given problem in terms of the input (unique decipherability). Thus the initial set of DNA strands must be a code. In addition, it should satisfy some restrictions, called here DNA properties, in order to prevent them from interacting in undesirable ways. So a new interest towards the design of efficient algorithms for testing whether a language X is a code, has arisen from (wet) DNA Computing, but, as far as we know, only when X is a finite set. In this paper we provide an algorithm for testing whether an infinite but regular set of words is a code that avoids some DNA properties among unwanted intermolecular and intramolecular hybridizations.

Keywords: DNA Computing, Formal Languages, Automata Theory
2000 MSC: 68Q45, 68Q05

1. Introduction

Variable-length codes were investigated in depth for the first time by Schützenberger (1955), by linking the theory of codes with classical noncommutative algebra. Informally, given a finite alphabet Σ, a code is a set of words X on Σ such that any message composed from its elements is uniquely decipherable (UD) and so X guarantees a secure and economic transmission. Sardinas and Patterson [26] characterized codes through a decidable property for a regular

☆ Partially supported by the *FARB* Project *"Aspetti algebrici e computazionali nella teoria dei codici e dei linguaggi formali"* (University of Salerno, 2012), the *FARB* Project *"Aspetti algebrici e computazionali nella teoria dei codici, degli automi e dei linguaggi formali"* (University of Salerno, 2013) and the *MIUR PRIN* 2010-2011 grant *"Automata and Formal Languages: Mathematical and Applicative Aspects"*, code H41J12000190001.

* Corresponding author
Email addresses: zaccagnino@dia.unisa.it (Rocco Zaccagnino), zizza@dia.unisa.it (Rosalba Zizza)

set. When X is a finite set, efficient algorithms have been designed for testing the UD property, by using different approaches (see [4]). Others are known in the regular case [3, 4, 12, 22].

Recently a new interest with respect to the design and implementation of efficient algorithms for testing the UD property has arisen from (wet) DNA Computing [1, 24]. Briefly, every DNA molecule can be viewed as an oriented single-stranded sequence of nucleotides, i.e., a word over the finite alphabet $\Sigma = \{A, C, G, T\}$. The simple property of Watson-Crick complementarity (C and T are respectively complementary to G and A), which allows the precise matching between two oppositely-oriented complementary strands, can be used to assemble DNA strands (hybridization). One aspect of DNA Computing is the possibility of using DNA molecules for solving some "complicated" computational problems (like showed by Adleman in his seminal paper [1]). Briefly, in [1] a small instance of the Hamiltonian Path Problem, a well-known NP-complete problem, has been solved by encoding vertices and edges of the graph into a set of suitable molecules. So the solution, if exists, is nothing but a concatenation of initial molecules, and it has been obtained by biochemical processes, such as Gel Electrophoresis, length separation, PCR. Thus, the input data in DNA computing must be encoded into the form of single or double DNA strands. Due to the importance of this step for the success of computations, much attention has been devoted to define suitable sets of codewords.

Two different approaches have been adopted in the literature to investigate "good" DNA encodings. In [9], solutions based on biochemical conditions have been proposed (see also [21] and the bibliography of [9]). Here we follow the theoretical approach of [14, 15, 16, 17, 18], where the problem is faced from a formal language point of view, by defining a list of properties of words and languages, which if fulfilled by a given set of DNA strands, can guarantee the robustness during computation. The first property that a set of words should have, in order to be used for the encoding the problem, is the UD property, otherwise, starting from the final computation, which is a concatenation of the input DNA strands, we are not able to uniquely recover the solution of the problem. In addition, as complementary parts of single strands can bind together forming a double stranded DNA sequence, we have to impose some restrictions (in the sequel called *DNA properties*) on these sets of codewords languages to prevent them from interacting in undesirable ways. Two classes of hydridizations may be considered: *intermolecular* hybridizations, i.e., two different single stranded bind together, forming a double-stranded DNA molecule; *intramolecular* hybridization, i.e., a piece of a long single stranded molecule hybridizes with another piece of the same molecule. As an example, *hairpins* occur when a factor of a long single stranded DNA molecule is complementary to another factor of the same length. In both situations these hybridizations are unwanted for DNA Computing, since the obtained molecules cannot be used for encoding information. (Another aspect, which will not be considered here, is that molecular processes are not error-free: two DNA sequences which are not perfectly Watson-Crick complementary may hybridize, producing false positives or negatives.) Among them, θ-compliance, θ-freedom and θ-k-hairpin-freeness

have been defined, where θ is a morphism or antimorphism on the alphabet and k is a positive integer. The first two properties avoid two intermolecular hybridizations for which the Watson-Crick complement of a word v in X cannot be a factor of another word of X (resp. the concatenation of two words of X). It is already known that the problem of deciding whether a regular language X is θ-compliant or θ-free is decidable and can be solved in quadratic time with respect to the size of a finite state automaton recognizing X [14]. The θ-k-hairpin-freeness property guarantees that no word of the language contains both a factor v and its involution $\theta(v)$. Obviously, the length of v is a parameter with respect to considering the property (otherwise, the property could be trivial, e.g., when $k = 1$). Thus, more precisely, a language L is θ-k-hairpin-free if there exists no word v such that v and $\theta(v)$ are factors of the same word in L, with $|v| \geq k \geq 1$ and θ is an involution. In this way, we require that "long" factors do not hybridize and we decide how long they are by fixing k. It is known that it is possible to decide whether a regular language is θ-k-hairpin-free, for a given involution θ and k.

For the sake of completeness, we must say that in the literature hairpins are not everywhere considered "bad". As an example, in [25] hairpins are used for performing computation and so it is crucial to design algorithms for generating words with hairpins, as well as the hairpin completion operation [5, 20]. Our paper does not investigate hairpins from this "positive" point of view.

As far as we know, all known algorithms and implementations for testing intermolecular properties have been designed when X is a finite language [19, 23]. It is evident that, for practical usage, only finite set should be considered. Nevertheless, larger and larger instances of NP-complete problems are coded through a (big) set of different words satisfying DNA properties. In addition, consider a finite set X which has been verified to be "good" for DNA computing. If we need to increase the instance of only one word w, and so we examine $X \cup w$, we must check again the whole set $X \cup w$ w.r.t. the DNA properties and the UD property. On the contrary, it could be more useful to test DNA properties of an infinite set of DNA words, which have the same structural description, provided by a regular expression. In this way, we have a regular infinite set X which is a code satisfying some DNA properties, we can choose any subset Y of X to encode our problem, and we can add any word from X to Y, maintaining the goodness of our set. Furthermore, the set X is not a list of many different words, but it is described by a succinct regular expression. This is our motivation for effectively testing DNA properties on regular infinite sets.

In this perspective, we firstly consider the UD property, a necessary property for good encodings. It cannot be efficiently tested for a regular set, if X is given by an ambiguous automaton (otherwise, if X is given by an unambiguous automaton, efficient techniques are known [4]). In [6], the authors used the algorithm of [22] for testing UD property, that presents many advantages w.r.t. others (see Section 3 for details). In addition, McCloskey's tecnique has been also used in the same paper for testing the two intermolecular properties above mentioned. The time complexity of the algorithm is $O(n^2)$, where n is the size of the finite state automaton recognizing X. Following the approach of CodeGen

[19], in [6] a Java implementation is also provided, which takes care of space requirements for implementing automata. The software implements automata by using primitive data types and no external library.

This paper extends results presented in [6], where only intermolecular hybridizations were considered. Indeed, here we also present an algorithm for testing θ-k-hairpin-freeness property for regular languages, based on common characteristics of algorithms for intermolecular properties developed in [6].

The paper is organized as follows. In Section 2 we gathered basics on words, automata and codes. Sections 3-6 review the results of [6] by fixing definitions and providing lacking proofs. Precisely, Section 3 presents the problem of testing the UD property and the algorithm of [22] is detailed in Section 4. DNA properties are defined in Section 5 and Section 6 shows how we can test them on a regular set. Section 7 regards detection of hairpin-free languages. Conclusions give a glance at the implementation of the algorithm and work in progress. For the reader's convenience, the project can be downloaded at http://www.di.unisa.it/professori/zizza/cs2Bio/Implementazione.zip. [1]

2. Basics on Languages, Automata and Codes

We denote by Σ^* the free monoid over a finite alphabet Σ and we set $\Sigma^+ = \Sigma^* \setminus \lambda$, where λ is the empty word. For a word $w \in \Sigma^*$ and $a \in \Sigma$, $|w|$ is the length of w. A word $x \in \Sigma^*$ is a *factor* of $w \in \Sigma^*$ if there are $u_1, u_2 \in \Sigma^*$ such that $w = u_1 x u_2$. When $u_1 = \lambda$ (resp. $u_2 = \lambda$), the word x is named a *prefix* (resp. a *suffix*) of w. We denote by $Fact(w)$ the set of the factors of w. Thus, for a given language $L \subseteq \Sigma^*$, we set $Fact(L) = \cup_{w \in L} Fact(w)$ and by $Fact_k(L) = Fact(L) \cap \{w \in A^* || w| \geq k\}$, i.e., the factors of words in L which are longer than $k-1$.

We suppose the reader familiar with classical definitions on automata [13]. Here we fix only some notations. An NFA (*non-deterministic finite state automaton*) is a 5-tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$, where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final (or accepting) states, $\delta : Q \times (\Sigma \cup \lambda) \to 2^Q$ is the transition function. The transition $\delta(q, \lambda)$ is called a λ-*transition*. If we classically extend the domain of δ to $Q \times \Sigma^*$ we get the function δ^*. $\mathcal{A}$ is deterministic (DFA) if for each state q there is at most one transition labeled by a letter. The language recognized by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. A language L is λ-free is $\lambda \notin L$.

In the sequel, we will abuse the notation by not distinguishing between an NFA and its transition graph, i.e., the graph with node set Q and edge set composed by (p, a, s), for each transition from p to s labeled a. A path π from a state p to a state q is a sequence of transitions $(p, a_1, q_1), (q_1, a_2, q_2), \ldots, (q_{n-1}, a_n, q)$, the word $a_1 \cdots a_n$ is the *label* of the path π and the n is the *length* of π. A state p is *accessible* (resp. *coaccessible*) if there exists a path from q_0 to p (resp. and from p to $f \in Q$). If $p = q_0$ and $q \in F$, then π is an *accepting path*. A λ-path

[1] import the Eclipse project as "Existing project into workspace"

is a path labeled by λ. The *size* of $\mathcal{A}$ is the sum of the numbers of states and transitions in $\mathcal{A}$. (The number of symbols in Σ is taken to be a constant.)

Definition 2.1 *[4] A code $X \subseteq \Sigma^*$ is a set of words such that any word in Σ^* has at most one factorization as a product of elements in X, i.e., for all $c_1, \ldots, c_h, c'_1, \ldots, c'_k \in X$ we have*

$$c_1 \cdots c_h = c'_1 \cdots c'_k \Rightarrow h = k, \quad \textit{and} \quad \forall i \in \{1, \ldots, h\} \quad c_i = c'_i.$$

Classically, $X = \{a, ab, ba\}$ is not a code, since, for example, the word aba cannot be written uniquely as product of words in X. On the contrary, for any $p \geq 1$, the sets Σ^p of all words of length p are codes, named uniform codes. Also sets X such that no word in X is a prefix (resp. suffix) of another word in X, i.e., prefix (resp. suffix) sets, are codes, as well as biprefix sets (i.e., no word in X is a prefix or a suffix of another word in X). Finally, if X is a code, then $\lambda \notin X$ and each non-empty subset of X is a code.

3. The UD Problem

Unlike context-free languages, when X is a regular set, it is known that it is decidable whether X is a code or not [4]. The problem of testing whether a regular set is a code (unique decipherability - UD - problem) is a special case of a well-known problem in automata theory, namely testing whether a given rational expression is unambiguous. Standard decision procedures are known but, as noted in [4], the exact time complexity is unknown and so they cannot be used for our aim.

We recall that in [3, 4], the unambiguous automaton is presented and used for testing the UD property. A finite state automaton $\mathcal{A}$ is an *unambiguous* automaton if for each pair (p, q) of states, there is no word which is the label of two different paths from p to q. The main idea of using automata to decide whether a language is a code is to replace the computation on words by a computation on paths labeled by words. Thus, the uniqueness of factorizations for a code corresponds to the uniqueness of paths in the unambiguous automaton. The real Achilles' heel of this algorithm is clear: if X is specified by means of an ambiguous finite state automaton $\mathcal{A}$, it cannot be directly applied. In order to make $\mathcal{A}$ unambiguous, in the worst case, it is not possible to avoid the exponential explosion of the number of the states. To our knowledge, the unique technique to transform an ambiguous automaton into an unambiguous automaton is the well-known procedure that determinizes the ambiguous automaton (see Question 1 in [7]). Indeed, by definition, each DFA is trivially an unambiguous automaton (see again [7] for details). A possible solution for this problem is presented in [4] and uses the *flower automaton* of X, an "universal" automaton constructed starting from X. Unfortunately, the flower automaton of a language has many states, and it can be also infinite. In particular, the flower automaton of an infinite code is infinite. To overcome these difficulties, Head and Weber proposed an efficient algorithm that involves the construction

of nondeterministic finite transducer associated with X [12]. The complexity of the algorithm is $O(n^2)$, where n is the size of the input NFA. In [6] the authors implemented a technique, based on finite state automata only, which can be applied to regular (even infinite) languages. McCloskey's algorithm [22] is suitable for our scope, and it will be recalled in the next section.

4. UD and Restricted Automata

The algorithm given in [22] takes as input a finite state automaton $\mathcal{A}$ recognizing X and decides whether X is a code. Let n be the size of $\mathcal{A}$.

Definition 4.1 *[22] A finite state automaton $\mathcal{A}$ is in* restricted form *if it has only one accepting state and there are no λ-transitions into that state and no transition of any kind out of that state.*

We often say that $\mathcal{A}$ is a *restricted automaton* if $\mathcal{A}$ is in restricted form. An example from [22] is reported in Figure 1. Observe that any language recognized by a restricted finite state automaton is λ-free.

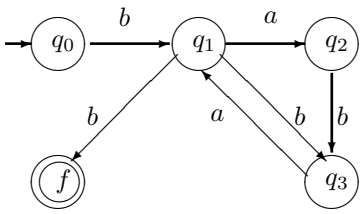


Fig. 1: An unambiguous NFA recognizing $b(aba + ba)^*b$

In [6] it is shown how to transform an NFA in its restricted form (by following the procedure in [8] for the construction given in [22]). We report this technique below.

Let $\mathcal{A} = (\Sigma, Q', \delta', q_0', F')$ be an NFA. We define $\mathcal{A}_{\mathcal{R}} = (\Sigma, Q, \delta, q_0, F)$ to be the NFA such that $Q = Q' \cup \{f\}$, $f \notin Q'$, $q_0 = q_0'$, $F = \{f\}$ and $\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$, defined as follows.

(a) For each $q' \in Q'$, for each $a \in \Sigma \cup \lambda$ such that $q \in \delta'(q', a)$, then $q \in \delta(q', a)$. Loosely speaking, all transitions in $\mathcal{A}$ are transitions in $\mathcal{A}_{\mathcal{R}}$.

(b) For each $q' \in Q'$, for each $a \in \Sigma$ such that $q \in \delta'(q', a)$ and $q \in F'$, then $f \in \delta(q', a)$. Loosely speaking, each non λ-transition that in $\mathcal{A}$ reaches a final state (so a state in F'), is transformed into a transition that reaches f in $\mathcal{A}_{\mathcal{R}}$.

(c) Finally, let $S \subseteq Q'$ such that for each $q' \in S$, we have $\delta'^*(q', \lambda) \cap F' \neq \emptyset$. Let $q'' \in Q'$ such that $\delta'(q'', a) \cap S \neq \emptyset$, $a \in \Sigma$. Thus $f \in \delta(q'', a)$. This step allows us to eliminate the λ-paths in $\mathcal{A}$ which end in a final state, making the path to end in f.

By using suitable Depth-First-Search (DFS) visits, it is clear that this procedure can be done in $O(n)$ time [22]. As classically done for the flower automaton

[4], a variant of the direct product of $\mathcal{A}$ can be used for detecting suitable paths in order to decide the UD property [22]. For a better typesetting, from now on, $[p, q]$ denotes the pair of states (p, q).

Definition 4.2 *[22] Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, \{f\})$ be in restricted form. We define $\langle \mathcal{A} \times \mathcal{A} \rangle = (Q \times Q, \Sigma, \delta', [q_0, q_0], [f, f])$, where, for each $p, q \in Q$, $a \in \Sigma$,*

$$\begin{aligned} \delta'([p, q], a) &= \delta(p, a) \times \delta(q, a), \\ \delta'([p, q], \lambda) &= ((\delta(p, \lambda) \cup \{p\}) \times (\delta(q, \lambda) \cup \{q\})) - \{[p, q]\} \end{aligned}$$

As a consequence, there is a path in $\langle \mathcal{A} \times \mathcal{A} \rangle$ from $[p, q]$ to $[r, s]$ labeled x if and only if $\mathcal{A}$ contains paths from p to r and from q to s, both of which are labeled x. Observe that the λ-transitions defined above cannot regard two final states, being $\mathcal{A}$ in restricted form. In addition, if $[p, q]$ is a *semi-final state*, i.e., either $p = f$ or $q = f$, with $p \neq q$, then any transition out of $[p, q]$ must be a λ-transition to another semi-final state.

Let us consider $\langle \mathcal{A} \times \mathcal{A} \rangle$. Insert, for each $s \neq f$, an edge (λ-transition) from each of $[s, f]$ and $[f, s]$ to $[q_0, s]$. The directed graph obtained as above is denoted by $\hat{\mathcal{A}}$ and named here the *square automaton* of $\mathcal{A}$. It is easy to check that it can be constructed in $O(n^2)$ time.

Clearly, $\mathcal{A}$ can be deterministic and not restricted, and vice versa, i.e., $\mathcal{A}$ can be restricted and non-deterministic (see Fig.1). However, it is not difficult to check that if $\mathcal{A}$ is restricted and deterministic, the trim part of $\hat{\mathcal{A}}$, i.e., the subgraph containing all the accessible and coaccessible states, which is restricted, has the same size of $\mathcal{A}$. So, the known procedure in [4] based on unambiguous automata is as so efficient as McCloskey's algorithm.

Theorem 4.1 *[22] Let $\mathcal{A}$ be in restricted form. Then $L(\mathcal{A})$ is a code if and only if there are no paths in $\hat{\mathcal{A}}$ from $[q_0, q_0]$ to $[f, f]$ that pass through at least one semi-final state.*

Let us recall McCloskey's algorithm. Let $\mathcal{A}$ be a finite state automaton recognizing X. **(1)** Check whether $\lambda \in L(\mathcal{A})$. If it is so, X is not a code (by definition). This step can be implemented by using a DFS visit on (the graph underlying) $\mathcal{A}$, by considering only the paths labeled by λ. **(2)** Check whether $\mathcal{A}$ is in restricted form (this can be trivially done by using Definition 4.1). If it is not so, construct the restricted automaton $\mathcal{A}_{\mathcal{R}}$ equivalent to $\mathcal{A}$. **(3)** Construct the product automaton $\hat{\mathcal{A}}$ of $\mathcal{A}_{\mathcal{R}}$. **(4)** Find a path in $\hat{\mathcal{A}}$ from the initial state to the (unique) final state, which passes through a semi-final state in $\hat{\mathcal{A}}$. If such a path exists, X is not a code; otherwise, X is a code. Also this step can be implemented by using a DFS visit.

Steps 1-2 can be performed in time linear in the size of $\mathcal{A}$, whereas Steps 3-4 in time linear in the size of $\hat{\mathcal{A}}$, i.e., in $O(n^2)$ time [22].

5. DNA properties for codes

In the following, we suppose X to be a regular set and we denote by $\theta\colon \Sigma \to \Sigma$ an *involution* on Σ, i.e., satisfying $\theta(\theta(a)) = a$ for all $a \in \Sigma$. Two natural extensions on Σ^* can be considered. If for each $x, y \in \Sigma^*$, $\theta(xy) = \theta(x)\theta(y)$, θ is a *morphism*. If $\theta(xy) = \theta(y)\theta(x)$, then θ is an *antimorphism*. At the moment, as DNA properties we consider only the two basic intermolecular properties that follow.

Definition 5.1 *[14, 19] Let Σ be an alphabet and X be a regular language on Σ. Let θ be a morphism or antimorphism on Σ. X is θ-compliant if $\Sigma^*\theta(X)\Sigma^* \cap X = \emptyset$.*

Loosely speaking, the involution of a word in X is not a factor of an element of X.

Definition 5.2 *[14, 19] Let Σ be an alphabet and X be a regular language on Σ. Let θ be a morphism or antimorphism on Σ. X is θ-free if $\Sigma^*\theta(X)\Sigma^* \cap X^2 = \emptyset$.*

Loosely speaking, the involution of a word in X is not a factor of the concatenation of two elements of X. Clearly, if the factor appears inside one of the two words, then X is also not θ-*compliant*, either.

Proposition 5.1 *[14] Let θ be a morphic or antimorphic involution and let X be a non-empty subset of Σ^*. If X is θ-free, then X is θ-compliant.*

Remark 5.1. Let us consider Definitions 5.1-5.2. In [6, 19], the authors take into account the cases in which Σ^* is replaced by Σ^+, i.e., when only proper factors are allowed. Since here we use the θ-compliance property also for testing θ-k-hairpin property, we require this more general condition.

In the Examples 5.1-5.3 below, we use the Watson-Crick complementary relation as involution.

Example 5.1. [14] Let us consider the code $X = \{ACCAA, ACCGT\}$. It is easy to check that the DNA properties are both satisfied.

Example 5.2. Let us consider the code $X = \{TT, AGA\}$. It is easy to check that the θ-compliance property is satisfied. On the contrary, the word TT allows us to detect the failure of the θ-freedom property, since $\theta(TT) = AA$ and $AG\mathbf{AA}GA \in X^2$.

Example 5.3. Let $X = \{GA, CTG\}$. We have $\theta(X) = \{CT, GAC\}$ and $X^2 = \{GAGA, CTGCTG,\ GACTG, CTGGA\}$. The factor CT, involution of GA, makes X not θ-compliant (and so not θ-free). In addition, $GAC = \theta(CTG)$ and $\mathbf{GAC}TG = GA \cdot CTG \in X^2$, thus X is not θ-free.

Example 5.4. Let us consider again the restricted automaton of Fig. 1 recognizing the code $X = b(aba + ba)^*b$ and $\theta(X) = a(bab + ab)^*a$, where we fix the natural involution $\theta(a) = b, \theta(b) = a$. We have that X is not θ-compliant, since $bb\mathbf{aaba}b \in X$ and $aaba = \theta(bbab)$, with $bbab \in X$. In virtue of Proposition 5.1, we already know that X cannot be θ-free. We can check this directly, since $bab\mathbf{ababbaba}b = bababab \cdot babab \in X^2$ and $ababbaba = \theta(babaabab)$, with $babaabab \in X$.

Some closure properties are studied for these DNA properties, in order to prove that they are decidable for regular languages [14]. Precisely, the time complexity of the decision procedure is quadratic in the size of the finite state automaton recognizing X. Now, recall that in this context, X must also satisfy the UD property. Thus, we investigate efficient algorithms for testing DNA properties of regular codes, since known algorithms, such as the one in [23], works only for finite sets of words. In addition, they test the DNA properties by using a technique which is different from the one used for the UD property. On the contrary, here we apply to restricted automata a technique which is similar to the one used for CodeGen applied to flower automata [19]. In this way, the algorithm tests both UD and DNA properties by using the same structures and still remains within the same time complexity.

6. Testing intermolecular properties of an infinite set of DNA Words

The UD property does not guarantee that a regular language is useful for DNA computations. We also need that the molecules of the code do not hybridize together due to the presence of complementary factors inside the code. In [6] an algorithm for testing θ-compliance and θ-freedom properties is designed. Here we review it providing a proof of the result sketched in [6] and by using simpler definitions.

The algorithm is based on two automata designed for testing these DNA properties.

Definition 6.1 *[19] Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ be an NFA recognizing $X \subseteq \Sigma^*$ and let θ be a morphism or antimorphism on Σ. The* Theta-automaton *$\mathcal{A}_\theta$ of X is an NFA automaton $(\Sigma, Q, \delta', q_0, F)$ such that $p \in \delta(q, a)$ if and only if $p \in \delta'(q, \theta(a))$.*

Clearly $L(\mathcal{A}_\theta) = \theta(X)$ and $\mathcal{A}_\theta$ can be naturally constructed starting from $\mathcal{A}$ by replacing each letter a with $\theta(a)$. Obviously, if $\mathcal{A}$ is in restricted form, also $\mathcal{A}_\theta$ is in restricted form.

Let us consider a variant of Definition 4.2, where instead of considering only $\mathcal{A}$, we define the same operation between $\mathcal{A}_\theta$ and $\mathcal{A}$. In this way we can constuct $\langle \mathcal{A}_\theta \times \mathcal{A} \rangle$.

Definition 6.2 *Let $X \subseteq \Sigma^*$ be a regular language recognized by $\mathcal{A}$ in restricted form. Let $\mathcal{A}_\theta$ be the Theta-automaton of X naturally constructed starting from*

$\mathcal{A}$. *The* involution square automaton *of X is the automaton I_θ obtained starting with $\langle \mathcal{A}_\theta \times \mathcal{A} \rangle$ and by adding a λ-transition from each of $[q, f]$ to $[q, q_0]$, for each $q \neq f$.*

These λ-transitions allow us to simulate the concatenation of two paths which are "successful" w.r.t. $\mathcal{A}$. As in the case of $\langle \mathcal{A} \times \mathcal{A} \rangle$, in $\langle \mathcal{A}_\theta \times \mathcal{A} \rangle$ any transition out of a semi-final state must be a λ-transition to another semi-final state. In addition these out-transitions are only of the form given in Definition 6.2.

Similarly to CodeGen [19], we can check the DNA properties in this context as follows (definitions are simplified with respect to [6]). Firstly, we identify special paths in I_θ.

Definition 6.3 *Let $p, q, k \in Q$. A path π in the involution square automaton I_θ is*

- θ-compliant *if π starts in $[q_0, p]$, ends in $[f, q]$ and passes at least once through a semi-final state $[k, f]$, with $k \neq q_0$*
- θ-free *if π starts in $[q_0, p]$, ends in $[f, q]$ and passes at least twice through semi-final states $[k, f]$, with $k \neq q_0$*

In Theorem 6.1 we show that we can decide whether a code X is θ-compliant (resp. θ-free) by searching for θ-compliant (resp. θ-free) paths in I_θ. If there exists at least one path $[q_0, p] \to \cdots \to [f, q]$ which is not θ-compliant (resp. θ-free), then X is not θ-compliant (resp. θ-free).

Theorem 6.1 *Let X be a regular language on Σ and let $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ be in restricted form such that $X = L(\mathcal{A})$. The language X is θ-compliant (resp. θ-free) if and only if in the involution square automaton I_θ all paths π from $[q_0, p]$ to $[f, q]$, with $p, q \in Q$, are θ-compliant (resp. θ-free).*

Proof :
Let us firstly make the following observation. By definition, a θ-compliant path π_1 is a path that starts in $[q_0, p]$, ends in $[f, q]$ and passes at least once through a semi-final state $[k, f]$, i.e., $\pi_1 = [q_0, p] \to \cdots \to [k, f] \to [k, q_0] \to [f, q]$. We recall that in I_θ there is a λ-transition from $[k, f]$ to $[k, q_0]$. Let w_1 be the label of π_1. In particular, w_1 is accepted in $\mathcal{A}_\theta$, since w_1 is the label of the path $q_0 \to \cdots \to f$. In addition, $w_1 = w'w''$ such that, in $\mathcal{A}$, w' is the label of the path $p \to \cdots \to f$ and w'' is the label of the path $q_0 \to \cdots \to q$. Since p is accessible in $\mathcal{A}$, there exists a path in $\mathcal{A}$ from q_0 to p and since q is coaccessible in $\mathcal{A}$ there exists a path in $\mathcal{A}$ from q to f, i.e., $w'w''$ is a factor of a word accepted by $\mathcal{A}$. Furthermore, since $w'w'' \in L(\mathcal{A}_\theta)$ there exists $s \in X$ such that $\theta(s) = w'w''$. Recall that f cannot be an intermediate state in paths of $\mathcal{A}$ or $\mathcal{A}_\theta$, being these automata in restricted form.

Let us show that if I_θ contains only θ-compliant paths from $[q_0, p]$ to $[f, q]$, then X is θ-compliant. By contradiction, if X were not θ-compliant, by Definition 5.1, there should exist $z, z' \in X$ such that $z = xhy$ and $h = \theta(z')$. In other

words, there is a successful path in $\mathcal{A}_\theta$ labeled by h and a successful path in $\mathcal{A}$ labeled by $z = xhy$. Thus h is the label of a path in $\mathcal{A}_\theta$ and h is a factor of a successful path in $\mathcal{A}$. Let p, q be the initial and final state of the path labeled by h in $\mathcal{A}$. Recall that f cannot be an intermediate state in these paths. As a consequence, we should have a path in I_θ from $[q_0, p]$ and $[f, q]$ labeled by h, but $[k, f]$ cannot be an intermediate state, otherwise f is an intermediate state in the path labeled by h in $\mathcal{A}$. This is a contradiction, being all paths in I_θ θ-compliant.

Vice versa, let us now prove that if X is θ-compliant, then all paths from $[q_0, p]$ to $[f, q]$ are θ-compliant. By contradiction, suppose that I_θ contains a path from $[q_0, p]$ to $[f, q]$ which never passes through a semi-final state $[k, f]$. This path is labeled by a word $w \in L(\mathcal{A}_\theta)$, i.e., $\theta(w) \in X$, being θ an involution. Moreover, there exists a word $z = xwy \in X$ such that w is the label of a path in $\mathcal{A}$ from p to q. (This path from p and q does not pass through f, by hypothesis, and so w is an effective path of $\mathcal{A}$, and not a path obtained by adding a λ-transition to $\langle \mathcal{A}_\theta \times \mathcal{A} \rangle$). So X cannot be θ-compliant, a contradiction.

The case of θ-freedom works similarly, by taking care of the λ-transitions from each of $[q, f]$ to $[q, q_0]$ in the involution square automaton of X (see Definition 6.2). Let us suppose that all paths from $[q_0, p]$ to $[f, q]$ in I_θ are θ-free. We show that X is θ-free. By contradiction, let us suppose that X is not θ-*free*. By definition, there exist $x, y \in X$ and $w' \in X$, such that $xy = x'w_1w_2y' \in X^2$, with $x = x'w_1$, $y = w_2y'$, and $\theta(w') = w_1w_2$. Thus, $\theta(w') = w_1w_2$ is the label of a (successful) path in the Theta-automaton $\mathcal{A}_\theta$. The λ-transitions in the involution square automaton I_θ, added to $\langle \mathcal{A}_\theta \times \mathcal{A} \rangle$, allow us to paste the path labeled by w_1 (that ends in f in $\mathcal{A}$) with the path labeled by w_2 (that starts in q_0 in $\mathcal{A}$). Recall that we cannot find another occurrence of f as an intermediate state, by construction. Thus we find a path which is not θ-free in the involution square automaton, a contradiction.

Vice versa, let us now prove that if X is θ-free, then all paths from $[q_0, p]$ to $[f, q]$ in I_θ are θ-free. By contradiction, suppose that I_θ contains a path π from $[q_0, p]$ to $[f, q]$ which is not θ-free, i.e., π passes at most once through a semi-final state $[k, f]$. Two cases are possible. If π never passes through a semi-final state, then π is not a θ-compliant path and so X in not θ-compliant. As a consequence, by Proposition 5.1, X is not θ-free, a contradiction. Suppose that π passes once through a semi-final state. Then π is labeled by a word $w \in L(\mathcal{A}_\theta)$, i.e., $\theta(w) \in X$, being θ an involution. Moreover, by using a similar argument of the proof for the θ-compliance property, we can show that there exist w', w'' such that w is a factor of $w'w''$, with $w', w'' \in X$, a contradiction, being X is θ-free. ∎

Remark 6.1. By definition, if in I_θ there is no path from $[q_0, p]$ to $[f, q]$, the language X is θ-free (and so θ-compliant).

Remark 6.2. We observe that the test is effective, even if the involution square automaton contains loops, since we do not check the label of these paths (which are clearly infinite), but only the states of the paths.

Example 6.1. (Example 5.1 continued). Let us consider the restricted finite state automaton recognizing $X = \{ACCAA, ACCGT\}$, whose paths are $q_0 \xrightarrow{A} q_1 \xrightarrow{C} q_2 \xrightarrow{C} q_3$, and $q_3 \xrightarrow{A} q_4 \xrightarrow{A} f$ and $q_3 \xrightarrow{G} q_5 \xrightarrow{T} f$. Let us consider the Theta-automaton of X and the involution square automaton of X. We can easily check that $[q_0, q_5]$ is the unique non isolated state of the form $[q_0, p]$. The unique path starting from $[q_0, q_5]$ is $[q_0, q_5] \to [q_1, f] \to [q_1, q_0]$ labeled by $T\lambda$. Since there is no path from $[q_0, q_5]$ to a state of the form $[f, q]$, we have that X is θ-free (and so θ-compliant).

Example 6.2. (Example 5.2 continued) Figure 2 shows the automaton recognizing $X = \{TT, AGA\}$ (on the right) and the automaton recognizing $\theta(X)$ (on the left). They have been depicted in this order to help the reader for the construction of I_θ. We can easily check that, in the involution square automaton of X, there exists the path $[q_0, q_3] \to [q_1, f] \to [q_1, q_0] \to [f, q_2]$ labeled by $A\lambda A$, which is not θ-free, since it passes once through a state of the form $[k, f]$, $k \neq q_0$. (This is exactly the factor AA detected in the Example 5.2.) So X is not θ-free. All paths are θ-compliant, so X is θ-compliant.

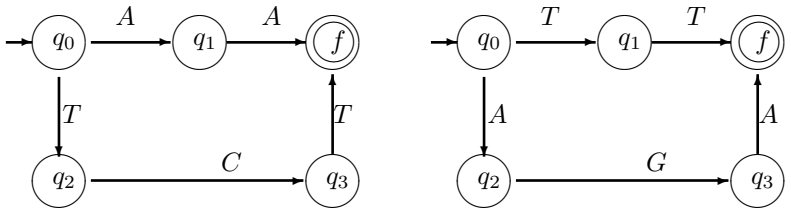


Fig. 2: (left) The Theta-automaton $\mathcal{A}_\theta$ constructed from the reduced automaton recognizing $X = \{TT, AGA\}$ (on the right)

Example 6.3. (Example 5.3 continued) Figure 3 shows the automaton recognizing $X = \{GA, CTG\}$ (on the right) and the automaton recognizing $\theta(X)$ (on the left). We can easily check that, in the involution square automaton of X, there exists the path $[q_0, q_0] \to [q_1, q_2] \to [f, q_3]$ labeled by CT, which is not θ-compliant, since it does not pass through a state of the form $[k, f]$, $k \neq q_0$. (This is exactly the factor CT detected in the Example 5.3.) So X is not θ-compliant. Clearly, this path is not θ-free, and so it allows to say that X is also not θ-free. Also the path $[q_0, q_0] \to [q_2, q_1] \to [q_3, f] \to [q_3, q_0] \to [f, q_2]$ (labeled $GA\lambda C$) is not θ-free.

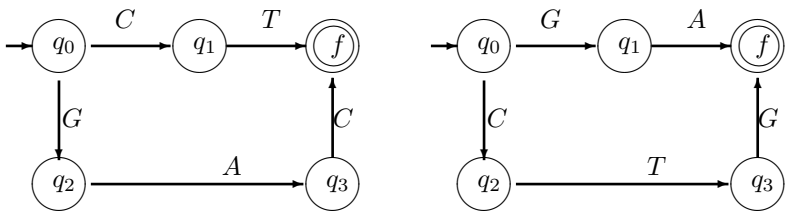


Fig. 3: (left) The Theta-automaton $\mathcal{A}_\theta$ constructed from the reduced automaton recognizing $X = \{GA, CTG\}$ (on the right)

Example 6.4. (Example 5.4 continued) We already know that the code $X = \{b(aba + ba)^*b\}$ is not θ-compliant. Let us consider the construction of the involution square automaton of X, starting from the restricted automaton of Figure 1. We can check that the path $[q_0, q_3] \to [q_1, q_1] \to [q_3, q_2] \to [q_1, q_3] \to [f, q_1]$ labeled by $aaba$ is not θ-compliant.

In addition, we already know that X is not *θ-free*. We can check that the path $[q_0, q_3] \to [q_1, q_1] \to [q_2, q_3] \to [q_3, q_1] \to [q_1, f] \to [q_1, q_0] \to [q_2, q_1] \to [q_3, q_2] \to [q_1, q_3] \to [f, q_1]$ labeled by $abab\lambda baba$ is not θ-free.

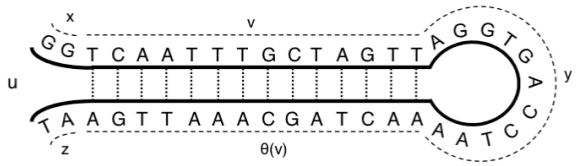
7. Testing an intramolecular property: θ-k-hairpin-freeness

In the previous sections we have considered two basic unwanted hybridizations that can happen between two molecules. In the sequel we analyze a kind of hybridization (hairpin) that regards a molecule and itself [16, 17, 18]. Basically, a language is "hairpin-free" if none of its words contains a factor which is the involution of another factor of the same word, with respect to a given involution. Otherwise, the molecule is wound on itself in correspondence of the two factors that bind due to the complementarity (see figure below). It is evident that it is not worth expecting a language to be totally free of such words. For instance, let us consider short factors, even of length one. Even if we could formally have easy hairpin formations, hairpins could not be generated due to the the weakness of the biochemical conditions. Thus, the literature deals with the θ-k-hairpin-freeness property, also defined $hp(\theta, k)$-hairpin-freeness, relating hairpin-freeness property to a morphic or antimorphic involution θ and an integer number $k \geq 1$.

Definition 7.1 *[16] Let $k \geq 1$ and θ be a morphic or anti-morphic involution on Σ. A word $u \in \Sigma^*$ is θ-k-hairpin-free, or $hp(\theta, k)$-free, if $u = xvy\theta(v)z$ implies $|v| < k$. A language $L \subseteq \Sigma^*$ is θ-k-hairpin-free if all its words are θ-k-hairpin-free.*

For example, when θ is the Watson-Crick involution, a word u is θ-k-hairpin-free if the length of the complementary factors is at most k.

Example 7.1. Let $k = 3$, $\Sigma = \{A, G, T, C\}$ and let θ be the Watson-Crick involution τ. The word $w = AATTGTCAAGT$ is not $hp(\tau, 3)$-free. Indeed, its factors TTG and CAA can generate an hairpin, since $\tau(TTG) = AAC^{-1} = CAA$. It is also easy to see that w is $hp(\tau, k)$-free for each $k > 3$, since there are no factors longer than three symbols which can generate hairpins.



Observe that the value k is an upper bound for the length of factors that can be involution of others, without loosing hairpin-freeness property. Clearly, the empty word λ, as well as a word on a unary alphabet, is $hp(\theta, 1)$-free.

Proposition 7.1 *[16] Let $u \in \Sigma^*$. If $|u| < 2k$, then u is $hp(\theta, k)$-free.*

Definition 7.2 *[16] We set $HpFree(\theta, k) = \{w \in \Sigma^* | w$ is θ-k-hairpin-free$\}$, and by $Hp(\theta, k)$ its complement with respect to Σ^*.*

If we want to check the $hp(\theta, k)$-hairpin-freeness rather than a more generic hairpin-freeness property, we choose the most appropriate value for k, depending on the number of molecular bases which are biochemically necessary in order to generate hairpin, or depending on other characteristics the application case needs. In [16], the problem of deciding θ-k-hairpin-freeness property for a given regular language L was proved to be decidable, by using the following algorithm.

1. Build an NFA $\mathcal{A}_k$ recognizing the language $Hp(\theta, k)$
2. Build an NFA $\mathcal{A}'$ recognizing the language $L \cap Hp(\theta, k)$
3. Check whether in $\mathcal{A}'$ there is a path from the initial state to a final state: if such a path exists, L is not θ-k-hairpin-free, otherwise it is θ-k-hairpin-free.

In this way, if there exists a word w accepted by $\mathcal{A}'$, we have that w belongs to L and to the language containing all words characterized by hairpin, because the factors are long at least k. Then L has at least one word generating an hairpin among at least two of its factors of length at least k, i.e., L is not $hp(\theta, k)$-free. In order to execute the above algorithm, the required time is $O(|\mathcal{A}'|) = O(|\mathcal{A}| \cdot |\mathcal{A}_k|)$, since we need to build and check paths of the NFA $\mathcal{A}'$, which is as the product of two automata. The above naive algorithm requires an automaton that accepts all words which are not $hp(\theta, k)$-free on a given alphabet and its size grows exponentially with k. On the contrary, we develop a direct technique to verify this property.

7.1. The proposed algorithm

We check the θ-k-hairpin-freeness property of regular languages by using the same idea explained in Section 6 for deciding intermolecular properties. Even in this case, indeed, we build product automata as theta and involution square automata, before analyzing particular paths in the obtained automata. Let us recall that the hairpin-freeness property needs comparisons among fragments of a given word, since hairpin is an intramolecular phenomenon. Moreover, θ-k-hairpin-freeness needs these fragments to be of length at least k. Thus, we propose an algorithm that firstly builds an NFA accepting only the factors long at least k of the words accepted by the input automaton, that is, the automaton recognizing the language to check; then, it works on the obtained automaton and its corresponding Theta-automaton.

Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, \{f\})$ be an NFA in restricted form, L be the language accepted by $\mathcal{A}$ and $k \geq 1$. We denote by $\mathcal{F}_k = (\Sigma, Q, \delta_{Fk}, q_0, \{f\})$ the automaton recognizing the factors of length at least k of $\mathcal{A}$, i.e., $L(\mathcal{F}_k) = Fact_k(L)$. Given $\mathcal{F}_k$, we denote by $\mathcal{F}_{k\theta}$ the Theta-automaton of $\mathcal{F}_k$, i.e., $L(\mathcal{F}_{k\theta}) = \theta(Fact_k(L))$.

We adapt the paths checking proposed for the θ-compliance property, in order to make it suitable for the new approach that requires a factorization of the analyzed words. We firstly identify the θ-compliant paths inside the involution square automaton. In the hairpin-freeness case, θ-compliant paths are associated with factors of the words of the input language L and not just to words of L. We build the involution square automaton as the product of $\mathcal{F}_k$ of L and its Theta-automaton $\mathcal{F}_{k\theta}$. To avoid ambiguity, we can define a new type of involution square automaton, which is effectively built to test θ-k-hairpin-freeness property.

Definition 7.3 *Let θ be a morphic or antimorphic involution, $\mathcal{A}$ be an NFA in restricted form recognizing L, $\mathcal{F}_k = (\Sigma, Q, \delta_{Fk}, q_0, \{f\})$ such that $L(\mathcal{F}_k) = Fact_k(L)$ and $\mathcal{F}_{k\theta} = (\Sigma, Q, \delta'_{Fk}, q_0, \{f\})$ the corresponding Theta-automaton. We define $\langle \mathcal{F}_{k\theta}, \mathcal{F}_k \rangle$ as the product $\mathcal{F}_{k\theta} \times \mathcal{F}_k$, where, for each $x \in \Sigma$ and $i, p \in Q$, we add*

$$\delta([i,p],x) = \delta'(i,x) \times \delta(p,x)$$

$$\delta([i,p],\lambda) = ((\delta'(i,\lambda) \cup \{i\}) \times (\delta(p,\lambda) \cup \{p\})) - (\{i,p\})$$

The involution square automaton *of $\mathcal{F}_k$, denoted $I_{Fk\theta}$, is obtained starting with $\langle \mathcal{F}_{k\theta}, \mathcal{F}_k \rangle$ by adding a λ-transition from each state $[q,f]$ to each state $[q,q_0]$, with $q \neq f$.*

Remark 7.1. The λ-transitions added to $\langle \mathcal{F}_{k\theta}, \mathcal{F}_k \rangle$ are added with the same motivation already explained for the θ-compliance case (see Section 6).

The algorithm is based on the theorem below. Briefly, if L is θ-k-hairpin-free, two cases can occur in $I_{Fk\theta}$: either we will never find a path π from a state $[q_0,p]$ to a state $[f,q]$, $p,q \in Q$, which is θ-compliant, i.e., there is no word in L which contains a factor and its θ involution (otherwise the language $Fact_k(L)$

should have been not θ-compliant), or there is a path π from a state $[q_0, p]$ to a state $[f, q]$, $p, q \in Q$, which is θ-compliant, but this path regards two factors of L which belong to two different words of L, i.e., w and $\theta(w)$ are labels which do not belong to a same accepting path in $\mathcal{A}$.

Theorem 7.1 *Let θ be an involution on Σ, $k \geq 1$, let $L \subseteq \Sigma^*$ be a regular language on Σ and $\mathcal{A} = (\Sigma, Q, \delta, q_0, \{f\})$ an automaton accepting L in restricted form. L is θ-k-hairpin-free if and only if, in the involution square automaton $I_{Fk\theta}$, every path π from a state $[q_0, p]$ to a state $[f, q]$, $p, q \in Q$, either is θ-compliant or π is labeled by a word w such that $\theta(w)$ is not on a same accepting path in $\mathcal{A}$.*

Proof :
By definition, if a regular language L is θ-k-hairpin-free, there is no word which contains two factors of length at least k such that one factor is the θ-involution of the other, i.e., for each accepting path in $\mathcal{A}$, we can never read a label of length k and its involution, or vice versa. So by definition, there exists no word $w \in Fact_k(L)$ and $z = hwy\theta(w)v \in L$.

Now, let us consider in $I_{Fk\theta}$ all paths of the form $\pi = [q_0, p] \rightarrow \cdots \rightarrow [f, q]$. Two cases can occur. If all these paths are θ-compliant, in virtue of Theorem 6.1, $\mathcal{F}_{k\theta}$ is θ-compliant, i.e., there exist no words w, w' such that $w \in \mathcal{F}_{k\theta}$ and $w = x\theta(w')y$. In other words, L is θ-k-hairpin-free since there exist no complementary factors. On the other side, if there exists a path which is not θ-compliant, its label and its complement cannot be read on the same accepting path of $\mathcal{A}$ too. Otherwise, $z = hwy\theta(w)v \in L$, a contradiction.

Conversely, let us consider $I_{Fk\theta}$ and suppose that all paths π from a state $[q_0, p]$ to a state $[f, q]$, $p, q \in Q$, are θ-compliant. It is easy to see that this implies that $\mathcal{F}_k$ and $\mathcal{F}_{k\theta}$ both recognize θ-compliant languages, respectively $Fact_k(L)$ and $\theta(Fact_k(L))$. Being $Fact_k(L)$ θ-compliant, no factor w of length at least k is such that $\theta(w)$ belongs to $Fact_k(L)$, i.e., there exists no word $z = hwy\theta(w)v \in L$, with $w \in Fact_k(L)$. This implies that L is θ-k-hairpin-free.

Now, suppose that there exists at least one path $\pi = [q_0, p] \rightarrow \cdots \rightarrow [f, q]$ in $I_{Fk\theta}$ which is not θ-compliant. L can still be θ-k-hairpin-free, because the presence of a non θ-compliant path π may be due to a correspondence between an involution $\theta(v_1)$ of a factor v_1, of length at least k, for a word $u_1 \in L$ and a factor v_2 of a word $u_2 \in L$, that is, a correspondence of factors of two words $u_1 \neq u_2$, i.e., v_1 and v_2 are labels that cannot be read on the same accepting path. This correspondence does not mean that θ-k-hairpin-freeness is not verified, since θ-k-hairpin-freeness concerns single words and not a comparison of distinct words of L. So, let us consider the case of a non θ-compliant path in $I_{Fk\theta}$, $\pi_1 = [q_0, p] \rightarrow \cdots \rightarrow [f, q]$, labeled by w_1, such that $\theta(w_1)$ is not readable on a same accepting path of $\mathcal{A}$ on which we read w_1. Clearly, if two sequences of symbols do not share an accepting path in a given automaton, these sequences cannot be factors of a same word accepted by the automaton. Thus, by hypothesis, w_1 and $\theta(w_1)$ are not factors of a same word recognized by $\mathcal{A}$, and so they cannot generate hairpin. If L is composed by a word generating a hairpin because of

a factor long at least k, this factor is accepted by $\mathcal{F}_k$, while the corresponding θ involution is accepted by $\mathcal{F}_{k\theta}$. So there exist a non θ-compliant path of type $[q_0, p] \to \cdots \to [f, q]$ in $I_{Fk\theta}$, and one of such a path for each factor long at least k and potential hairpin generator. In addition, if all of these factors belong to words in L which do not contain also their θ-involutions, since they do not share the same accepting path in $\mathcal{A}$, all of the factors cannot generate hairpin. Formally, there exists no accepting path with label $z = hw_1y\theta(w_1)v \in L$. Thus, L is θ-k-hairpin-free. ∎

Theorem 7.1 suggests the algorithm below, where the input is an NFA $\mathcal{A}$ in restricted form recognizing L and the output is NO, if L is not θ-k-hairpin-free, YES, otherwise.

Algorithm for deciding θ-k-hairpin-freeness

1. Build the automaton $\mathcal{F}$ accepting all the factors of the words accepted by $\mathcal{A}$.
2. Build the automaton $\mathcal{F}_k$ accepting all the words in $\mathcal{F}$ of length at least k
3. Build the Theta-automaton $\mathcal{F}_{k\theta}$ of $\mathcal{F}_k$
4. Build the involution square automaton $I_{Fk\theta}$
5. (*) For each path π from $[q_0, p]$ to $[f, q]$ in $I_{Fk\theta}$, check if it passes through at least one semi-final state $[k, f]$:
   (a) if it includes such a state, continue to check the other paths. If all paths from $[q_0, p]$ to $[f, q]$ have already been analyzed, return YES;
   (b) otherwise, check if the word labeling π and its θ-involution are on a same accepting path in $\mathcal{A}$: if it is so, return NO; otherwise, go to (*).

The first step of the algorithm requires building an automaton $\mathcal{F}$ recognizing all the factors of the language accepted by $\mathcal{A}$, that must necessarily be in restricted form. In the process of building $\mathcal{F}$, we need to maintain the restricted form of $\mathcal{A}$, to avoid an excessive growth of the size of the obtained automaton. A naive procedure to build $\mathcal{F}$ needs (a) to copy all the states and the transitions of $\mathcal{A}$ in $\mathcal{F}$; (b) for each transition t of $\mathcal{A}$ not ending in the final state of $\mathcal{A}$, add another transition in $\mathcal{F}$ sharing the same origin state and label of t but ending in the final state of $\mathcal{F}$; (c) for each state s of $\mathcal{F}$, excluding its final and initial states, add a transition in $\mathcal{F}$ such that it starts in the initial state of $\mathcal{F}$ and ends in s labeled by λ. Time complexity of the above procedure is $O(n)$, if n is the size of the input automaton $\mathcal{A}$. Effectively, it just requires mainly a visit of all states and transitions of $\mathcal{A}$. It is worth noting that the obtained automaton $\mathcal{F}$ is still in restricted form: no new final state is added, nor λ-transitions ending in the final state, nor λ-transitions starting from the final state.

The second step requires the construction of the automaton $\mathcal{F}_k$, accepting factors of length at least k of $L = L(\mathcal{A})$. Simply, $\mathcal{F}_k$ can be obtained as the intersection of $\mathcal{F}$ and an automaton $A_{\Sigma k}$ accepting all words on Σ of length at least k. Such an intersection maintains the restricted form of $\mathcal{F}$, since it is done making a product of two automata which are in restricted form and by removing those states and transitions which are not part of any accepting path,

i.e., which are not useful for the next operations. Unfortunately, the output automaton grows along with the growth of the size of the alphabet Σ, which determines the transitions for $A_{\Sigma k}$. However, this alphabet is small in many application cases, e.g., $|\Sigma| = 4$. Time complexity of second step is $O(n \cdot k \cdot m)$, where $m = |\Sigma|$. Indeed, we build a product of $\mathcal{F}$ and $\mathcal{A}_{\Sigma k}$, where $\mathcal{A}_{\Sigma k}$ includes $k + 1$ states and $(k + 1) \cdot m$ transitions.

The third step builds $\mathcal{F}_{k\theta}$, the Theta-automaton of $\mathcal{F}_k$. Simply, this can be done by substituting the labels of $\mathcal{F}_k$ with the corresponding θ involutions and making little changes to the transitions, such that time complexity of the step is $O(|\mathcal{F}_k|)$, that is, $O(n \cdot k \cdot m)$. Observe that, in general m is smaller than n, e.g., $m = |\Sigma| = 4$, k is fixed depending on the needs, but is rarely higher than n, otherwise molecular encoding of data could get very hard to handle.

7.2. Paths analysis

The fourth step allows us to obtain the automaton used to verify the θ-k-hairpin-freeness. It can be implemented building a product automaton $\langle \mathcal{F}_{k\theta} \times \mathcal{F}_k \rangle$ for obtaining $I_{Fk\theta}$, by adding some transitions, whose number may be ignored. Indeed, let n be the size of $\mathcal{A}$ and $m = |\Sigma|$. Then, the added transitions would be $\sqrt{n}$, because one of them is added for any of the $\sqrt{n}$ states $[q, f]$. Similarly, for $I_{Fk\theta}$, this would cost $O(|\mathcal{F}_{k\theta}| \cdot |\mathcal{F}_k| + \sqrt{n})$=$O(n^2 \cdot k \cdot m)$. Again, k and m are smaller than n, so their value is negligible for time complexity analysis.

The fifth step tests if $\mathcal{A}$ recognizes an hairpin-free language for k long factors, in relation to a given involution θ. In fact this step can be implemented in such a way that only a finite number of paths is analyzed, since we examine states and not labels (see Section 7.3 and the implementation). In order to verify this, we can trace the states we meet in every path between states $[q_0, p]$ and $[f, q]$, inserting them in a list *Connected*. If *Connected list* has no state, the algorithm returns YES: there exists no $[q_0, p] \to \cdots \to [f, q]$ path to analyze. Otherwise, for each $([q_0, p], [f, q])$, we can verify θ-compliance for every path between the current $[q_0, p]$ and the current $[f, q]$ inside automaton $I_{Fk\theta}$. Assuming $[q_0, p] \to \cdots \to [f, q]$ is a path for a given pair, this analysis performs these steps:

a) start from $[q_0, p]$, trace the sequence of characters met in a *Word list* and go through only the states of the *Connected list* which are not still included in the current path, until a semi-final state $[k, f]$ or the state $[f, q]$ is met

b) if a semi-final state $[k, f]$ is met, the path is θ-compliant. Thus check the other paths starting again from step (a). If there are no such paths, change the pair $([q_0, p], [f, q])$ and, if there is no more pair to analyze, the algorithm returns YES. Indeed, all paths $[q_0, p] \to \cdots \to [f, q]$ of $I_{Fk\theta}$ have been analyzed and either they are θ-compliant or their labels have been not found on the same accepting path of their involution in $\mathcal{A}$. So the language is θ-k-hairpin-free.

c) On the contrary, if the chosen state $[f, q]$ is met, the analyzed path is ended and it is non θ-compliant, so we have to verify its label w. The

label w of a non θ-compliant path is included in the *Word list*. We can check if $\theta(w)$, in automaton $\mathcal{A}$, can be read on a same accepting path in $\mathcal{A}$. This can be done by visiting states and transitions in $\mathcal{A}$, as follows

d) from a state in $\mathcal{A}$ having an outgoing transition labeled by the first character of w, from which the analysis of w never started before, start a visit of $\mathcal{A}$ and continue analyzing the transitions. Stop when the remaining letters of w are all sequentially met in the same order they compose w

e) if the exact sequence of characters of w is met, start a search for $\theta(w)$ in the same way w has been searched, considering only states and transitions included in the path $q_0 \to \cdots \to f$ in $\mathcal{A}$ where w has been found:

   I. if $\theta(w)$ is not found, w and $\theta(w)$ are factors of two distinct words of the language, then go to step (b);

   II. if $\theta(w)$ is found, return NO, since we have found a factor whose θ involution is readable on the same accepting path in $\mathcal{A}$. Thus, the language is not θ-k-hairpin-free

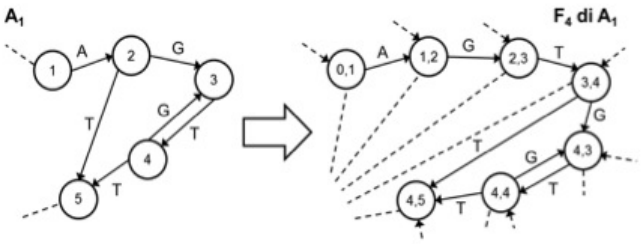
f) if the exact sequence of characters of w is not found, go to step (d), starting the search from a different transition.

In the above steps for any pair $([q_0, p], [f, q])$, we analyze all factors of length at least k which can generate hairpins. Effectively, we need $O(n^2)$ to calculate the states reachable from $[q_0, p]$ and $[f, q]$, by performing a DFS or a BFS visit of the graph representing $I_{Fk\theta}$, whose size is n^2. Moreover, for every path, we can check the θ-compliance property in $O(n^3)$. Actually, assuming n_s as the number of states and n_t as the number of transitions of $\mathcal{A}$, we need to check any path between $[q_0, p]$ and $[f, q]$ in $I_{Fk\theta}$. There are $\sqrt{n}_s$ states of the form $[q_0, p]$ and $[f, q]$. In the worst case, we have to check all paths of $n_s = \sqrt{n}_s \cdot \sqrt{n}_s$ pairs of type $([q_0, p], [f, q])$. We know that $I_{Fk\theta}$ is a product automaton built with $\mathcal{F}_{k\theta}$ and $\mathcal{F}_k$. Thus, for any pair $([q_0, p], [f, q])$, we can have at most $m \cdot n_s^2 \cdot n_t^2$ paths. Indeed, each state $[q_0, p]$ may have at most n_s^2 adjacent states, which may have at most m ingoing distinct transitions from $[q_0, p]$. Nevertheless, regarding the remaining of the path to $[f, q]$ we can have at most other $n_t^2 - 1$ transitions to check, being $I_{Fk\theta}$ a product $F_{k\theta} \times F_k$, so it has n_s^2 states and n_t^2 transitions. Multiplying this complexity for the total number of $([q_0, p], [f, q])$ pairs, the total number of paths costs $O(n \cdot m \cdot n_s^2 \cdot n_t^2)$. Obviously, in the worst case, we have to add the complexity of the searches of all the words of the analyzed path. This costs $O(|\mathcal{A}|) = O(n)$, since in the worst case we have to visit twice the whole automaton $\mathcal{A}$. The worst case is rather much rare for most applications. It is indeed very rare to need to work with languages whose involution square automaton has a high number of paths between $[q_0, p]$ and $[f, q]$ states. For many languages, these paths are just a few and very short and many of them are θ-compliant, so their analysis is very fast and, oftentimes, it is not necessary to search their associated words. Average case analysis reveals a much more efficient algorithm.

7.3. Advantages and other details

The main advantage of the proposed algorithm is that we can verify if a regular language is a good code, without any difference between finite or infinite sets of words. Our method is based on the analysis of a limited number of paths inside automata easily obtained from the input automaton, rather than a direct analysis of the language.

Finiteness of the total number of verified paths has two reasons. First of all, we maintain a finite number of transitions and states of all the automata, since we use intersection, product and other operations on NFA which preserve the finiteness of input automaton. In addition, the algorithm avoids cycles, which could let the number of analyzed paths grow infinitely. For instance, step (a) of the verifying procedure does not visit twice the states that have been already visited for the current path. Involution square automaton is obtained after an intersection between a factor automaton and the automaton recognizing words of length at least k. This intersection saves the cycles of the input automaton in such a way that the θ-k-hairpin-freeness is still decidable. More precisely, each of these cycles is transformed in the union of an acyclic union of k or more states and transitions and only one cycle (like a frying-pan shape). Only the acyclic part is taken in account analyzing paths in $I_{Fk\theta}$. As an example, let us consider $L_1 = AA(GT)^*TACAC$, accepted by $\mathcal{A}_1$ as showed in the figure below. Let us choose $k = 4$, $\Sigma_1 = \{A, T, G, C\}$ and the Watson-Crick involution as θ. L_1 is clearly not $hp(\theta, 4)$-free, because some of its words have $GTGT$ and $ACAC$ as factors (of length 4), and one is the involution of the other. This peculiarity can be found by repeating the sequence GT at least once and following the correspondent cycle at least twice in the automaton $\mathcal{A}_1$ that recognizes L_1. After building the factor automaton $\mathcal{F}$ and then $\mathcal{F}_4$ with an intersection, the cycle is transformed as shown in the figure.



We can see that the constrution of $\mathcal{F}_4$ takes the portion of $\mathcal{A}_1$ associated with GT^*. We have that the path $2 \to 3 \leftrightarrow 4 \to 5$ has a cycle and can be extended in a longer sequence composed by a first acyclic part $[1,2] \to [2,3] \to [3,4] \to [4,3] \to [4,4] \to [4,5]$ and a second cyclic part $[4,3] \leftrightarrow [4,4]$. This is clearly reflected in the involution square automaton. More specifically, analyzing only the first acyclic part is enough to analyze the sequence $GTGT$, that reveals that L_1 is not $hp(\theta, 4)$-free. Details can be found in the implementation of the algorithm.

8. Conclusion

In this paper we have faced the problem of detecting some undesiderable bonds between DNA molecules which are used in the design of good DNA encodings. We have proposed a technique for testing an infinite (regular) set of words instead of a (large amount of) finite words. The main advantage is that there is a unified approach for testing UD property, intermolecular and intramolecular properties. For the UD property, we have used McCloskey's algorithm, and for the DNA properties we have developed a technique inspired by CodeGen, but designed in this new context. We have considered an extension from finite languages to regular languages since we cannot decide the UD property for context-free languages yet [4].

The algorithm has also been implemented using Java as programming language (JDK 1.7 and IDE Eclipse). The object oriented aspect of such language is often an advantage in the choice of the architecture and the composition of the classes. Nevertheless, there are many aspects for which such choice should be handled with care. The natural choice of implementing automata as graphs represented by adjacency lists (and so by using for example java.util.AdjacencyListGraph) has been considered (firstly in [8] only for the UD property), but discarded here for another representation that represents a good compromise between run-time and memory usage.

Clearly, the pitfall of the algorithm is the construction of the (involution) square automata, due to the cardinality of the set $Q \times Q$. Briefly, we describe $\mathcal{A}$ by building three parallel arrays V_1, V_2, L as follows: for each edge (q_i, a, q_j), insert q_i into V_1, insert q_j into V_2, insert a into L (precisely, we insert the integer index of these states). This means that for each index i, $(V_1[i], L[i], V_2[i])$ is an edge of $\mathcal{A}$. This simple representation has many advantages in the construction of the square automaton. In fact, given the states $q_0, q_1, \ldots, q_{|Q|-1}$ of $\mathcal{A}$, we obtain the new states $q'_0, q'_1, \ldots, q'_{(|Q|-1)\times(|Q|-1)}$ of the (involution) square automaton, as follows. By using the representation based on the three arrays, given q_i, q_j two states of $\mathcal{A}$, we can calculate immediately the index k of corresponding state $q'_k = [q_i, q_j]$ into the (involution) square automaton as follows: $k = (i \times |Q|) + j$. Our Java implementation takes care of the memory space requirement, by involving primitive data types.

Future investigations regard both theory and implementations, e.g., a more space efficient implementation. We are examining the possibility of testing other properties with this technique, for example, absent words (i.e., given a text, find all shortest words that do not appear as factors in the text) [10]. Another interesting extension of this investigation may be to study the DNA codeword design problem when also partial matches are allowed, instead of perfect involutions. Indeed, as said in the introduction, hybridizations can occur also when there is not a perfect correspondence between bases. However, the main issue is to define all these approaches in terms of circular words. Indeed, recent investigations prove the existence of circular codes in particular genes [2] and confirm that a DNA computation performed on circular molecules, instead of linear molecules, may be more efficient [11].

Acknowledgements

We wish to thank an anonymous referee of [6] for encouraging a more depth investigation on detection of coding properties for DNA computing.

References

[1] A. Adleman. Molecular computation of solution of combinatorial problems. *Science*, 266:1021–1024, 1994.

[2] Arques and Michel. A complementary circular code in the protein coding genes. *J. Theor. Biol.*, 182:45–58, 1996.

[3] M.-P. Béal and D. Perrin. Codes, unambiguous automata and sofic systems. *Theor. Comput. Sci.*, 356(1-2):6–13, 2006.

[4] J. Berstel, D. Perrin, and C. Reutenauer. *Codes and Automata, Encyclopedia od Mathematics and its Applications.* Number 129. Cambridge University Press, 2009.

[5] D. Cheptea, C. Martin-Vide, and V. Mitrana. On the hairpin completion of regular languages. In *ICTAC 2009*, volume 5684 of *Lect. Not. in Comp. Sci.*, pages 170–184. Springer-Verlag, 2009.

[6] M. Cianciulli, R. Zaccagnino, and R. Zizza. An easy automata based algorithm for testing coding properties of infinite sets of (dna) words. In *TPNC 2012*, volume 7505 of *Lect. Not. in Comp. Sci.*, pages 121–132. Springer-Verlag, 2012.

[7] T. Colcombet. Forms of determinism for automata (invited talk). In *STACS 2012*, pages 1–23, 2012.

[8] L. D'Auria and R. Zizza. A note on the McCloskey's algorithm for deciding whether a regular language is a code, 2010. ICTCS2010, Camerino, Italy.

[9] Max H. Garzon and Russell J. Deaton. Codeword design and information encoding in DNA ensembles. *Natural Computing*, 3(3):253–292, 2004.

[10] J. Harold, S. Kurtz, and R. Giegerich. Efficient computation of absent words in genomic sequences. *BMC Bioinformatics*, 9:167, 2008.

[11] T. Head, G. Rosenberg, R.S. Bladergroen, C.K.D. Breek, P.H.M. Lommerse, and H.P. Spaink. Computing with DNA by operating on plasimids. *Biosystems*, 57:87–93, 2000.

[12] T. Head and A. Weber. Deciding code related properties by means of finite transducers. In R.M. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II*, pages 260–272. Springer-Verlag, 1993.

[13] J.-E. Hopcroft, R. Motwani, and J.-D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 2001.

[14] S. Hussini, L. Kari, and S. Konstantinidis. Coding properties of DNA languages. *Theor. Comput. Sci.*, 290(3):1557–1579, 2003.

[15] N. Jonoska and K. Mahalingam. Languages of DNA based code words. In J. Chen and J. Reif, editors, *DNA9*, Lect. Not. in Comp. Sci., pages 61–73. Springer-Verlag, 2003.

[16] L. Kari, S. Konstantinidis, E. Losseva, P. Sosik, and G. Thierrin. A formal language analysis of dna hairpin structures. *Fundamenta Informaticae*, 71:453–476, 2006.

[17] L. Kari, S. Konstantinidis, E. Losseva, P. Sosik, and G. Thierrin. Hairpin structures in DNA words. In *DNA11*, volume 3892 of *Lect. Not. in Comp. Sci.*, pages 158–170. Springer-Verlag, 2006.

[18] L. Kari, S. Konstantinidis, P. Sosik, and G. Thierrin. On hairpin-free words and languages. In *DLT2005*, volume 3572 of *Lect. Not. in Comp. Sci.*, pages 296–307. Springer-Verlag, 2005.

[19] D.-E. Kephart and J. LeFevre. CODEGEN: The generation and testing of DNA code words. In *IEEE Congress on Evolutionary Computation*, 2004.

[20] F. Manea, T. Yokomori, and V. Mitrana. Two complementary operations inspired by the dna hairpin formation: Completion and reduction. *Theor. Comput. Sci.*, 410(4-5):417–425, 2009.

[21] G. Mauri and C. Ferretti. Word design for molecular computing: a survey. In *DNA9*, volume 2943 of *Lect. Not. in Comp. Sci.*, pages 37–47. Springer-Verlag, 2004.

[22] R. McCloskey. An $O(n^2)$ time algorithm for deciding whether a regular language is a code. *Journal of Computing and Information*, (10):79–89, 1996. updated version downloaded at http://www.cs.uofs.edu/∼mccloske/publications/code_alg_header_mar2011.pdf.

[23] P. Oprocha. Fast solutions for DNA code words test. Schedae Informaticae 15 (2006).

[24] G. Paun, G. Rozenberg, and A. Salomaa. *DNA Computing, new computing paradigms.* Springer, 1996.

[25] K. Sakamoto, H. Gouzu, K. Kkomiya, D. Kiga, S. Yokoyama, T. Yokomori, and M. Hagiya. Molecular computation by dna hairpin formation. *Science*, 288:1223–1226, 2000.

[26] A. Sardinas and C. Patterson. A necessary and sufficient condition for the unique decomposition of coded messages. *IRE Intern. Conv. Record*, (8):104–108, 1953.