SCHEDAE INFORMATICAE

VOLUME 15 2006

Fast Solutions for DNA Code Words Test

Piotr Oprocha
AGH University of Science and Technology, Faculty of Applied Mathematics
Al. Mickiewicza 39, 30-059 Kraków, Poland, and
Jagiellonian University, Institute of Mathematics
Reymonta 4, 30-059 Kraków, Poland
e-mail: oprocha@agh.edu.pl

Abstract. An essential part of any DNA computation is to encode data on DNA strands. Performing such computation we must be sure that no undesirable hybridization will occur. The aim of this paper is to present algorithms which can test if a given set of code words satisfies certain coding and involutory properties, as it is necessary to prevent undesirable DNA strands interactions. These algorithms are based on the theory of codes and pattern matching methods.

If we denote by m sum of the lengths of words in a given set, then all presented algorithms can be realized with $\mathcal{O}(m^2)$ complexity both for time and space.

Keywords: DNA, strings properties, code, complexity, algorithms.

1. Introduction

Every DNA molecule is an oriented sequence of nucleotides, so it can be represented as a single word over alphabet $\Delta = \{A, C, G, T\}$. It is important property of DNA that C is complementary to G, A is complementary to T and two complementary sequences (single DNA strands) with opposite orientations joint together (hybridize) forming a double stranded DNA molecule (this property is called Watson-Crick complementarity).

It is interesting that DNA may be used in computational problems. As DNA strands join together almost in a blink, it give the possibility to solve some complicated problems (e.g. the existence the Hamilton Cycle problem) very fast. There are many technical problems with such experiments, however there is hope for some improvements in future. Main ideas and results in DNA computing the reader may find in [6].

In laboratory experiments it is very important to avoid some unnecessary hybridizations (see Fig. 1). A situation, when a part of a given DNA sequence (longer than k) is complement to the other part of this sequence with the same length, is undesired. For a given set of DNA sequences, a situation, when one sequence is a reverse complement of the subword of another sequence or it is a reverse subword of concatenation of other two sequences, is also unliked.

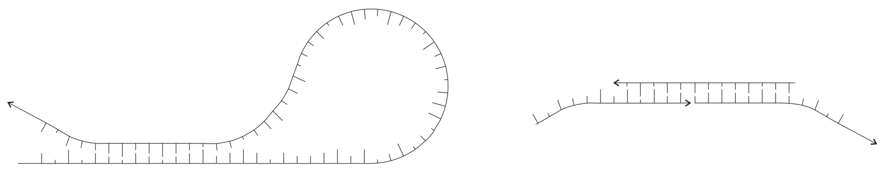

Fig. 1. Unallowed hybridizations

Papers [9, 6] define properties of words and languages, which if fulfilled by a given set of DNA strands, prevent situations described above. Authors of [7, 10] present methods and try to implement theoretical results in software program. The most important part of this program is to perform tests of generated sequences. The algorithm introduced there work in $\mathcal{O}(m^4)$ time, where m denotes the sum of the lengths of DNA sequences in a given set.

In this paper we describe how to construct algorithms with complexity $\mathcal{O}(m^2)$, which speed up whole process of DNA code words generation. In the presented algorithms the theory of codes is used as well as string matching algorithms like KMP or Aho-Corasick algorithm (see [3, 4]).

2. Basic definitions and properties

An alphabet Λ is any nonempty finite set. We will denote by Δ the alphabet $\{A, C, G, T\}$ representing the DNA nucleotides.

The word $w \in \Lambda^*$ is a prefix (resp. suffix) of the word $x \in \Lambda^*$, which we denote $w \sqsubset x$ (resp. $w \sqsupset x$), if there exists word $u \in \mathcal{A}^*$ such that $wu = x$ (resp. $uw = x$). The length of the word $w = a_1 \dots a_n$ we denote by $|w| = n$.

Definition 1. *Let Λ be an alphabet. A subset X of the free monoid Λ^* is a code over Λ if, for all $n, m \geq 1$ and for all $x_1, \ldots, x_n, x'_1, \ldots, x'_m \in X$, the condition*

$$x_1 x_2 \ldots x_n \quad = \quad x'_1 x'_2 \ldots x'_m$$

implies

$$n = m \quad \text{and} \quad x_i \quad = \quad x'_i \quad \text{for all} \quad i = 1, \ldots, n.$$

The *flower automaton* of X is by the definition the automaton $\mathcal{A}(X) = (Q, E, I)$ containing a set of states Q, the labelling (transition) $E \subset Q \times \Lambda \times Q$ and state I which is the initial and terminal state at once. If we denote $I = \{*\}$ then we may define the set of states as

$$Q = \{*\} \cup \{q_i^w \ : \ w = a_1 \ldots a_n \in X, \ i = 2, \ldots, n\}$$

and then the transitions in $\mathcal{A}(X)$ are

$$\begin{aligned} E \quad = \quad & \{(*, a_1, q_1^w) \mid w = a_1 \ldots a_n \in X\} \cup \{(q_n^w, a_n, *) \mid w = a_1 \ldots a_n \in X\} \cup \\ & \left\{(q_i^w, a_i, q_{i+1}^w) \mid w = a_1 \ldots a_n \in X, \ i = 2, \ldots, n-1\right\}. \end{aligned}$$

We also define labelling function $\lambda : E \to \Lambda$ such that $\lambda((q_1, a, q_2)) = a$. The labelling function can be extended to paths labelling in the usual way.

The automaton $\mathcal{A}(X)$ is *unambiguous* if for all $p, q \in Q$ and $w \in \Lambda^*$ there is at most one path form p to q with label w.

The *square* of $\mathcal{A}(X)$ is the automaton $\mathcal{A}'(X) = (Q', E', I')$ such that $Q' = Q \times Q$, $I' = \{*'\} = \{(*, *)\}$ and

$$E' = \left\{((p, q), a, (p', q')) \mid (p, a, p'), (q, a, q') \in E, \ a \in \Lambda\right\}.$$

Proposition 1. *(see [2, chapt. IV thm. 2.1]) A finite set of words X is a code if and only if the flower automaton $\mathcal{A}(X)$ is unambiguous.*

The following proposition follows from the definitions

Proposition 2. *The flower automaton $\mathcal{A}(X)$ is unambiguous if and only if there is no path in $\mathcal{A}'(X)$ from a state (p, p) to a state (q, q) visiting a state (r, s) with $r \neq s$.*

Corollary 1. *For any state $(q, q) \in Q'$ there is a path from $*'$ to (q, q) and a path from (q, q) to $*'$.*

COROLLARY 2. *A finite set of words X is a code if and only if there is no cycle in $\mathcal{A}'(X)$ from a state $*'$ visiting a state (r,s) with $r \neq s$.*

We recall that oriented graph G is a pair (V,E), where V is any finite set, and $E \subset V \times V$. For any $u \in V$ we will denote by $Adj(u)$ the set $Adj(u) = \{v \in V \,:\, (u,v) \in E\}$. If $G = (V,E)$ then its transposition is the graph $G^T = (V,E^T)$, where $E^T = \{(v,u) \,:\, (u,v) \in E\}$.

An involution $\theta : \Lambda \to \Lambda$ of a set Λ is a mapping such that $\theta^2 = id_\Lambda$. We will define two special involutions ν and ρ of the set Δ^*. The mapping ν is defined by $\nu(A) = T$, $\nu(T) = A$, $\nu(C) = G$ and $\nu(G) = C$, and may be extended to an involution of Δ^*. The mapping ρ is defined inductively as $\rho(s) = s$ and $\rho(us) = \rho(s)\rho(u) = s\rho(u)$ for any $s \in \Delta$ and $u \in \Delta^*$.

DEFINITION 2. *Let θ be an involution of a set Λ and let $X \subset \Lambda^*$ be a finite set.*

1. *The set X is called $\theta(k,m_1,m_2)$-subword compliant if for all $u \in \Lambda^*$ such that $|u| \geq k$ we have $\Lambda^* u \Lambda^m \theta(u) \Lambda^* \cap X = \emptyset$ for $m_1 \leq m \leq m_2$.*
2. *The set X is called θ-compliant if $\Lambda^*\theta(x)\Lambda^* \cap X = \emptyset$ for all $x \in X$.*
3. *The set X is called θ-free if $X^2 \cap \Lambda^*\theta(x)\Lambda^* = \emptyset$ for all $x \in X$.*
4. *The set X is called strictly θ if $\theta(x) \notin X$ for all $x \in X$.*

COROLLARY 3. *The set X is $\theta(k,m_1,m_2)$-subword compliant if and only if for all $u \in \mathcal{A}^k$ we have $\mathcal{A}^* u \mathcal{A}^m \theta(u) \mathcal{A}^* \cap X = \emptyset$ for $m_1 \leq m \leq m_2$.*

EXAMPLE 1. *Let us consider following sets of words*

$$X_1 = \{AA, TT, CTT\}\ ,\ X_2 = \{AA, TAC, GT\}\,.$$

Obviously both defined sets are prefix codes. Let $\widetilde{X_i} = \theta(X_i)$, where θ denotes morphism $\theta = \nu\rho$. In this case

$$\widetilde{X_1} = \{AA, TT, AAG\}\ ,\ \widetilde{X_2} = \{TT, GTA, AC\}\,.$$

Observe that X_1 is not strictly θ as $\theta(AA) = TT \in X_1$. If we modify X_1 to $X_1' = X_1 \setminus \{AA\}$ then it becomes strictly θ but not θ-compliant as $G\theta(TT) = GAA \in X_1'$. Set X_2 is strictly θ and θ-compliant but it is not θ-free. We have that $(X_2)^2 = \{AAAA, TACAA, GTAA, \ldots, GTGT\}$ and $GTAA \in \theta(TAC)\mathcal{A} \subset \mathcal{A}^*\theta(TAC)\mathcal{A}^*$. Set X_2 is θ-compliant so it is also $\theta(k,m_1,m_2)$-subword compliant for any parameters k, m_1, m_2.

It is an important property of a single-stranded DNA molecule that its two ends are physically different. By convention, one end is called $5'$ end and the other one is called the $3'$ end. Let $u \in \Delta^*$ denote DNA strand in its $5' \to 3'$ orientation. If we denote by $\overleftarrow{\overline{u}}$ the Watson-Crick complement of the word u (i.e. oppositely-oriented complementary DNA strand which can bind to u forming double-stranded DNA), also in orientation $5' \to 3'$, then it is easily seen that $\nu(\rho(u)) = \rho(\nu(u)) = \overleftarrow{\overline{u}}$ (word $\overleftarrow{\overline{u}}$ consists of symbols (nucleotides) complement to symbols of u and its order is inverted).

We would like to test if a given set of words over alphabet Δ (DNA strands) is $\nu\rho$-free and $\nu\rho$-subword compliant (fulfills conditions (1)–(4) for involution $\theta = \nu\rho$). These properties describe intermolecular hybridizations which are unlikely for us. We will try to construct fast algorithms checking all these properties.

3. Algorithms

We present algorithms checking if a given set is a code, θ-compliant and θ-free. We will also show that the upper bound for the complexity function of presented algorithms is $\mathcal{O}(n^2)$. All algorithms will be presented in the simplified pseudocode.

3.1. A test for a code

By Corollary 2 it is enough to check if there exists path from $*'$ to $*'$ going through state (r, s), where $r \neq s$. It is also important that paths through states (p, p) need not to be checked as the next state on the path is $*'$ or (q, q) for some q.

The number of states n and the number of transitions m in the flower automaton $\mathcal{A}(X)$ for a code $X = \{x_1, x_2, \ldots, x_{|X|}\}$ are equal to

$$n = 1 + \sum_{i=1}^{|X|} (|x_i| - 1) \quad \text{and} \quad m = \sum_{i=1}^{|X|} |x_i|.$$

We will assume that $|x| > 1$ for each $x \in X$. With this assumption $m < 2n$ and automaton is "thin" so we will represent the flower automaton by transition lists. In case of DNA we have only 4 symbols, so we will use one list for each symbol $a \in \Lambda$.

First, we present an algorithm which for a given set of words X builds a flower automaton $A(X)$.

```
Flower-Automaton(X)

1: $q \leftarrow 0$
2: for each $x \in X$
3:     do Add-Trans($Q[0]$,$x_1$,$Q[q+1]$)
4:         Add-Trans($Q[q+|x|-1]$,$x_{|x|}$,$Q[0]$)
5:         for $i \leftarrow 2$ to $|x|-1$
6:             do $q \leftarrow q+1$
7:                 Add-Trans($Q[q]$,$x_i$,$Q[q+1]$)
8: return $Q$
```

States of the $A(X)$ are stored in the table Q and state $*$ is represented by $Q[0]$. Function Add-Trans(q,a,q') add transition (q,a,q') to the list of transitions of the state q. This function may be realized in $\mathcal{O}(1)$, so the flower automaton is build in linear time $\mathcal{O}(n+m)=\mathcal{O}(m)$.

The next step is to build an $\mathcal{A}'(X)$ automaton. As we are only interested in the existence of a special cycle visiting $*'$ it is enough for us to construct an oriented graph G with the vertex set equal to Q'. There is an edge between two vertices q and p if there exists $a \in \Lambda$ such that transition $(q,a,p) \in E'$. In fact there exists at most one such a. Vertices of graph G with its adjacency lists are stored in table $V[i][j]$, where $i,j = 1,\ldots,n$.

```
Square-Graph(Q)

1: $n \leftarrow |Q|$
2: for $i \leftarrow 0$ to $n-1$
3:     do for $j \leftarrow 0$ to $n-1$
4:             do Add-Edges$(i,j)$
5: return $V$
```

Function ADD-EDGES(i, j) works as follows. For each $a \in \Lambda$ it takes transition list $L(a, i)$ of the state $Q[i]$ and transition list $L(a, j)$ of the state $Q[j]$. If transition $(Q[i], a, Q[r]) \in L(a, i)$ and $(Q[j], a, Q[s]) \in L(a, j)$ then it adds an edge from $V[i][j]$ to $V[r][s]$ to the adjacency list of vertex $V[i][j]$. Observe that every pair of transitions is considered only once. It implies that complexity of the procedure SQUARE-GRAPH(Q) is equal to $\mathcal{O}(n^2 + m^2) = \mathcal{O}(m^2)$.

The last part of the algorithm is construction of the procedure which determines if there exists a cycle starting in vertex $V[0][0]$ and visiting vertex $V[i][j]$ for some $i \neq j$. We will use some modification of the algorithm STRONGLY-CONNECTED-COMPONENTS presented in [3] to check this property. We will also use procedure INITIALISE-COLOURS which assigns colour *white* to vertices $V[i][i]$ and *gray* to vertices $V[i][j]$, where $i \neq j$. It obviously may be realized in $\mathcal{O}(m^2)$ operations. Colours of vertices are stored in table *colour*. We will also use temporary table *visited*.

```
DFS-VISIT$(u, V)$

 1: $visited[u] \leftarrow$ TRUE
 2: for each $v \in Adj(u)$
 3:     do if $visited[v] =$ FALSE
 4:         then if $colour[v] \neq$ BLACK
 5:             then if $colour[v] =$ GRAY
 6:                 then $colour[v] \leftarrow$ BLACK
 7:                 if DFS-VISIT$(v, V)$ $=$ FALSE
 8:                 then return FALSE
 9:             else return FALSE
10: return TRUE
```

```
CODE-TEST(X)

 1: $Q \leftarrow$ FLOWER-AUTOMATON(X)
 2: $V \leftarrow$ SQUARE-GRAPH(Q)
 3: INITIALISE-COLOURS(V)
 4: $u \leftarrow V[0][0]$
 5: for each $v \in V$
 6:     do $visited[v] \leftarrow$ FALSE
 7: DFS-VISIT$(u, V)$
 8: change $G$ to $G^T$
 9: for each $v \in V$
10:     do $visited[v] \leftarrow$ FALSE
11: return DFS-VISIT$(u, V)$
```

Function DFS-VISIT(u) is a simple modification of standard DFS. When it reaches gray vertex its colour is changed to black, which means that there exists path from $V[0][0]$ to this vertex. In the first use of DFS it is impossible to reach a black unvisited vertex. White vertices are not considered as there could happen anything "bad" on the path form such vertex to $*'$ or from $*'$. If there exists a path in G^T from $V[0][0]$ to an unvisited vertex coloured black, DFS-VISIT will return FALSE and then given set X is not a code.

For graph G defined by adjacency lists transposition graph G^T may be calculated with complexity $\mathcal{O}(|V[G]| + |E[G]|)$ (see [3, chapt. 23]), which in our case equals to $\mathcal{O}(n^2 + m^2) = \mathcal{O}(m^2)$. It means that function CODE-TEST(X) has complexity bounded by $\mathcal{O}(m^2)$. Memory complexity is also bounded by $\mathcal{O}(m^2)$. Another approach to this problem (with the same complexity) is presented in [4]. Both approaches are faster than the algorithm introduced in [7] which has time complexity $\mathcal{O}(n^4) = \mathcal{O}(m^4)$.

3.2. A test for θ-compliance and a strictly θ test

At the beginning we must recall KMP algorithm as it is necessary for us to match patterns in linear time.

Suppose we are given text $T = [1, \dots, t]$ and pattern $P = [1, \dots, p]$. We will use notation $P_q = P[1..q]$. The basic part of this algorithm is to compute the prefix function π defined as

$$\pi[q] = \max\{k \ : \ k < q \ , \ P_k \sqsupset P_q\} .$$

We will use COMPUTE-PREFIX-FUNCTION to compute $\pi[i]$ for $i = 1, \dots, p$. It may be done with complexity $\mathcal{O}(p)$ (see [3, chapt. 34, p. 975]). This function is the main part of KMP-MATCHER which decides if P is a substring of T. This function needs $\mathcal{O}(p + t)$ time and $\mathcal{O}(p)$ space [3].

We will use the naive algorithm to check if a given set X is strictly θ, as it seems that there is no faster method.

```
STRICTLY-$\theta$-TEST(X)

1: for each $x \in X$
2:     do $w \leftarrow \theta(x)$
3:         for each $u \in X$
4:         do if $|u| = |w|$ AND KMP-MATCHER$(w, u)$
5:             then return FALSE
6: return TRUE
```

It is easy to see that complexity of this algorithm is bounded by

$$\mathcal{O}(\sum_{i=1}^{|X|}(|x_i| + m)) = \mathcal{O}(|X|\ m) \leq \mathcal{O}(m^2).$$

To check θ-compliance we need only to remove condition $|w| = |u|$ from the line 4 of the STRICTLY-θ-TEST function.

Next we will construct an algorithm testing if X is $\theta(k, m_1, m_2)$-subword compliant. Main idea of this algorithm is given by Corollary 3. We will use modified KMP-MATCHER which sets $M[i]$ with TRUE if $T[i \dots i + |P| - 1] = P$ and FALSE otherwise, for all $i = 1, \dots, |T|$. We will call such table M match table.

```
KMP-COMPUTE-MATCH-TABLE$(T, P, \pi, M)$

1: for $i \leftarrow 1$ to $|T|$
2:     do $M[i]$ = FALSE
3: $q \leftarrow 0$
4: for $i \leftarrow 1$ to $|T|$
5:     do while $q > 0$ and $P[q+1] \neq T[i]$
6:             do $q \leftarrow \pi[q]$
7:         if $P[k+1] = T[i]$
8:             then $q \leftarrow q + 1$
```

```
 9:         if $q = |P|$
10:             then $M[i] = $ TRUE
11:                 $q \leftarrow \pi[q]$
```

SUBWORD-COMPLIANCE-TEST(k, m_1, m_2,X)

```
 1: for each $x \in X$
 2:     do for $i \leftarrow 1$ to $|x| - k$
 3:             do $u \leftarrow x[i \ldots i + k - 1]$
 4:                 $w \leftarrow \theta(u)$
 5:                 $\pi_u \leftarrow$ COMPUTE-PREFIX-FUNCTION($u$)
 6:                 $\pi_w \leftarrow$ COMPUTE-PREFIX-FUNCTION($w$)
 7:                 for each $y \in X$
 8:                     do KMP-COMPUTE-MATCH-TABLE($y, u, \pi_u, M_u$)
 9:                         KMP-COMPUTE-MATCH-TABLE($y, w, \pi_w, M_w$)
10:                         $i, j \leftarrow 1$
11:                         while $i \leq |y|$ and $j \leq |y|$
12:                             do if $m_2 < j - i - k$ or $M_u[i] = $ FALSE
13:                                     then $i \leftarrow i + 1$
14:                                 if $m_1 > j - i - k$ or $M_w[j] = $ FALSE
15:                                     then $j \leftarrow j + 1$
16:                                 if $m_1 \leq j - i - k \leq m_2$ and
17:                                         $M_u[i] = $ TRUE and $M_w[j] = $ TRUE
18:                                     then return FALSE
19: return TRUE
```

Function Subword-Compliance-Test returns TRUE if a given set is $\theta(k, m_1, m_2)$-subword compliant and FALSE otherwise. Observe that lines 8 to 18 of the procedure are realized in linear time $\mathcal{O}(|y|)$. We may assume that time complexity of these lines is bounded by Cy and constant C is independent of y. We may also assume that $k < m$ otherwise procedure stops at line 2. Lines 3 to 6 need $4k$ operations. If we take $C' = C + 4$ then complexity of the whole function may be calculated as follows

$$\sum_{i=1}^{|X|}\sum_{j=1}^{|x_i|}(4k + \sum_{y\in X} Cy) \leq \sum_{i=1}^{|X|}\sum_{j=1}^{|x_i|}(4 + C)m = \sum_{i=1}^{|X|} C'|x_i|m = C'm^2 = \mathcal{O}(m^2).$$

3.3. A test for θ freedom

The problem of θ freedom detection may be considered as the exact matching problem with a set of patterns. Let us denote by P the set of all involutions of elements of the set X, i.e. $\mathcal{P} = \{\theta(x) : x \in X\} = \{p_1, \ldots, p_k\}$, and let $\mathcal{T}$ denote the set $\mathcal{T} = X^2 = \{uv : u, v \in X\} = \{t_1, \ldots, t_l\}$, where $k = |X|$ and $l = k^2$. Observe that X is θ-free iff p_i is not a subword of t_j for any $i = 1, \ldots, k$ and $j = 1, \ldots, l$.

To solve the problem, we will use Aho-Corasick algorithm. Main idea of this algorithm is to build a keyword tree of the set $\mathcal{P}$ (see Fig. 2).

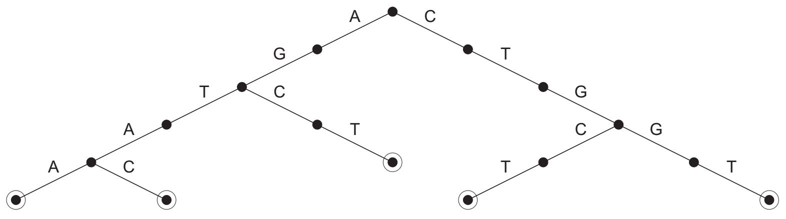

Fig. 2. Keyword tree for $\mathcal{P} = \{$AGTAA,CTGCT,AGCT,AGTAC,CTGGT$\}$

Let $\mathcal{L}(v)$ denote labelling of the path from root of $\mathcal{K}$ to node v. Define $lp(v)$ as the length of the longest proper suffix of word $\mathcal{L}(v)$ that is prefix of some pattern in $\mathcal{P}$. Then for node v let n_v denote the unique node in $\mathcal{K}$ labelled with the suffix of $\mathcal{L}(v)$ with length $lp(v)$. When $lp(v) = 0$ then n_v is the root of $\mathcal{K}$. We call ordered pair (v, n_v) a *failure link* (see Fig. 3).

In fact a keyword tree with failure links may be constructed in linear time (see [5, chapt. 3.4.]), i.e. the upper bound of time complexity is equal to $\mathcal{O}(\sum p_i) = \mathcal{O}(m)$.

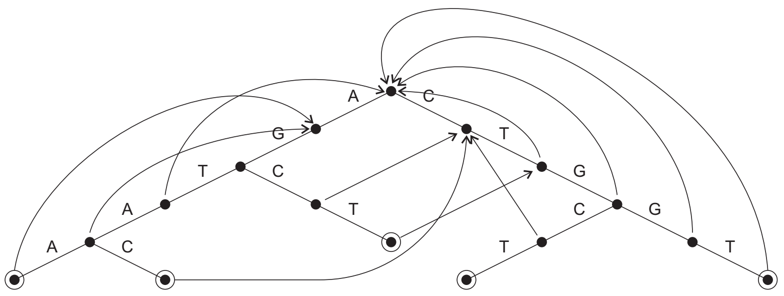

Fig. 3. Keyword tree showing the failure links

Given a keyword tree with failure links and word t_j we need only $\mathcal{O}(|t_j|)$ time to verify if there exists i such that p_i is a subword of t_j (see [5]). We are searching for pattern matching in the set of texts $\mathcal{T}$, so in our case time complexity of the searching process is bounded by

$$\mathcal{O}(\sum_{i=1}^{l} |t_i|) = \mathcal{O}(\sum_{i,j=1}^{|X|} |x_i||x_j|) = \mathcal{O}(m^2).$$

Then θ freedom test complexity has $\mathcal{O}(m^2)$ as the upper bound ($\mathcal{O}(m)$ for tree construction and $\mathcal{O}(m^2)$ for words matching).

4. References

[1] Aho A.V., Hopcroft J.E., Ullman J.D.; *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[2] Bertsel J., Perrin D.; *Theory of Codes*, Academic Press, Inc. Orlando Florida, 1985.

[3] Cormen T.H., Leiserson C.E., Rivest R.L.; *Introduction to Algorithms*, Cambridge 1991.

[4] Crochemore M., Rytter W.; *Text Algorithms*, Oxford University Press, 1994.

[5] Gusfield D.; *Algorithms on Strings, Trees and Sequences*, Cambridge 1997.

[6] Jonoska N., Mahalingham K.; *Languages of DNA based code words*, in: J. Chien, J. Reif, (eds.), *Preliminary Proceedings of the 9th International Meeting on DNA Based Computers*, 2003, pp. 58–68.

[7] Jonoska N., Kephart D.E., Mahalingam K.; *Generating DNA Code Words*, Congressus Numerantium, 156, 2002, pp. 99–110.

[8] Jonoska N.; *Trends in Computing with DNA*, J. Comput. Sci. Technol., 19, 2004.

[9] Kari L., Konstantinidis S., Losseva E., Wozniak G.; *Sticky free and overhang free DNA languages*, Acta Informatica, 40(2), 2003, pp. 119–157.

[10] Kephart D.E., LeFevre J.; *CODEGEN: The Generation and Testing of DNA Code Words*, to appear in IEEE.

Received September 30, 2004