# Research Report

## On the Time Complexity of the TEIRESIAS Algorithm

**Aris Floratos, Isidore Rigoutsos**
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich**

# On The Time Complexity Of The TEIRESIAS Algorithm

*Aris Floratos*[1,2,3] *and Isidore Rigoutsos*[1,3]

**Abstract**

In this paper we discuss *TEIRESIAS*, a novel algorithm for the discovery of rigid patterns in sets of biological sequences. *TEIRESIAS* is a combinatorial algorithm able to detect *all* patterns satisfying a user defined minimum support requirement. In previous work ([1, 2]) we presented a high level description of the algorithm, focusing mainly on its applications on specific input sets and on qualitive evaluations of the obtained results. Here we provide a detailed look at questions of implementation and complexity.

## 1 Introduction

One of the problems arising in the analysis of biological sequences is the discovery of sequence similarity in the primary structure of related proteins or genes. Such similarity usually corresponds to residues conserved during evolution due to an important structural or functional role.

Several methods have been proposed for dealing with this problem. One widely used class of algorithms [6, 3, 5, 12, 13, 14, 16, 28] employs *global string alignment* [11]; in this context *edit* operations along with their associated costs are used for transforming one sequence to another. What is sought is a minimum cost *consensus* sequence that highlights the regions of similarity among the input sequences.

Alignment algorithms suffer from several inherent drawbacks. First, the task of optimally aligning a set of strings is computationally very expensive (it is known to be an NP-hard problem [15]). Second, alignment can reveal only global similarities [19, 23]; if the sequences under comparison are distantly related or if the relative order of their similar regions varies among sequences (*domain swapping*) it is quite possible than no substantial alignment can be produced.

One way to overcome the difficulty that alignment algorithms have in identifying local similarities is to focus on the discovery of *patterns* shared by the input sequences . A pattern is a formal way to define the notion of local similarity. As an example consider the alphabet of the 20 amino acids; in this context *"A.CH"* is a valid pattern, describing all oligopeptides that start with an Alanine, have an arbitrary amino acid in the second position and end with a Cysteine followed by a Histidine. A protein *matches "A.CH"* if it contains at least one peptide stretch that is described by this pattern. The assumption behind pattern discovery approaches is that a pattern that appears often enough in a set of biological sequences is expected to play a role in defining the respective sequences' functional behavior, and/or evolutionary relationships.

A number of pattern discovery algorithms have been steadily appearing in the literature the past few years [29, 23, 22, 21, 20, 19, 30]. Most of the proposed approaches proceed by *enumerating* the solution space; i.e. they generate all (or most of) the possible patterns and then verify, for each one, that it has sufficient *support* (i.e. it appears in sufficiently many input sequences - the exact number is usually provided by the user). Unfortunately, unless the nature of the patterns sought is extremely simple, the problem of detecting all existing patterns is NP-hard. Possible remedies include the use of heuristics [21, 22, 23, 20] which offer enhanced performance (but frequently at the expense of sacrificing the completeness of the results), and/or the structural restriction of the patterns sought (e.g. most algorithms restrict the maximum length that a pattern can have).

*TEIRESIAS* belongs to this last, pattern discovery genre of algorithms. It is capable of detecting and reporting *all* existing patterns in a set of input sequences *without* enumerating the entire solution space and *without* using alignment. Furthermore, the patterns reported are guaranteed to be *maximal*, i.e. they are as specific as possible. For example, if *"A.CE"* is a pattern appearing in a given number of positions within the input then it makes no sense reporting the pattern *"A..E"* if this second pattern appears at the exact same positions and nowhere else.

---

[1] Bioinformatics and Pattern Discovery, Computational Biology Center, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA.

[2] Courant Institute Of Mathematical Sciences, New York University, New York, NY 10012, USA.

[3] Email: aris@us.ibm.com, rigoutso@us.ibm.com

# 2 Algorithm Description

*TEIRESIAS* operates in two phases: *scanning* and *convolution*. the scanning phase identifies *elementary patterns* with sufficient support. Then, during the convolution phase, these elementary patterns are combined into progressively larger and larger patterns until all the existing, maximal patterns have been generated. Furthermore, the order in which the convolutions are performed makes it easy to identify and discard non-maximal patterns.

## 2.1 Terminology and Problem Statement

Let $\Sigma$ be the alphabet of residues at hand (e.g. the set of all amino acids). A pattern is defined as a regular expression of the form

$$\Sigma(\Sigma \cup \{'.'\})^* \Sigma$$

where '.' (referred to as the *"don't care character"*) is used to denote a position that can be occupied by an arbitrary residue. Being a regular expression, every pattern $P$ defines a language $\mathbf{G}(P)$ consisting of all the strings that can be obtained from $P$ by substituting each don't care by an arbitrary residue from $\Sigma$ For the pattern *"A.CH..E"* for example, the following peptides are elements of $\mathbf{G}$*("A.CH..E")*:

ADCHFFE, ALCHESE, AGCHADE

For any pattern $P$, any substring of $P$ that is itself a pattern is called a *subpattern* of $P$. For example, *"H..E"* is a subpattern of the pattern *"A.CH..E"*. A pattern $P$ is called a $<L, W>$ pattern (with $L \leq W$) if every subpattern of $P$ with length $W$ or more contains at least $L$ residues.

A string of residues over $\Sigma$ is said to *match* a given pattern $P$ if it contains at least one substring that belongs in $\mathbf{G}(P)$.

Given a pattern $P$ and a set of sequences $S = \{s_1, s_2, \ldots, s_n\}$ we define the *offset list of P with respect to S* (or simply the *offset list of P*, when $S$ is unambiguously implied) to be the following set

$$L_S(P) = \{(i, j) \mid \text{sequence } s_i \text{ matches } P \text{ at offset } j\}.$$

A pattern $P'$ is said to be *more specific* than a pattern $P$ if $P'$ can be obtained from $P$ by changing one or more don't care characters to residues or by appending an arbitrary string of residues and don't cares to the left or/and right of $P$. The following patterns are all more specific than the pattern *"A.CH..E"*

*"AFCH..E", "A.CHL.E.K", "SA.CH..E"*

Notice that if a pattern $P'$ is more specific than a pattern $P$ then for every set $S$ of input sequences $|L_S(P')| \leq |L_S(P)|$.

Given a set of sequences $S$, a pattern $P$ is called *maximal* with respect to $S$ if there exists no pattern $P'$ which is more specific than $P$ and such that $|L_S(P)| = |L_S(P')|$. Conversely, if $P$ is not maximal then there exists a maximal pattern $P'$ such that $|L_S(P)| = |L_S(P')|$. We then say that $P'$ subsumes $P$.

Using the above definitions we can succinctly describe the problem addressed by *TEIRESIAS* as follows:

**Problem Definition:**

*"given a set $S = \{s_1, s_2, \ldots, s_n\}$ of input sequences and parameters L, W, K, find all maximal $<L, W>$ patterns that have support at least K (i.e. they appear in at least K distinct sequences in S)."*

In what follows, the letters $L$, $W$ and $K$ are used to denote the parameters specified in the Problem Definition given above and are assumed to have values which have been provided by the user. Furthermore, when the term "pattern" is used without further qualification it always implies an $<L,W>$ pattern.

## 2.2 Implementation

*TEIRESIAS* begins by scanning the sequences in the input set $S$ and locating all *elementary patterns* with support at least $K$. An elementary pattern is just a $<L, W>$ pattern containing exactly $L$ residues. For example, if

$$S = \{s1 = SDFBASTS, s2 = LFCASTS, s3 = FDASTSNP\}$$

then the set of all <3, 4> elementary patterns with support at least 3 is

$$\{ \text{"F.AS", "AST", "AS.S", "STS", "A.TS"} \}$$

The set of elementary patterns thus collected becomes the input to the convolution phase. The key observation behind the convolution phase is that an original pattern $P$ can be reconstructed by piecing together pairs $A$, $B$ of

intermediate patterns such that a suffix of $A$ is the same as a prefix of $B$. Consider again the set of sequences $S$ given above. This set contains the pattern *"F.ASTS"*. It is possible to reconstruct this pattern in the following way:

- Combine together the elementary *"F.AS"* and *"AST"* into the pattern *"F.AST"* (observe that *"AS"* is both a suffix of the pattern *"F.AS"* and a prefix of *"AST"*).
- Combine the newly generated pattern *"F.AST"* with the elementary *"STS"* to get *"F.ASTS"* (again, *"ST"* is a suffix of *"F.AST"* and a prefix of *"STS"*).

To make the above description more precise, we need the following definitions: given any pattern $P$ with at least $L$ residues, let *prefix(P)* be the (uniquely defined) subpattern of $P$ that has exactly *(L-1)* residues and is a prefix of $P$. Similarly, let *suffix(P)* denote the suffix subpattern of $P$ with exactly *(L-1)* residues. For example, if $L = 3$, then

*prefix("F.ASTS") = "F.A", prefix("AST") = "AS",*
*suffix("F.A...S") = "A..S", suffix("ASTS") = "TS".*

We can now describe *convolution*, a new binary operation (denoted by $\oplus$), between pairs of patterns: let $P$, $Q$ be arbitrary patterns with at least $L$ residues each; the *convolution* of $P$ with $Q$ is a new pattern $R$ defined as follows:

$$R = P \oplus Q = \begin{cases} PQ' & \text{if } suffix(P) = prefix(Q) \\ \varnothing & \text{otherwise} \end{cases}$$

where $Q'$ is a string such that $Q = prefix(Q)Q'$ (i.e. $Q'$ is what remains of $Q$ after the *prefix(Q)* is thrown away) and $\varnothing$ denotes the empty string. The patterns $P$, $Q$ are called *convolvable* if $P \oplus Q \neq \varnothing$. Here are some examples for the case $L = 3$:

*"DF.A.T"* $\oplus$ *"A.TSE" = "DF.A.TSE", "AS.TF"* $\oplus$ *"T.FDE"* $= \varnothing$.

If two patterns $P$, $Q$ are convolvable and $R = P \oplus Q$ then the offset list $L_S(R)$ of the resulting pattern $R$ is the subset of $L_S(P)$ defined as

$$L_S(R) = \{(i, j) \in L_S(P) \mid \exists (i, k) \in L_S(Q) \text{ such that } k - j = |P| - |suffix(P)|\},$$

where $|P|$ denotes the number of characters (counting both residues and don't cares) in the pattern $P$.

Let $\sigma_1, \sigma_2 \in \Sigma$ and $x, y \in (\Sigma \cup \{'.'\})^*$. Then

- $\sigma_1 x <_{pf} \varnothing$ , $\varnothing <_{pf} .x$ , $\sigma_1 x <_{pf} .y$
- $\sigma_1 x <_{pf} \sigma_2 y$ iff $x <_{pf} y$
- $.x <_{pf} .y$ iff $x <_{pf} y$
- $x\sigma_1 <_{sf} \varnothing$ , $\varnothing <_{sf} x.$ , $x\sigma_1 <_{sf} y.$
- $x\sigma_1 <_{sf} y\sigma_2$ iff $x <_{sf} y$
- $x. <_{sf} y.$ iff $x <_{sf} y$

**Figure 1** : The definition of the two partial orderings on the elements of $(\Sigma \cup \{'.'\})^*$. For any two strings $x$, $y$ we say that *"x is prefix-wise less than y"* when $x <_{pf} y$ and that *"x is suffix-wise less than y"* when $x <_{sf} y$.If neither $x <_{pf} y$ nor $x <_{pf} x$, then $x$, $y$ are said to be prefix-wise equal and we then write $x =_{pf} y$ (suffix-wise equality is defined similarly).

Convolution is the main operation used by our algorithm for the reconstruction of the original patterns. The order in which the convolutions are performed is dictated by two partial orderings on the universe of the patterns. The use of these orderings guarantees that (a) all patterns are generated and (b) a maximal pattern $P$ is generated before any non-maximal pattern subsumed by $P$. This way a non-maximal pattern can be detected with a minimal effort, just by comparing it against all patterns reported up to that point (comparisons can be made very efficiently using the appropriate hashing scheme to keep track of the maximal patterns). These two orderings (referred to as *"prefix-wise less"* and *"suffix-wise less"*) are described in Figure 1.

```
S = {s1, s2, ..., sm}: set of input sequences
si[j]  : the j-th character in the i-th sequence
EP     : set of elementary patterns
DirP(w): subset of EP containing all the elementary patterns starting with w
DirS(w): subset of EP containing all the elementary patterns ending in w
Maximal: set of maximal patterns
IsMaximal(R): returns 0 if R is subsumed by a pattern in Maximal, and 1 otherwise.
counts[i]: offset list corresponding to the i-th character in Σ(Σ is arbitrarily ordered)
```

**Scanning Phase**

```
EP = null
for all σ in Σ
  if support(σ) >= K
      Extend(σ)
  end-if
end-for



Extend(pattern P)
offset list counts[|Σ|]

A = # of regular characters in P
if A = L
  Add P to EP                  (1)
  return
end-if

for i = 0 to (W-|P|-L+A)       (2)
  for all σ in Σ               (3)
      counts[σ] = empty
  P' = P concatenated with i dots
  for (x, y) in LS(P)          (4)
      if (y+|P|+i) < |sx|
          σ= sx[y+|P|+i]
          Add (x, y+|P|+i) to
                counts[σ]
      end-if
  end-for
  for all σ in Σ
      if support(P'σ) >= K
          Extend(P'σ)
      end-if
  end-for
end-for
```

**Convolution Phase**

```
Order EP according to prefix-wise less and let
   all entries in EP be unmarked.
Create the directory strutures DirS and DirP
Maximal = empty
Clear stack
while EP not empty
  P = prefix-wise smallest unmarked element in EP
      (ties are resolved arbitrarily)
  mark P
  push(P)
  while stack not empty
start:
      T = top of stack
      w = prefix(T)
      U = {Q in DirS(w) | Q, T have not
                  been convolved yet}
      while U not empty
          Q = minimum element of U
              (according to suffix-wise less)   (5)
          R = Q ⊕ T                             (6)
          if |Ls(R)| = |Ls(T)|
              pop stack
          end-if
          if support(R) >= K && IsMaximal(R)
              push(R)
              goto start
          end-if
      end-while

      w = suffix(T)
      U = {Q in DirP(w) | Q, T have not
                  been convolved yet}
      while U not empty
          Q = minimum element of U
              (according to prefix-wise less)   (7)
          R = T ⊕ Q                             (8)
          if |Ls(R)| = |Ls(T)|
              pop stack
          end-if
          if support(R) >= K && IsMaximal(R)
              push(R)
              goto start
          end-if
      end-while

      T = pop stack
      Add T in Maximal                          (9)
      Report T
  end-while
end-while
```

**Figure 2** : Pseudo-code for both the scanning and the convolution phases.

Figure 2 gives pseudo code describing the convolution phase. The patterns are built inside a stack. When the stack gets empty a new elementary pattern is placed on its top to initiate the next round of convolutions. At each point in time, the top of the stack is occupied by a pattern which has been obtained from this elementary pattern through successive convolution operations.

The algorithm always works with the pattern $T$ which is found at the top of stack - we call this pattern the *current top*. First, $T$ is extended to the "left" (prefix-wise) by convoluting it with all the elementary patterns $Q$ that are convolvable with $P$, i.e. $Q \oplus P \neq \emptyset$. The convolutions are performed in the order prescribed by $<_{pf}$, the ties being arbitrarily broken. If the pattern $R$ resulting from such a convolution has support less than $K$ (this can be easily checked by examining the offset list of $R$) or it is non-maximal then $R$ is discarded, the current top remains unchanged and the next convolution is tried out. Otherwise, $R$ is placed at the top of stack, thus becoming the new current top and the procedure starts over again, this time with the new current top. After the pattern $T$ at the top of the stack can no longer be extended to the prefix direction, the same process is applied trying now to extend $T$ to the right (suffix-wise).

When extension in both directions has been completed, the current top is popped from the stack and is placed in the output set.

# 3 Correctness

In order to prove the correctness of the proposed algorithm, the following two things must be shown:

- all the maximal patterns are reported;
- no pattern that is non-maximal is ever reported.

The last of the above two points is also closely related to the performance of the algorithm. Our means of checking the maximality of a patterns is through the function *IsMaximal*() of Figure **2**. Since this function is called after every successful convolution (i.e. one that generates a pattern R with the prescribed minimum support $K$) and it requires the lookup of the structure *Maximal* containing the patterns reported up to that point, it is very important to be make sure that (i) the number of successful convolutions is not unnecessarily inflated and (ii) the size of *Maximal* is as small as possible. In order to achieve this goal the algorithm is so designed as to guarantee that all maximal patterns are reported before any non-maximal pattern. Based on this property we can prove that the structure *Maximal* contains at any time only maximal patterns and that a non-maximal pattern is recognized and discarded immediately.

For the discussion that follows, some extra terminology is needed. Any <L, W> pattern that is the result of a convolution (lines (6), (8) in the pseudo-code of Figure **2**) is said to be *generated* during the convolution phase. The subset of the generated patterns that make it to the line (9) of Figure **2** are called the *reported* patterns. All the patterns generated (reported) between the time an elementary pattern $P$ is placed on the stack and before the next elementary pattern is placed on the stack, are said to generated (reported) *during the expansion* of $P$.

For every $<L, W>$ pattern $P$ define the *core set* of $P$ as the ordered set of elementary patterns $CS(P) = (e_1, e_2, ..., e_k)$ such that $P = e_1 \oplus e_2 \oplus ..... \oplus e_k$. Let *seed(P)* be that among all elements of *CS(P)* which is first placed on the stack during the convolution phase. Lemma 1 below is a direct consequence of the definition of maximality.

**Lemma 1:** Let $P'$ be a non-maximal pattern and $P$ the maximal pattern subsuming $P$. Then

$$seed(P) <_{pf} seed(P') \quad \text{or} \quad seed(P) = seed(P')$$

**Lemma 2:** Let $P$ be an arbitrary pattern. If $P$ is generated at all then the first time that this happens is during the expansion of *seed(P)*. Furthermore, if $P$ is maximal then $P$ is always generated.

**Proof:** The proof is an easy induction on the size of the core set *CS(P)*. Details are deferred to the final version of the paper

**Theorem 1:** Let $P$ be a maximal pattern. Then when $P$ is generated for the first time, no non-maximal pattern $P'$ subsumed by $P$ has been reported.

**Proof:** (Sketch -The details will be given in the final version of the paper.) From Lemma 2 it follows that, since it is the first time that $P$ is generated, the elementary pattern currently under expansion is *seed(P)*. Let now $P'$ be an arbitrary non-maximal pattern subsumed by $P$. If it is the case that $seed(P) <_{pf} seed(P')$ then *seed(P')* has not yet been expanded (the order of expansion for the elementary patterns obeys the prefix-wise minimum ordering). As a result, $P'$ cannot have been reported since, according to Lemma 2, it has not even been generated. If, on the other hand, $seed(P) = seed(P')$ then, again by Lemma 2, P' is generated (if at all) during the expansion of seed(P). Analyzing the pseudo-code in

Figure 2 we can show that if even if P' is indeed generated it has yet not be reported at the time of the generation of P.

Given Theorem 1 above it is trivial to show that all the maximal patterns are indeed generated and reported. Furthermore, we have shown that every maximal pattern is reported before any non-maximal pattern that it subsumes. Given this, we can substantiate our claim that the structure *Maximal* always contains only maximal patterns. A detailed discussion will appear in the final version of the paper along with experimental data regarding the running time of the algorithm.

# 4 Time Complexity

In the implementation of *TEIRESIAS* the main object of our manipulations is the data structure representing a pattern. This data structure is a pair $(P, L_S(P))$ where $P$ is the string representation of the pattern and $L_S(P)$ is the offset list of the pattern. The list $L_S(P)$ is assumed to be ordered. This means that the offset $(x, y)$ appears before the offset $(x', y')$ if (i) $x < x'$ or (ii) $x = x'$ and $y < y'$. Several other data structures are used, mainly as repositories for patterns. These will be explained as they are encountered.

The remaining part of this section is divided in two parts, one addressing the scanning phase and the other the convolution phase of the algorithm. In what follows we use the character $m$ to denote the size of the input set $S = \{s_1, s_2, \ldots, s_n\}$, i.e. $m = \Sigma |s_i|$

## 4.1 Scanning Phase

During the scanning phase of the algorithm the input is repeatedly examined in order to construct the elementary patterns which will be deposited into the set *EP* (this set is implemented simply as a linked list of patterns). The components dominating the complexity of this phase are the for-loops of lines (2) and (4) in Figure 2. The operation of inserting an elementary pattern in the set *EP* in line (1) takes constant time while the initialization of the array *counts[]* is an $O(|\Sigma|)$ operation, which can also be considered constant time since our alphabet is fixed (it is, depending on the input sequences, either the set of nucleotides or the set of amino acids). As a result, all that is needed in order to determine the time required by the scanning phase is to determine how many times the two aforementioned for-loops are executed.

In order to facilitate the discussion that follows we introduce the notion of the *template*. Templates provide a conceptual way for constructing the elementary patterns. More specifically, we define an $(L, W)$ template to be an arbitrary string from the set $\{1, 0\}$ abiding to the following restrictions.

- it has length between $L$ and $W$ (inclusive),
- its has exactly $L$ '1's,
- it starts and it ends with '1'.

Notice that from an $(L, W)$ elementary pattern we can obtain an $(L, W)$ template if we turn every residue to '1' and every don't care character to '0'. Reversly, given an $(L, W)$ template we can produce $(L, W)$ patterns by "sliding" it over a sequence (see Figure 1 below): when the template is placed over any sequence position a pattern is obtained by maintaining all residues that are aligned with an '1' and turning into a don't care each residue aligned with a '0'.

**Lemma 3:** For any two integers $L$, $W$ (with $L \leq W$) let $A(L, W)$ denote the total number of $(L, W)$ templates. Then

$$A(L, W) = \binom{W-1}{L-1}$$

The for-loops in the lines (2) and (4) of Figure 2 force each input character to be examined some number of times. Disregarding any amount of pruning (which occurs when a pattern $P'\sigma$ in Figure 2 is found to have support less than $K$) each input character is visited no more than

$$A(1, W-L+1) + A(2, W-L+2) + \ldots + A(L, W) = \sum A(i, W-L+i)$$

times (this is actually an overestimate for the characters that are close to the beginning or the end of a sequence). That this claim is true can be verified by inspection of the function *Extend()* used by the code of the scanning phase.

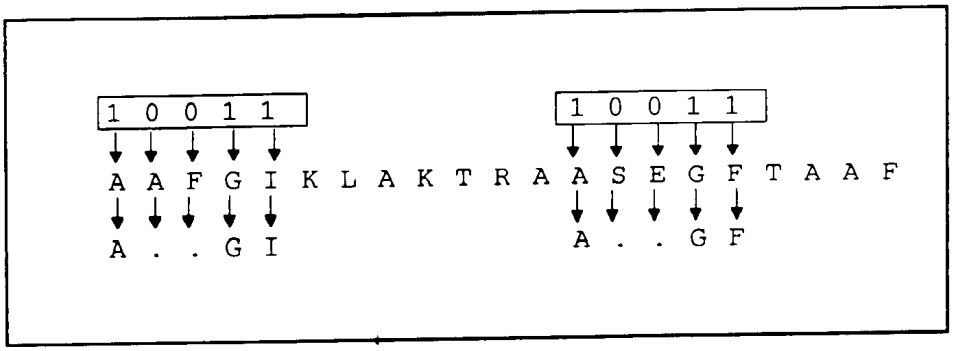


**Figure 3** : The template 10011 placed over the sequence AAFGIKLAKTRAASEGFTAAF at offsets 1 and 13. The solid lines indicate the alignment of the '1' characters and the residues of the sequence preserved in the resulting patterns. Broken lines point to those residues which are turned to don't care characters.

**Lemma 4:** The complexity of the scanning phase is $O(mW^L)$.

**Proof:** Each input character is scanned no more than $\sum A(i, W-L+i)$ times. Since the input contains $m$ characters, the total time spent in examining them is at most

$$m \sum A(i, W-L+i) \leq mLA(L, W) = mL\binom{W-1}{L-1} \leq mLW^{L-1} \leq mW^L$$

## 4.2 Convolution Phase

The convolution phase begins with a preprocessing step intended to facilitate the operations to follow. This preprocessing involves the ordering of the set of elementary patterns $EP$ and the generation of the directory structures $DirP$ and $DirS$. Since the complexity of these tasks depends on the size of the set $EP$, we first give an upper bound for $|EP|$:

**Lemma 5:** The number of elementary patterns discovered during the scanning phase is no larger than $m\binom{W-1}{L-1}/K$.

**Proof:** Consider the $j$-th residue of the $i$-th input sequence. There are at most $A(L, W)$ elementary patterns that can start at the offset $(i, j)$. Consequently, the offset $(i, j)$ appears at most $A(L, W)$ times in the offset lists of all the elementary patterns. Since the input has size $m$, the aggregate size of all the offset lists in $EP$ is then no larger than $mA(L, W)$. Furthermore, the minimum support requirement dictates that every offset list of an elementary pattern must have at least $K$ offsets. This means that

$$|EP| \leq mA(L, W)/K = m\binom{W-1}{L-1}/K$$

Before proceeding with the analysis of the preprocessing step let us give a quick description of the structures $DirP$ and $DirS$. These directory structures are designed for the efficient execution of the lines (5) and (7) in the pseudo-code. Each one of them is implemented as a balanced tree. In the case of $DirP$, every node of the tree is labeled with a string $w$ and contains a non-empty list of all the elementary patterns $P$ with $prefix(P) = w$. The patterns within every such list are ordered according to the $<_{pf}$ relationship. The implementation of $DirS$ is completely analogous, only that here within every node with label $w$ we find all the patterns $P$ satisfying $suffix(P)=w$. Using these structures (and with some minimal bookkeeping inside every pattern) it is possible to execute the lines (5) and (7) of Figure 2 in constant time.

**Lemma 6:** The time needed for the preprocessing of the set of elementary patterns $EP$ and the generation of the structures $DirP$ and $DirS$ is $O(W^4 mlogm)$.

**Proof:** Generating the structure $DirP$ involves (i) ordering the set $EP$ according to the $<_{pf}$ relationship and (ii) for every elementary pattern in $EP$ (considered in the $<_{pf}$ order) locating or creating the balanced tree entry labeled $prefix(P)$ and appending $P$ at the end of the corresponding list. The first of the above

steps requires, using any standard sorting algorithm, $O(|EP|log|EP|)$ comparison, each comparison in our case taking time $O(W)$. The second step takes for each pattern $P$ an (amortized) number of $O(log|EP|)$ comparisons for locating the node with label $prefix(P)$ and constant time for inserting $P$ into the node's list. Putting everything together and using Lemma 5 above, we have that the time needed for constructing $DirP$ is

$$O(W|EP| \log|EP|) = O(W\tfrac{m}{k}\binom{W-1}{L-1} \log \tfrac{m}{k}\binom{W-1}{L-1}) = O(W^L m \log m).$$

The structure $DirS$ is constructed in an analogous way and the same bounds hold there too.

After the preprocessing is done, the main body of the convolution phase commences. The complexity here is dominated by the time spent in the following two operations (the notation used refers to Figure 2): (i) convolutions of the pattern $T$ at the top of the stack with appropriate elementary patterns $Q$ and (ii) checking the resulting patterns $R$ for maximality. We will address each kind of operation separately, starting with the convolutions.

**Lemma 7:** Let $T$ be an arbitrary pattern at the top of the stack. The number of elementary patterns $Q$ that $T$ is convolved with (lines (6) and (8) of the pseudo-code) is no more than $2|\Sigma|W$.

**Proof:** When $T$ is extended to the left in line (6) of Figure 2, it has to be combined with an elementary pattern $Q$ such that $prefix(T) = suffix(Q)$. Given that the last $(L-1)$ residues of $Q$ are thus restricted and that an elementary pattern has size no more than $W$, it follows that there are at most $|\Sigma|W$ elementary patterns satisfying the requirement $prefix(T) = suffix(Q)$. A similar argument holds for the line (8) of the pseudo-code. In total, $T$ is conlolved with at most $2|\Sigma|W$ elementary patterns.

When the convolution $R = T \oplus Q$ is performed, the offset lists of the patterns $T$ and $Q$ must be compared in order to compute the offset list of the resulting pattern $R$. Because all offset lists are maintained ordered, this comparison can be done in a single traversal of the two lists, thus requiring $|L_S(T)|+|L_S(Q)| \leq 2m$ operations. Combining this with Lemma 7 above we get that

**Lemma 8:** For any pattern $T$ that ever appears on top of the stack the total time spent performing convolutions that involve $T$ is $O(Wm)$.

So, all that is needed in order to bound the time spent doing convolutions is to determine the number of patterns that are ever placed on top of the stack. As we are going to show in the full version of the paper, the only case that a non-maximal pattern $T$ can be placed on the stack is on the way to building the maximal pattern $T'$ that subsumes $T$. If we let $rc(T')$ denote the number of residues (i.e. the non-don't cares) in $T'$ then it can be shown that at most $(rc(T')-1)$ such non-maximal patterns $T$ can be placed at the top of the stack while building $T'$. Taking into consideration that every maximal pattern is also placed on the stack at some point and using Lemma 8 above we conclude

**Lemma 9:** The total time spent doing convolutions is $O(Wm \sum_{T' \text{ maximal}} rc(T'))$.

We now turn our attention to the operation of checking the patterns $R$ (the results of the convolutions in Figure 2) for maximality. The main data structure used here is the set *Maximal* which at any time contains all the maximal patterns which have been generated up to that time. The set *Maximal* can be thought of as a hash table (although it can also be implemented as a balanced tree for space efficiency). When a new pattern $R$ (with support at least $K$) is generated, the function $IsMaximal(R)$ is called in order to check if *Maximal* contains a pattern which subsumes $R$. Since this function is called very often, it is important that it is implemented efficiently. Towards that end, every pattern $R$ is assigned a hash value $H(R)$. Ideally, we would like these hash values to have the following properties: (i) no two maximal patterns take the same value and (ii) for every non-maximal pattern $R$ subsumed by the maximal pattern $R'$, $H(R) = H(R')$. In such a case and under the light of Theorem 1 in Section 3, checking a pattern $R$ for maximality is simply a question of computing the hash value $H(R)$ and inspecting the corresponding hash entry: $R$ is maximal if and only if that hash entry is empty.

Unfortunately we can construct reasonable hash functions which satisfy (provably) only the property (ii) above. It is, however, possible to design hash functions which almost always satisfy property (i) as well. One such function which has proved particularly appropriate (at least for all test cases we checked it for) maps every pattern $R$ to a pair

$H(R) = (|L_S(R)|, diff\_sum)$. In order to compute *diff_sum* we first "flatten out" all offsets in $LS(R) = \{(x_1, y_1),\ldots,(x_r, y_r)\}$ by mapping every offset $(x, y)$ to the integer $f(x, y) = y + \sum_{i=1}^{x-1} |s_i|$. The value *diff_sum* is then calculated as

$$diff\_sum = \sum_{i=1}^{r-1} (f(x_{i+1}, y_{i+1}) - f(x_i - y_i)).$$

**Lemma 10:** The time required for all the calls of the function *IsMaximal()* as well as for the insertion of all maximal patterns in the set *Maximal* (line (9) of the pseudo-code) is

$$O(W(Cm + t_H) \sum_{P \text{ maximal}} rc(P))$$

where $C$ is the average number of patterns found in a hash entry of *Maximal* and $t_H$ is the time needed for locating the hash entry corresponding to any given hash value.

**Proof:** (Sketch - the complete proof is deferred to the full version of the paper). Combining Lemma 7 with the fact that no more than $\Sigma_{P \text{ maximal}} rc(P)$ are ever placed on the top of the stack, we conclude that there are at most $W\Sigma_{P \text{ maximal}} rc(P)$ patterns generated by a convolution. In the worst case every such pattern $R$ will have to be checked, through a call to *IsMaximal(R)*, against the set *Maximal*. This involves first computing the hash value $H(R)$ (time $|L_S(R)| = O(m)$), second locating the hash entry in *Maximal* corresponding to $H(R)$ (time $t_H$) and, finally, comparing the offset list of $R$ with the offset list of every pattern found in that hash entry (on the average, time $Cm$). (We note here that if we select a balanced tree implementation for the hash table *Maximal* then the time required in order to locate the hash entry corresponding to any hash value $H(R)$ is $\log F$, where $F$ is the total number of maximal patterns. As a result, $t_H \leq \log F$.)

Putting now together Lemmas 4, 6, 9 and 10:

**Theorem 2:** The worst-case time complexity of the *TEIRESIAS* algorithm is

$$O(W^L m \log m + W(Cm + t_H) \sum_{P \text{ maximal}} rc(P)).$$

In the full version of the paper we also provide experimental results providing evidence that for all practical cases the value of the variable $C$ in the above bound is 1. These results are obtained by running *TEIRESIAS* on randomly selected families from the PROSITE database [10]. Furthermore, we argue why the first term of the bound given above is really not that bad. More specifically, it is more of an impediment of the worst-case type of analysis given here. In practice, it really depends on the size of the result set of maximal patterns.

# REFERENCES


[1] Rigoutsos, I. and Floratos, A., *"Motif Discovery in Biological Sequences Without Alignment or Enumeration"*, Proceedings 2nd ACM Annual Conference on Computational Molecular Biology (RECOMB), pp. 221-227, 1998.

[2] Rigoutsos, I. and Floratos, A., *"Combinatorial Pattern Discovery in Biological Sequences: The TEIRESIAS Algorithm"*, Bioinformatics, 14, 55-67, 1998.

[3] Waterman M.S., D.J. Galas, R. Arratia, *"Pattern Recognition In Several Sequences: Consensus And Alignment"*, Bulletin Of Mathematical Biology, 46, 515-527, 1984.

[4] Smith T.F., M.S. Waterman, and W.M. Fitch, *"Comparative Biosequence Metrics"*, Journal Of Molecular Evolution, 18, 38-46, 1981.

[5] Delcoigne A, and P. Hansen, *"Sequence Comparison By Dynamic Programming"*, Biometrika, 62, 661-664, 1975.

[6] Needleman S.B., and C.D. Wunsch, *"A General Method Applicable To The Search For Similarities In The Amino Acid Sequence Of Two Proteins"*, Journal Of Molecular Biology, 48, 443-453, 1970.

[7] Smith TF, and M.S. Waterman, *"Identification Of Common Molecular Subsequences"*, Journal Of Molecular Biology, 147, 195-197, 1981.

[8] Gibbs A.J., and G.A. McIntyre, *"The Diagram, A Method For Comparing Sequences"*, European Journal Of Biochemistry, 16, 1-11, 1970.

[9] Martinez H.M., *"An Efficient Method For Finding Repeats In Molecular Sequences."* Nucleic Acids Research, 11, 4629-4634, 1983.

[10] Bairoch A, P. Bucher, and K. Hofmann, *"The PROSITE database: its status in 1995."* Nucleic Acids Research, 24, 189-196, 1996.

[11] Carrillo, H. and Lipman, D., *"The Multiple Sequence Alignment Problem in Biology"*, SIAM Journal of Applied Mathematics, 1073-1082, 1988.

[12] Sobel, E. and Martinez, M., *"A Multiple Sequence Alignment Program"*, Nucleic Acids Research, pp. 363-374, 1986.

[13] Martinez, M., *"A Flexible Multiple Sequence Alignment Program"*, Nucleic Acids Research, pp. 1683-1691, 1988.

[14] Neville-Manning, C.G., Sethi, K.S., Wu, D. and Brutlag, D.L., *"Enumerating and Ranking Discrete Motifs"*, Intelligent Systems for Molecular Biology, 1997 (to be published).

[15] Wang, L. and Jiang, T., *"On the Complexity of Multiple Sequence Alignment"*, Journal of Computational Biology, pp. 337-348, 1994.

[16] Smith, R.F. and Smith, T.F., *"Automatic Generation of Primary Sequence Patterns from Sets of Related Protein Sequences"*, Nucleic Acids Research, pp. 118-122, 1990.

[17] Roytberg, M.A., *"A Search for Common Patterns in Many Sequences"*, CABIOS, pp. 57-64, 1992.

[18] Smith, T.F. and Waterman M.S., *"Identification of Common Molecular Subsequences"*, Journal of Molecular Biology, pp. 195-197, 1981.

[19] Smith, H.O., Annau, T.M., Chandrasegaran, S., *"Finding Sequence Motifs in Groups of Functionally Related Proteins"*, Proceedings of the National Academy of Sciences, pp. 826-830, 1990.

[20] Jonassen, I. Collins, J.F. and Higgins, D.G., *"Finding Flexible Patterns in Unaligned Protein Sequences"*, Protein Science, pp. 1587--1595, 1995.

[21] Neuwald, A.F. and Green, P., *"Detecting Patterns in Protein Sequences"*, Journal of Molecular Biology, pp. 698-712, 1994.

[22] Sagot, M.F. and Viari, A., *"A Double Combinatorial Approach to Discovering Patterns in Biological Sequences"*, Proceedings of the 7th Symposium on Combinatorial Pattern Matching, pp. 186-208, 1996.

[23] Wang, J., Marr, T.G., Shasha, D., Shapiro, B.A. and Chirn G., *"Discovering Active Motifs in Sets of Related Protein Sequences and Using them for Classification"*, Nucleic Acids Research, pp. 2769-2775, 1994.

[24] Wang, J., Chirn, G., Marr, T.G., Shapiro, B.A., Shasha, D. and Zhang, K., *"Combinatorial Pattern Discovery for Scientific Data: Some Preleminary Results"*, Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 115-124, 1994.

[25] Guan, X. and Uberbacher, E.C., *"A Fast Look-Up Algorithm for Detecting Repetitive DNA Sequences"*, Pacific Symposioum on Biocomputing, pp. 718-719, 1996.

[26] Califano, A. and Rigoutsos, I., *"FLASH: A Fast Look-Up Algorithm for String Homology"*, Proceedings First International Conference on Intelligent Systems for Molecular Biology, pp. 56-64, 1993.

[27] Benson, G. and Waterman, M.S., *"A Method for Fast Database Search for all k-nucleotide Repeats"*, Proceedings of the 2nd International Conference on Intelligent Systems for Molecular Biology, pp. 83-98, 1994.

[28] Wu, T.D. and Brutlag, D.L., *"Identification of Protein Motifs Using Conserved Amino Acid Properties and Partitioning Techniques"*, Proceedings of the 3rd International Conference on Intelligent Systems for Molecular Biology, pp. 402-410, 1995.

[29] Suyama, M., Nishioka, T. and Jun'ichi, O., *"Searching for Common Sequence Patterns Among Distantly Related Proteins"*, Protein Engineering, pp. 1075-1080, 1995.

[30] Sagot, M.-F., Viari, A. and Soldano, H., *"Multiple Sequence Comparison: A Peptide Matching Approach"*, Proceedings of the 6th Symposium on Combinatorial Pattern Matching, pp. 366-385, 1995.

[31] Henikoff, S. and Henikoff, J., *"Automatic Assembly of Protein Blocks for Database Searching"*, Nucleic Acids Research, Vol. 19, No. 23, pp. 6565-6572, 1991.

[32] Henikoff, S. and Henikoff, J., *"Protein Family Classification Based on Searching a Database of Blocks"*, Genomics, Vol. 19, pp. 97-107, 1994.