*Fundamenta Informaticae 71 (2006) 259–277*
*IOS Press*


# The Weighted Suffix Tree: An Efficient Data Structure for Handling Molecular Weighted Sequences and its Applications*

**Costas S. Iliopoulos**
*Department of Computer Science*
*King's College London, Strand, London WC2R2LS, England*
*csi@dcs.kcl.ac.uk*

**Christos Makris, Yannis Panagis†, Katerina Perdikuri, Evangelos Theodoridis**
*Department of Computer Engineering and Informatics*
*University of Patras, 26504 Patras, Greece*
*makri, panagis, perdikur, theodori@ceid.upatras.gr*

**Athanasios Tsakalidis**
*Research Academic Computer Technology Institute*
*N. Kazantzaki Str., Rio 26504 Patras, Greece*
*tsak@cti.gr*

**Abstract.** In this paper we introduce the Weighted Suffix Tree, an efficient data structure for computing string regularities in weighted sequences of molecular data. Molecular Weighted Sequences can model important biological processes such as the DNA Assembly Process or the DNA-Protein Binding Process. Thus pattern matching or identification of repeated patterns, in biological weighted sequences is a very important procedure in the translation of gene expression and regulation. We present time and space efficient algorithms for constructing the weighted suffix tree and some applications of the proposed data structure to problems taken from the Molecular Biology area such as pattern matching, repeats discovery, discovery of the longest common subsequence of two weighted sequences and computation of covers.

**Keywords:** Molecular Weighted Sequences, Suffix Tree, Pattern Matching, Identifications of repetitions, Covers.

*A preliminary version of the paper appears in [15, 13]

†Address for correspondence: Yannis Panagis, Department of Computer Engineering and Informatics, University of Patras, 26504 Patras, Greece

# 1. Introduction

Molecular Weighted Sequences appear in various applications of Computational Molecular Biology. A molecular weighted sequence is a molecular sequence (either a sequence of nucleotides or aminoacids), where each character in every position is assigned a certain weight. This weight could model either the probability of appearance of a character or the stability that the character contributes in a molecular complex.

Thus, in the first case a molecular weighted sequence can be the result of a DNA Assembly process. The key problem today in sequencing a large string of DNA is that only a small amount of DNA can be sequenced in a single read. That is, regardless of whether the sequencing is done by a fully-automated machine or by a more manually assisted method, the longest unbroken DNA substring that can be reliably determined in a single laboratory procedure is about 300 to 1000 (approximately 500) bases long [5, 6]. A longer string can be used in the procedure but only the initial 500 bases will be determined. Hence, to sequence long strings or an entire genome, the DNA must be divided into many short strings that are individually sequenced and then used to assemble the sequence of the full string. The critical distinction between different large-scale sequencing methods, is how the task of sequencing the full DNA is divided into manageable subtasks, so that the original sequence can be reassembled from sequences of length 500.

Reassembling DNA substrings introduces a degree of uncertainty for various positions in a biosequence. This notion of uncertainness was initially expressed with the use of "don't care" characters denoted as "$*$". A "don't care" character has the property of matching against any symbol in the given alphabet. For example the string $p = AC * C*$ matches the pattern $q = A * GCT$ under the alphabet $\Sigma = \{A, C, G, T, *\}$. In some cases though, scientists are able to go one step further and determine the probability of a certain character to appear at the position previously characterised as wildcard. In other words, a "don't care" character is replaced by a probability of appearance for each of the alphabet symbols. Such a sequence is modelled as a *weighted sequence*.

In the second case, a molecular weighted sequence can model the binding site of a regulatory protein. Each base in a candidate motif instance makes some positive, negative or neutral contribution to the binding stability of the DNA-protein complex [9, 22]. The weights assigned to each character can be thought of as modelling those effects. If the sum of the individual contributions is greater than a treshold, the DNA-protein complex can be considered stable enough to be functional.

Thus, we need new and efficient algorithms in order to analyze molecular weighted sequences. A fundamental problem in the analysis of Molecular Weighted Sequences is the computation of significant repeats which represent functional and structural similarities among molecular sequences. In [14] authors presented a simple algorithm for the computation of repeats in molecular weighted sequences. Although their algorithm is simple and easy to be implemented, it is not efficient in terms of space. In this paper we present an efficient algorithm, both in time and space limitations, to construct the Weighted Suffix Tree, an efficient data structure for computing string regularities in biological weighted sequences. The Weighted Suffix Tree, was firstly intoduced in [13]. In this work, which is primarily motivated by the need to efficiently compute repeats in a weighted sequence, we further extend the use of the Weighted Suffix Tree to other applications on weighted sequences.

There is a strong connection between our work and work related to *regulatory motifs* [26, 23, 31, 18] and *probabilistic suffix trees* [27, 30, 24]. The term regulatory motif is used to characterize short sequences of DNA that determine the timing location and level of gene expression. The approaches that

have been presented to extract regulatory motifs can be divided into two categories: the approaches that exploit word counting heuristics [17, 24] and the approaches that are based on the use of probabilistic models [10, 12, 21, 28, 32, 33]. In the second category of approaches the motifs are represented by position-probabilistic matrices, while the remainder of the sequences are represented by background models. This representation bears strong (though not absolute) similarity with our representation of weighted sequences. On the other hand, a probabilistic or *prediction* suffix tree is basically a stochastic model that employs a suffix tree as its index structure in order to compactly represent the distribution of conditional probabilities for a cluster of sequences. Each node of a probabilistic suffix tree is associated with a probability vector that stores the probability distribution for the next symbol, given the label of the node as the preceding segment. Algorithms that exploit probabilistic suffix trees to extract regulatory motifs can be found in [30, 24]. Our proposed weighted suffix tree differs from the probabilistic suffix tree in that:

- it does not model any stochastic process, since its edges and its nodes do not carry any information concerning the probabilistic distribution,
- it is designed for having the same functionality as an ordinary suffix tree, when handling weighted sequences and hence, supports various operations in optimal time and space complexities. The deployment of a probabilistic suffix tree (or of a suitable variant) to handle these operations could be inefficient, since the time complexities could be non-optimal and the extra modelling capabilities provided by the probabilistic suffix tree could be left unexploited.

Overall the two approaches (our structure and the probabilistic suffix tree) seem to solve different problems and their performance differs; it maybe, however, possible for a suitable combination of these two structures to be a prospective path of future research.

The remainder of this paper is structured as follows. In Section 2 we give all the basic definitions used in the rest of the paper, in Section 3 we present the Weighted Suffix Tree while in Section 4 we list a set of applications for the data structure. Finally in Section 5 we conclude and discuss our research interest in open problems of the area.

## 2. Preliminaries

Let $\Sigma$ be a finite alphabet which consists of a set of characters (or symbols). The cardinality of an alphabet, denoted by $|\Sigma|$, expresses the number of distinct characters in the alphabet. A *string* or *word* is a sequence of zero or more characters drawn from an alphabet. The set of all words over the alphabet $\Sigma$ is denoted by $\Sigma^+$. A word $w$ of length $n$ is represented by $w[1..n] = w[1]w[2]\cdots w[n]$, where $w[i] \in \Sigma$ for $1 \leq i \leq n$, and $n = |w|$ is the length of $w$. The empty word is the empty sequence (of zero length) and is denoted by $\varepsilon$; we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Moreover a word is said to be *primitive* if it cannot be written as $v^e$ with $v \in \Sigma^+$ and $e \geq 2$.

A subword $u$ of length $p$ is said to occur at position $i$ in the word $w$ if $u = w[i..i+p-1]$. In other words $u$ is a substring of length $p$ occurring at position $i$ in word $w$. A word has a *repeat* when it has two equal subwords.

A subword $u$ of $w$ is called a *cover* of $w$ if and only if $w$ can be constructed by concatenations and superpositions of $u$, so that every position of $w$ lies within some occurrence of $u$ in $w$. Two problems have been investigated in the computation of covers, known as the *shortest-cover* problem (finding the

Word $w$

| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | C | T | T | (A,0.5) | T | C | (A,0.5) | T | T | T |
| | | | | | (C,0.5) | | | (C,0.3) | | | |
| | | | | | (G, 0) | | | (G,0) | | | |
| | | | | | (T, 0) | | | (T,0.2) | | | |

Figure 1. Example of a weighted word with two weighted positions. Positions consisting of a single character indicate that this character appears with probability 1.

shortest cover of a given string of length $n$), and the *all-covers* problem(finding all the covers of a given string). Apostolico, Farach and Iliopoulos first introduced the notion of covers in [1] as well as that of *shortest-cover*, where a linear-time algorithm for this problem was presented.

In the case that for a given position of a word $w$ we consider the presence of a set of characters each with a given probability of appearance, we define the concept of a weighted sequence $X$.

**Definition 2.1.** A weighted sequence $X = X[1]X[2]\cdots X[n]$ is a sequence of positions, where each position $X[i]$ consists of a set of ordered pairs. Each pair has the form $(\sigma, \pi_i(\sigma))$, where $\pi_i(\sigma)$ is the probability of having the character $\sigma$ at position $i$. For every position $X[i]$, $1 \leq i \leq n$, $\sum_{\forall \sigma} \pi_i(\sigma) = 1$.

For example, if we consider the DNA alphabet $\Sigma$ = {A,C,G,T}, the sequence $X$ shown in Fig. 1 represents a word having 11 letters: the first four are definitely ACTT, the fifth can be either A or C each with 0.5 probability of appearance, letters 6 and 7 are T and C, and letter 8 can be A, C or T with probabilities 0.5, 0.3 and 0.2, respectively, and finally, letters 9 to 11 are T. Some of the words that can finally be produced are: $w_1 = ACTT\underline{A}TC\underline{A}TTT$, $w_2 = ACTT\underline{C}TC\underline{A}TTT$[1], etc. The probability of presence of a word is the cumulative probability, which is calculated by multiplying the relative probabilities of appearance of each character in every position. For the above example, $\pi(w_1) = \pi_1(A) * \pi_2(C) * \pi_3(T) * \pi_4(T) * \pi_5(A) * \cdots * \pi_8(T) = \pi_5(A) * \pi_8(A) = 0.25$. Similarly $\pi(w_2) = \pi_5(C) * \pi_8(A) = 0.25$. The definition of subword can be easily extended to accommodate weighted subsequences.

## 2.1. The Suffix Tree

The suffix tree is a fundamental data structure supporting a wide variety of efficient string processing algorithms. In particular, the suffix tree is well known to allow efficient and simple solutions to many problems concerning the identification and location either of a set of patterns or repeated substrings (contiguous or not) in a given sequence. The reader can find an extended literature on such applications in [11].

**Definition 2.2.** We denote by $T(w)$ the suffix tree of string $w$, as the compressed trie of all the suffixes of $w\$$, $\$ \notin \Sigma$. Let $L(v)$ denote the path-label of node $v$ in $T(w)$, which results by concatenating the edge labels along the path from the root to $v$. Leaf $v$ of $T(w)$ is labeled with index $i$ iff $L(v) = w[i..n]$. We define the leaf-list $LL(v)$ of $v$ as a list of the leaf-labels in the subtree below $v$.

Linear time algorithms for suffix tree construction are presented in [25, 35].

[1] underlined letters indicate the choice of a particular letter in a weighted position

# 3. The Weighted Suffix Tree

In this section we present a data structure for storing the set of suffixes of a weighted sequence with probability of appearance greater than $1/k$, where $k$ is a given constant. We use as fundamental data structure the suffix tree, incorporating the notion of probability of appearance for every suffix stored in a leaf. Thus, the introduced data structure is called the *Weighted Suffix Tree* (abbrev. WST).

The weighted suffix tree can be considered as a generalization of the ordinary suffix tree to handle weighted sequences. We give a construction of this structure in the next section. The constructed structure inherits all the interesting string manipulation properties of the ordinary suffix tree. However, it is not that straightforward to give a formal definition of it as it is with its ordinary counterpart. A quite informal definition appears next.

**Definition 3.1.** Let $X$ be a weighted sequence. For every suffix starting at position $i$ we define a list of possible weighted subwords so that the probability of appearance for each one of them is greater than $1/k$. We denote each of them as $X_{i,j}$, where $j$ is the subword rank in arbitrary numbering. We define $WST(X)$ the weighted suffix tree of a weighted sequence $X$, as the compressed trie of a portion of all the weighted subwords starting within each suffix $X_i$ of $X\$,\ \$ \notin \Sigma$, having a probability of appearance greater than 1/k. Let $L(v)$ denote the path-label of node $v$ in $WST(X)$, which results by concatenating the edge labels along the path from the root to $v$. Leaf $v$ of $WST(X)$ is labeled with index $i$ if $\exists j > 0$ such that $L(v) = X_{i,j}[i..n]$ and $\pi(X_{i,j}[i..n]) \geq 1/k$, where $j > 0$ denotes the $j$-th weighted subword starting at position i. We define the leaf-list $LL(v)$ of $v$ as a list of the leaf-labels in the subtree below $v$.

We will use an example to illustrate the above definition. Consider again the weighted sequence shown in Fig. 1 and suppose that we are interested in storing all suffixes with probability of appearance greater than a predefined parameter. We will construct the suffix tree for the sequence incorporating the notion of probability of appearance for each suffix.

For the sequence in Fig. 1 with $\pi(X_{i,j}) \geq 1/4$, for an appropriate subword $X_{i,j}$, we have the following possible prefixes for every suffix:

- Prefixes for suffix $X[1..11]$: $X_{1,1}$=$ACTT\underline{A}TC\underline{A}TTT$, $\pi(X_{1,1}) = 0.25$, and $X_{1,2} = ACTT\underline{C}TC\underline{A}TTT$, $\pi(X_{1,2}) = 0.25$.
- Prefixes for suffix $X[2..11]$: $X_{2,1} = CTT\underline{A}TC\underline{A}TTT$, $\pi(X_{2,1}) = 0.25$, and $X_{2,2} = CTT\underline{C}TC\underline{A}TTT$, $\pi(X_{2,2}) = 0.25$, etc.

The weighted suffix tree for the above subwords appears in Fig. 2.

## 3.1. Construction of the WST

In this paragraph we describe an efficient algorithm for constructing the WST for a given weighted sequence $X = X[1..n]$, of length $n$. Firstly, we describe the naive approach, which is quadratic in time. As already discussed, the weighted suffix tree is is a generalized suffix tree (GST) consisting of all subwords with probability of appearance greater than $1/k$, $k$ is a given constant. The weighted suffix tree can be built as follows.

**Step 1:** For each $i$, $(1 \leq i \leq n)$, generate all possible weighted suffixes of the weighted sequence with probability of appearance greater than $1/k$.

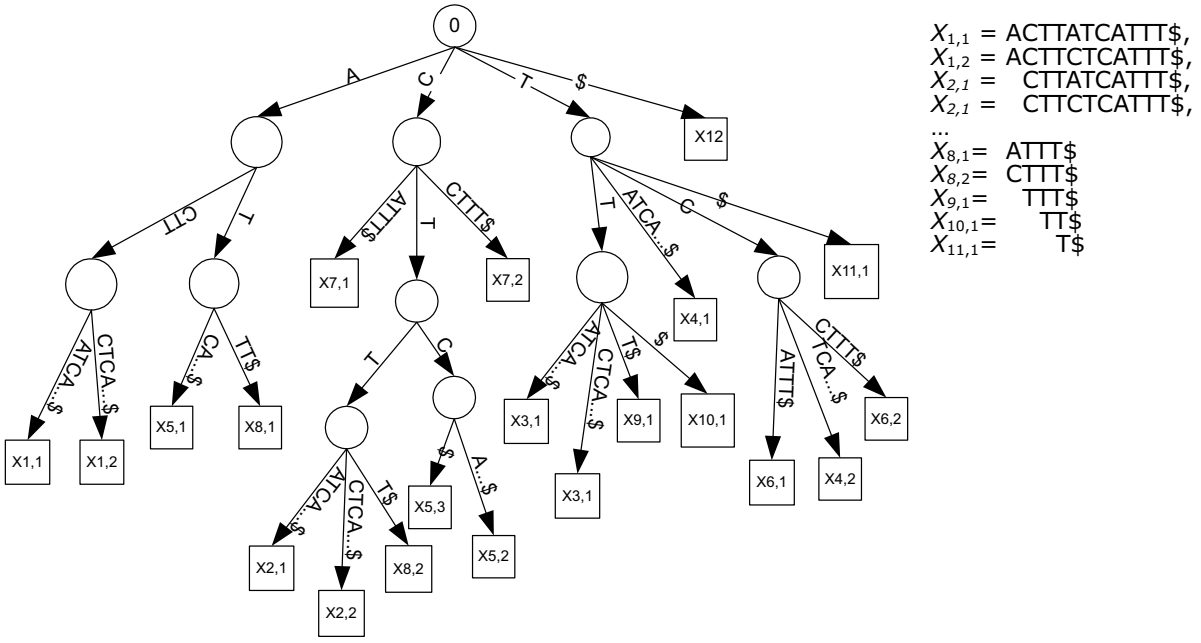


Figure 2. A Weighted Suffix Tree example.

**Step 2:** Construct the Generalized Suffix Tree, for the list of all possible weighted suffixes.

A naive analysis would conjecture that each suffix starting at a certain position $i$ could possibly continue until the end of the string. Thus, suffix $i$ can have length up to $n - i + 1$. If in step 1 all the sequence positions have to be examined then step 1 costs $O(n^2)$ and up to $O(n)$ discrete strings are generated. The second observation leads to an $O(n^2)$ upper bound for step 2, as well.

The above naive approach is not optimal since the time for construction is $O(n^2)$. In the following paragraphs we present an alternative efficient approach. Our method consists of three phases: **Coloring**, **Generation** and **Construction**. The exact steps of our methodology for construction are:

**Coloring:** Scan all the positions $i$ ($1 \leq i \leq n$) of the weighted sequence and mark each one according to the following criteria:

- mark position $i$ *black*, if *none* of the possible characters, listed at position $i$, has probability of appearance greater than $1 - 1/k$,
- mark position $i$ *gray*, if *one* of the possible characters listed at position $i$, has probability of appearance greater than $1 - 1/k$,
- and finally mark position $i$ *white*, if *one* of the possible characters has probability of appearance *equal to* 1.

Note that the following facts hold:

1. Only one possible character appears at white positions, thus we can call them *solid* positions.

2. At black positions since no character appears with probability greater than $1-1/k$, more than one character appear with probability greater than $1/k$ hence we can call them *branching* positions.
3. At gray positions, only one character eventually survives, since all the possible characters with the exception of one, have probability of appearance less than $1/k$, which implies that they can not produce an eligible subword (i.e. $\pi(subword) \geq 1/k$).

During the first step we maintain a list $B$ of all black positions. **Coloring** is formally presented in Algorithm 1.

**Algorithm 1** Coloring

```
1: input Weighted Sequence X
2: n ← |X|
3: B: a list of integers
4: COLOR[n]: array of n integers
5: for i ← 1 to n do
6:    if ∃σ ∈ Σ : π_i(σ) = 1 then
7:       COLOR[i]← 0 /* 0 ⇒ white */
8:    else if ∃σ ∈ Σ : π_i(σ) > 1 − 1/k then
9:       COLOR[i]← 1 /* 1 ⇒ gray */
10:   else
11:      COLOR[i]← 2 /* 2 ⇒ black */
12:      B.add_back(i) /* add to the right end of the list */
13:   end if
14: end for
15: return COLOR, B
```

**Generation:** After the *coloring* phase, the *generation* phase begins having as input the weighted sequence, the colors' array and the list $B$. At this phase all the substrings that satisfy the probability of appearance constraint are generated. In the current phase all the positions in list $B$ are scanned from left to right. At each black position $i$ a list of possible subwords starting from this position is created. The production of the possible subwords is done as follows: moving rightwards, we extend the current subwords by adding the same single character whenever we encounter a white or gray position, only one possible choice, and creating new subwords at black positions, where potentially many choices are provided. The process is illustrated in Fig. 3.

At this point we define for every produced subword two cumulative probabilities $\pi',\ \pi''$. The first one measures the actual subword probabilities and the second one is defined by temporarily treating gray positions as white[2]. The generation of a subword stops when it meets a black position and $\pi''$ (which effectively ignores gray positions) has reached the $1/k$ threshold. We call this position *extended position* (See Fig. 4). Notice that the actual subword may actually be shorter as $\pi'$ (which incorporates gray positions) may have met the $1/k$ threshold earlier. For every subword we store the difference $D$ of the actual ending position and the extended one, as shown in Fig. 5. Notice

[2]The reason for this special behavior is justified by Remark 3.2.

that only the actual subwords need to be represented with the GST. The extension is executed by iteratively applying Algorithm 3. A formal treatment of **Generation** is given in Algorithm 2.

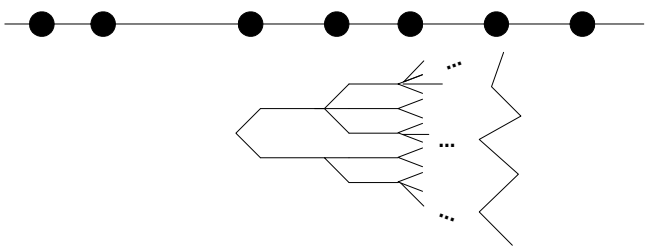

Figure 3. Producing all possible subwords from left to right

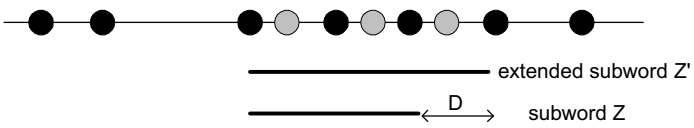


Figure 4. The extended and actual subwords

**Construction:** Having produced all the subwords from every black position, we insert the actual subwords in the generalized suffix tree in the following way (See Algorithm 4). For every subword we initially insert the corresponding extended subword in the GST and then remove from it the redundant portion $D$. To further illustrate the case, suppose that $Z' = X[i..\ i+f'-1]$ is the extended subword of the actual subword $Z = X[i..\ i+f-1]$ ($f \leq f'$) that begins at black position $i$ of the weighted sequence $X$ in Fig. 5. Observe the following two facts:

**Remark 3.1.** There is no need to insert every suffix of $Z$ in the GST apart from those starting to the left of the next black position $i'$, as all the other suffixes will be taken into account when **Generation** phase processes $i'$.

**Remark 3.2.** A suffix of $Z'$ can possibly extend to the right of position $i+f-1$, where the actual subword ends, since $\pi'$ does not take gray positions into account (cf. Fig. 5). No suffix can end though, at a position greater than $i+f'-1$, where the extended subword ends.

Let $D_j$ denote the redundant portion of suffix $Z'[i+j..i+f'-1]$ of $Z'$ (cf. Fig. 5). We can compute $D_j$'s by applying an iterative method. More specifically, $D_{j+1}$ can result from $D_j$ and the actual probability of appearance $\pi'$ of $Z'[i+j..i+f'-1]$ using Eqn. 1.

$$D_{j+1} = \begin{cases} D_j & \text{if } color_{i+j} = white \\ D_j - \lambda_j & \text{if } color_{i+j} = gray \end{cases} \tag{1}$$

**Algorithm 2** Generation

```
1: input Weighted Sequence X, array COLOR, list B
2: LT: list of tries
3: for i ← B.front() to B.back() do
4:    /* Scan list from left to right */
5:    T: a trie of generated substrings
      /* Each node consists of a float π', a float π'' and an integer D */
6:    counter ← 0
7:    for all σ ∈ Σ : π_i(σ) ≥ 1/k do
8:       Extend(T, root, σ, π_i(σ),COLOR[i])
         /* see Algorithm 3 for implementation of the Extend method */
9:       counter ← counter+1
10:   end for
11:   j ← i;
12:   while counter > 0 do
13:      /* counter keeps the # of extensions. If no extension occurs during a step, while loop ends */
14:      j ← j+1;
15:      counter ← 0
16:      for all leaves lv of T do
17:         for all σ ∈ Σ : π_j(σ) ≥ 1/k do
18:            Extend(T, lv, σ, π_j(σ), COLOR[j]) /* see Algorithm 3 */
19:            counter ← counter+1
20:         end for
21:      end for
22:   end while
23:   LT.add_back(T)
24: end for
25: return LT
```

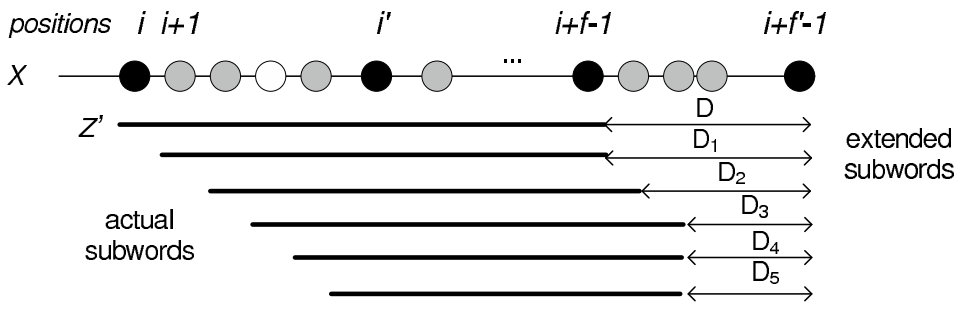


Figure 5. Insertion of subwords in the GST

**Algorithm 3** Extend

**input** a trie $T$, a trie node $x$, a char $\sigma$,a float $\pi_i(\sigma)$, an integer $color$
**if** $x.\pi'' * \pi_i(\sigma) \geq \frac{1}{k}$ **then**
  **Create** a new node $y$ under node $x$ and a connecting edge $e$.
  label $e$ with character $\sigma$
  $y.\pi' \leftarrow x.\pi' * \pi_i(\sigma)$
  **if** $color$=0 or $color$=1 **then**
    $y.\pi'' \leftarrow x.\pi'' * 1$
    **if** $x.D > 0$ **then**
      $y.D \leftarrow x.D + 1$
    **else**
      $y.D \leftarrow 0$
    **end if**
  **else**
    $y.\pi'' \leftarrow x.\pi'' * \pi_i(\sigma)$
    $y.D \leftarrow 0$
  **end if**
**end if**
**return**

We estimate the value of $\lambda_j$ by dividing the $\pi'(Z'[i+j..i+f'-1])$ with $\pi_i(Z'(i+j))$ and then by moving to the right of position $i+f-1$ (namely to the rightmost positions where the actual subword of $Z'[i+j..i+f'-1]$ ends) and multiplying with the probabilities of occurrences until the $\frac{1}{k}$-constraint is violated. In other words, the extension stops when the right end of the actual subword of $Z'[1..i+f'-1]$ has been reached. The number of extensions dictates the value of $\lambda_j$. The total cost for computing all $D_j$'s is bounded by $O(D+k)$, where $k$ is the number of proper suffixes of $Z' = X[i..\ i+f'-1]$ that are going to be inserted in the GST during this step.

After we have inserted the extended subword $Z' = X[i..i+f'-1]$ and the proper suffixes using McCreight's algorithm[25], we have to remove all the $D_j$'s from the GST. Starting from the leaf corresponding to the entire $Z'$, we move upwards the tree by $D$ characters. Without loss of generality suppose that this movement ends at an edge. At the current position we eliminate the extra portion of $Z'$, by splitting the edge and by creating a new leaf. By means of the previous operations, we have established the insertion of the actual subword $Z$ in the GST. The next redundancy of length $D_1$ has to be removed by creating a new leaf at the end of path $Z'[2..|Z'|] = X[i+1..i+f-1]$. We locate the end of this path using the suffix link of the node last traversed in the previous upward movement. During the last edge-split the label had been divided in two parts $z_1$ and $z_2$ (cf. Fig 6). By following the suffix link and then moving upwards by $|z_2|$ characters we approach the exact position of $Z'[2..|Z'|] = X[i+1..i+f-1]$. Then by moving downwards by $\lambda_1 = D - D_1$ characters we locate the path-end of the actual subword of the first proper suffix of $Z'$. Let $\lambda_d = |D_{d-1}| - |D_d|,\ d > 1$ and $\lambda_1 = D - D_1$. At this position we probably create a new node (if we end in the middle of an edge), we set the suffix link of the previously created node and create a new leaf of this suffix of $Z'$. At this point we discard the old leaf. We continue the elimination procedure for the remaining suffixes of $Z'$, as outlined above.

The entire process costs at most $\sum_{d>0} \lambda_d = O(D)$, which in turn is the time required to complete the suffix tree construction.

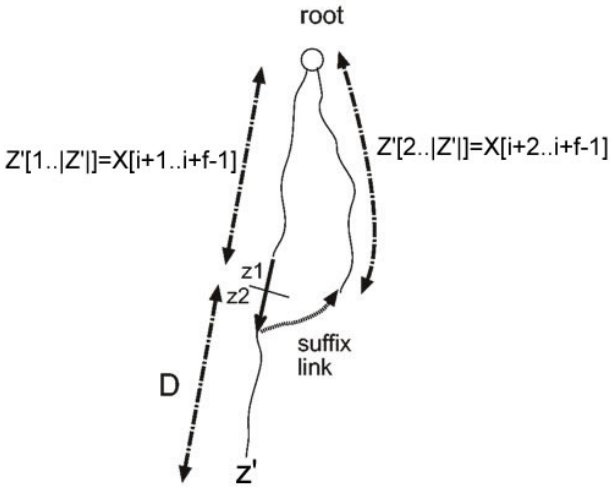


Figure 6. Removing the redundancies

From the above discussion we can derive the following lemma.

**Lemma 3.1.** The computational cost for each extended subword $Z'$ to calculate the redundancies $D_j$ and to remove them from the GST is $O(D + k)$, where $k$ is the number of proper suffixes of $Z'$ that are finally inserted.

We outline the construction algorithm of WST in Algorithm 4.

**Algorithm 4** Construction

```
1: input Weighted Sequence T, list of tries LT, array COLOR
2: WST: a generalized suffix tree
3: for all leaves lv in LT do
4:    Z' ← path label of lv
5:    compute redundancies D_j
6:    insert Z' in WST
7:    restore the actual subwords of Z' and its proper suffixes in the WST
8: end for
9: return GST
```

**Note:** The above description implicitly assumes that there are no positions $i$ where $\pi_i(\sigma) < 1/k, \forall \sigma \in \Sigma$. If this is not the case, the sequence can be divided into subsequences where this assumption holds and these subsequences can be separately processed, according to the previous algorithms.

### 3.2. Correctness of the Construction Algorithm

The correctness of the construction algorithm is evident by the following lemma.

**Lemma 3.2.** The phase **Construction** creates a set of leaves in the GST for every actual subword with probability of occurrence greater than $\frac{1}{k}$.

**Proof:**
The lemma is derived by the observation that an actual subword begins either at a black or a gray/white position. The actual subword is going to be inserted directly in the GST, by an insertion of an extended subword in the former case and by an insertion of a proper suffix of the in question extended subword in the latter case. Hence, no subword meeting the probability of appearance constraint is going to be omitted.

□

### 3.3. Time and Space Analysis on the Construction of the WST

The time and space complexity analysis for the construction of the WST is based on the combination of the following lemmas:

**Lemma 3.3.** At most $O\left(|\Sigma|^{\log k/\log(\frac{k}{k-1})}\right)$ subwords could start at each branching position $i$ ($1 \leq i \leq n$) of the weighted sequence.

**Proof:**
Consider for example position $i$ and the longest subword $u$ which starts at that position. If we suppose that $u$ is $\lambda$ characters long, its cumulative probability will be $\pi(u[1..\ \lambda]) = \pi_i(u[1]) * \pi_{i+1}(u[2]) * \cdots * \pi_{i+\lambda-1}(u[\lambda])$. In order to produce this subword we have to pass through $l$ black positions of the weighted sequence. Recall that at black positions none of the possible characters has probability of appearance greater than $\hat{\pi} = 1 - 1/k$. If we assume that there are no gray positions that could reduce the cumulative probability, then $\pi(u[1..\lambda]) \leq \hat{\pi}^l$ (taking only black positions into account). In order to store this subword its cumulative probability is $\hat{\pi}^l \geq 1/k$ and thus $l \leq \log k/\log(\frac{k}{k-1})$, by taking logarithms (all logarithms are $\log_2$).

Thus, regardless of considering or not the gray positions, $u$ includes at most $l = O(1)$ black positions, or in other words, positions where new subwords are produced. Hence, every position $i$ of the weighted sequence can be the starting point of at most $|\Sigma|^l$ number of subwords. This concludes the proof □

For example, typical values of $l$ are $\cong 21.9$ for $k = 20$ and $\cong 1046$ for $k = 200$.

**Lemma 3.4.** The number of subwords with probability greater than or equal to $1/k$ is at most $O(n)$.

**Proof:**
Since every position $i$ of the weighted sequence is the starting point of $|\Sigma|^l$ subwords (Lemma 3.3), the total number of subwords is $O(n)$. □

**Lemma 3.5.** The phase **Generation** of the construction algorithm takes $O(n)$ time.

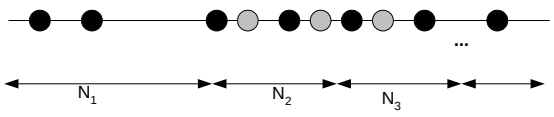


Figure 7. Time cost for step 2

**Proof:**
Suppose that the weighted sequence is divided into windows $N_j, j \geq 1$ (cf. Fig. 7). Each window contains $l = \log k / \log(\frac{k}{k-1})$ black positions. Notice that a window can contain more than $l$ positions of all types and that $\sum_{j\geq 1} |N_j| = n$. Lets consider window $N_i$. **Generation** scans the black positions inside $N_i$. Every black position will generate $O(1)$ subwords (according to Lemma 3.3) and none of them is going to exceed window $N_{i+1}$ because it can not be extended to more than $l$ black positions. Thus, the length of subwords will be at most equal to $|N_i| + |N_{i+1}|$. Thus, for the window $N_i$, **Generation** costs at most $O\left(l|\Sigma|^l(|N_i| + |N_{i+1}|)\right) = O(|N_i| + |N_{i+1}|)$ time. Summing up the costs for all windows we conclude that **Generation** incurs a total of $O\left(\sum_{\forall i}\left(|N_i| + |N_{i+1}|\right)\right) = O(n)$ cost. □

**Lemma 3.6.** Phase **Construction** of the construction algorithm takes $O(n)$ time.

**Proof:**
Consider again the windows scheme as in the previous lemma and in particular window $N_i$. In **Construction** we insert the extended subwords in the WST that correspond to that window. Each one of them has length at most $|N_i| + |N_{i+1}|$. The cost to insert those extended subwords in the WST using McCreight's algorithm [25] is no more than $O(l \cdot (|N_i| + |N_{i+1}|)) = O(|N_i| + |N_{i+1}|)$ and the cost to repair the WST (as we described in **Construction** phase Lemma 3.1) is $O(l \cdot D + k)$. Observe that $D$ is always smaller than $|N_i| + |N_{i+1}|$ and $\sum k_i$ for all extended subwords is bounded by $n$ thus for window $N_i$ **Construction** costs $O(|N_i| + |N_{i+1}| + k_i)$ time. Summing the costs for all windows, **Construction** yields $O\left(n + \sum k_i\right) = O(n)$ time in total. □

Based on the previous lemmas we derive the following theorem.

**Theorem 3.1.** The time and space complexity of constructing the WST is linear to the length of the weighted sequence.

**Proof:**
The WST, which is a compact trie data structure, stores $O(n)$ subwords (by Lemma 3.4) and thus the space is $O(n)$. None of the three construction steps takes more than $O(n)$ time so the total time complexity is $O(n)$. □

**Remark 3.3.** In the preceding discussion we tacitly assumed $l = \log k / \log(\frac{k}{k-1})$ and $|\Sigma|^l$ to be constants. This assumption is sound in the presence of a constant-size alphabet (like the DNA alphabet) and whenever $k$ is also a constant independent of $n$. This is indeed the case in the sequences that we consider, where the size of the sequence $n$ is huge in comparison to the alphabet size or the probability threshold.

# 4. Applications

The Weighted Suffix Tree finds use in interesting biological applications, since molecular weighted sequences can model important biological processes such as the DNA assembly process or the DNA-protein binding process. Moreover, in a series of applications one often needs to handle protein families (or superfamilies - see Gusfield [11] section 14.3) where a family is a set of proteins having similar biological functions or similar two- or three-dimensional structures. Examples of such families include the *globins* and the *immunoglobulin* proteins. In these families it is usually more desirable to handle the various string manipulation problems not by processing each string separately but by using a unique representation that models the whole family. String manipulation problems that are met in these applications are: pattern matching, repeats discovery, discovery of the longest common subsequence of two sequences and computations of covers. In [3, 7] a set of theoretical and experimental results are presented on algorithms designed to handle these problems for molecular weighted sequences representing synthetic and real data. In the sequel we will present a set of algorithms for handling efficiently the above problems by employing the Weighted Suffix Tree.

In the following sections we present applications of the Weighted Suffix Tree, namely: pattern matching in weighted sequences, computing repeats in weighted sequences, detection of the longest common subsequence in weighted sequences and computation of covers in weighted sequences.

## 4.1. Pattern Matching in Weighted Sequences

The classical pattern matching can be reformulated in weighted sequences as follows:

**Problem 1**. *Given a pattern $p$ and a weighted sequence $X$, find the starting positions of $p$ in $X$, each with probability of appearance greater than $1/k$ .*

**Solution**. Firstly, we build the WST for $X$ with parametre $k$. We distinguish two cases. If $p$ consists entirely of non-weighted positions we spell $p$ from the root of the tree until at an internal node $v$, either we have spelled the entire $p$, in which case we report all items in $LL(v)$, or we cannot proceed any further and thus we report failure. If $p$ contains weighted positions we decompose it into solid patterns each with $\Pr\{\text{occurence}\} \geq 1/k$ and match each one of them using the above procedure. Apparently, pattern matching can be solved in $O(m + \alpha)$ time, $m = |p|$ and is $\alpha$ the output size, with $O(n)$ preprocessing.

## 4.2. Computing the Repeats

A lot of work has been done for identifying the repeats in a word. In [2, 8] and [29], authors have presented efficient methods that find occurrences of squares in a string of length $n$ in $O(n \log n)$ time plus the time to report the detected squares. Moreover in [19] authors presented efficient algorithms to find maximal repetitions in a word. In the area of computational biology, algorithms for finding identical repetitions in biosequences are presented in e.g. [20] and [34].

Using the WST we can compute in linear time the repeats of a weighted sequence $X$. In particular, we compute the repeats of all subwords $u$, with $\Pr\{u\} \geq 1/k, \ \forall u$. This version of the problem is of particular biological interest.

**Problem 2**. *Given a weighted sequence $X$ and an integer $k$ find all the repeats of all possible words having a probability of appearance greater than $1/k$ .*

**Solution**. We build the WST with parametre $k$ and traverse it bottom-up. At each internal node $v$, with $|LL(v)| > 1$ we report the items of $LL(v)$, in pairs. This process requires $O(n)$ time by Lemma 3.4

In the example shown in Fig. 1 and Fig. 2, the longest repeat is the word CTT, which appears in suffixes: $(X_{2,1}, X_{8,2}), (X_{2,2}, X_{8,2})$ (with probability greater than 1/4). The time required by the solution is $O(n + \alpha)$, where $\alpha$ denotes the output size.

**Remark 4.1.** Apart from the repeats problem the repetitions detection in weighted molecular sequences can be solved in $O(n \log n + \alpha)$ time, by appropriately extending either of the approaches in [4, 29].

### 4.3. Longest Common Substring in Weighted Sequences

A classical problem in string analysis is to find the longest common substring of two given strings $X_1$ and $X_2$. Here we reformulate the *longest common substring problem* for weighted sequences.

**Problem 3**. *Given two weighted strings $X_1$ and $X_2$, find the longest common substring with probability of appearance greater than $1/k$ in both strings.*

**Solution**. An efficient and simple way to find the longest common substring in two given weighted strings $X_1$ and $X_2$ is to build a generalized weighted suffix tree for $X_1$ and $X_2$. The path label of any internal node is a substring common to both $X_1$ and $X_2$ with probability of appearance greater than $1/k$. The algorithm merely finds the node with greatest string-depth and having leafs that contain indexes from both sequences. A preorder traversal of the WST suffices to compute the longest string-depth (for details see [11]). It is easily derived that the above procedure runs in $O(n)$ time.

### 4.4. Computing the Covers in a Weighted Sequence

In this section we address the problem of computing the set of covers in a weighted sequence. In a more formal manner the problem can be defined as:

**Problem 4** *Given a weighted sequence $X$ of length $n$ and an integer k, find all possible covers of $X$ that have probability of appearance larger than* 1/*k*.

Using the WST we can compute in $O(n)$ time the covers of a weighted sequence. All proper covers of $X$ along with $X$ itself compose the set of covers of the weighted sequence.

**Solution**. First of all we will provide a methodology that uses the Suffix Tree for finding all covers of an ordinary string $w$.

We use an array $G[1..n]$ which we will call *gap array*. An arbitrary array position $i$ contains a pointer to a record $r_i$. All records form together a doubly-linked list. Elements of the list are positions in the string that have not been visited thus far. Initially, the list contains $n$ elements but this number decreases over time. Each node-record $r_i$ contains three fields: a field $posn$ storing $i$, fields $succ$ and $pred$ that contain pointers to the next and previous items in the list. We further need to use two more variables $max$ and $last$, initialised to 0 and $n$, respectively.

The algorithm starts at the root and follows the path to suffix $w[1..n]$. At each intermediate node $v$, for each sibling $u$ of $v$ we collect the items in leaf-list $LL(u)$. For each such item $j$ we reach $G[j]$ and follow the pointer to $r_j$. Then the algorithm accesses $p := r_j.pred$ and $s := r_j.next$. The new gap generated after removing $r_j$ from the list is simply $s.posn - p.posn$. If the latter is greater than $max$ then we update $max$. Furthermore, if $j = last$ we update last to $r_j.pred.posn$. In order to check whether $L(v)$ is a cover we check the validity of Remark 4.2.

**Remark 4.2.** Once at an internal node $v$ of the suffix tree, $L(v)$ is a cover whenever $max < |L(v)|$ and $n - last < |L(v)|$.

If the conditions in Remark 4.2 do not hold, the procedure stops. Note that there is no need to proceed any deeper in the tree since, as we move from the root to the leaves, the leaf-lists become more sparse and thus the maximum gap can only increase. Moreover, once the maximum gap is larger than the length of the path label we can no longer expect to find gaps inside the sequence. It is also true that sometimes a cover falls in the middle of an edge. Therefore, we need to spell each symbol on every edge separately. Nevertheless, this operation does not increase the overall complexity.

**Observation 1.** There is no need to explicitly keep the elements of $LL(u)$ for an internal node $u$. We can infer the latter if we have precomputed at each internal node $u$, two pointers to the leftmost and rightmost leaves of $u$'s subtree. At the point where traversal is needed we access the rightmost leaf and traverse leaves rightwards until we reach the righmost pointer.

**Theorem 4.1.** Computing all covers requires $O(n)$ time.

**Proof:**
The algorithm follows a single path from root to the leaf storing suffix $w[1..n]$. At each transition from a node $v$ to its child on the path to $w[1..n]$ a number of operations need to be made to the gap array $G$. The number of operations equals the cardinality of each $LL(u)$, $\forall u = \text{sibling}(v)$. Each of these insertions only updates pointers thus it costs $O(1)$. The item of each $LL$ is only once considered. Furthermore, each of the $n$ positions in $w$ occurs only once within a leaf-list, thus it causes an operation in $G$, only once. After the operation in $G$ and the following list deletion, a predecessor and successor operations have to be performed each of which also cost $O(1)$ time. Hence, our algorithm consists of at most $O(n)$ steps each costing $O(1)$ for a total of $O(n)$ time complexity. □

The above discussion is based on ordinary Suffix Trees. It is not difficult however to generalize the results to the WST and weighted sequences. First of all we build WST$(X)$ with parametre $k$ for the sequence $X$ in which every subword appears with probability above $1/k$. Instead of one path we will have to visit up to $|\Sigma|$ different paths, namely $X_{1,1}[1..n], \ldots, X_{1,|\Sigma|}[1..n]$. A common cover for all sequences is defined on the path from the root to the nearest common ancestor of the $|\Sigma|$ leaves. Since our algorithm is top-down we can detect the node where the different paths split. If the paths to the different leaves split at the root then the cover algorithm simply detects covers but at individual sequences this time.

## 4.5. Motif Extraction from Weighted Sequences

Recently, some of the authors proposed in [16] methods for extracting motifs from weighted sequences. They make heavy use of the WST as an underlying indexing structure and propose new methods to solve motif inference problems. More precisely they achieve the following results:

- Given a set of $N$ weighted sequences $S = X_1, X_2, \ldots, X_n$, a small integer $k \geq 0$ and a quorum $q \leq N$, they find in $O(Nn \log(Nn) + \alpha)$ time, all maximal pairs $m$ such that each component of $m$ appears with probability $\geq 1/k$ and with no overlaps in at least $q$ sequences of the set $S$, where $\alpha$ is the size of the size of the answer.

- Given a set of $N$ weighted sequences $S = X_1, X_2, \ldots, X_n$, a small integer $k \geq 0$ and a quorum $q \leq N$, they can find in $O(Nn \log(Nn) + \alpha)$ time all maximal pairs $m$ such that each component of $m$ appears with probability $\geq 1/k$ and the gap is bounded by the constant $b$, in at least $q$ sequences of the set $S$, where $\alpha$ is the size of the output.

- Given a weighted sequence $X$, they can find in $O(nqV(e,l)) + \alpha$ time and $O(n\frac{N}{w})$ space the motifs repeated at least $q$ times, so that each component of motif appears with probability $\geq 1/k$, where $\alpha$ is the size of the output, $w$ the word size and $V(e,t) = \sum_{j=0}^{e} \binom{t}{j}(|\Sigma| - 1)^j \leq t^e|\Sigma|^e$.

- Given a set of $N$ weighted sequences $S = X_1, X_2, \cdots X_n$, a small integer $k \geq 0$ and a quorum $q \leq N$, they can find in $O(nV^2(e,t)q \log\log n)$ time all common motifs such that each motif $m$ appears with probability $\geq 1/k$, in at least $q$ sequences of the set $S$, where $\alpha$ is the size of the output and $V(e,t) = \sum_{j=0}^{e} \binom{t}{j}(|\Sigma| - 1)^j \leq t^e|\Sigma|^e$.

## 5. Conclusions

In this paper we have presented the Weighted Suffix Tree, an efficient data structure solving a wide range of problems in weighted sequences such as: pattern matching, identification of repeats, longest common substring in weighted molecular sequences, and computation of covers.

Our future direction is focused on using the WST for computing string regularities (like for example borders and palindromes) on weighted biological sequences. Some immediate applications in molecular biology include: using sequences containing degenerate bases, where a letter can replace several bases (for example, a B will represent a G, T or C and a H will represent A, T or C); using logo sequences which are more or less related to consensus: either from assembly or from blocks obtained by a multiple alignment program; analysis of DNA micro-arrays where expression levels of genes are recorded under different experiments.

Finally, we believe that the Weighted Suffix Tree can also be used in the analysis of weighted sequences in other applications of computer science. Weighted Sequences also appear in the field of event management for complex networks, where each event has a timestamp. The latter problem was formulated and studied in [36].

## References

[1] Apostolico, A., Farach, M., Iliopoulos, C.: Optimal superprimitivity testing for strings, *Information Processing Letters*, **39**, 1991, 17–20.

[2] Apostolico, A., Preparata, F.: Optimal off-line detection of repetitions in a string, *Theoretical Computer Science*, **22**, 1983, 297–315.

[3] Arikat, F., Bakalis, A., Christodoulakis, M., Iliopoulos, C., Mouchard, L.: Experimental Results in Pattern Matching on Weighted Sequences, *ICNAAM 2005*, 2005.

[4] Brodal, G., Lyngso, R., Pedersen, C. S., Stoye, J.: Finding Maximal Pairs with Bounded Gap, *10th Symposium on Combinatorial Pattern Matching (CPM99)*, 1999.

[5] Celera: The Genome Sequence of Drosophila melanogaster, *Science*, **287**, 2000, 2185–2195.

[6] Celera: The Sequence of the Human Genome, *Science*, **291**, 2001, 1304–1351.

[7] Christodoulakis, M., Iliopoulos, C., Mouchard, L., Tsichlas, K.: Pattern Matching on Weighted Sequences, *Algorithms and Computational Methods for Biochemical and Evolutionary Networks*, 2004.

[8] Crochemore, M.: An Optimal Algorithm for Computing the Repetitions in a Word, *Information Processing Letters*, **12**, 1981, 244–250.

[9] Grillo, G., Licciuli, F., Liuni, S., Sbisa, E., Pesole, G.: PatSearch: a program for the detection of patterns and structural motifs in nucleotide sequences, *Nucleic Acids Res.*, **31**, 2003, 3608–3612.

[10] Gupta, M., Liu, J.: Discovery of Conserved Sequence Patterns using a Stochastic Dictionary Model, *J. Am. Stat. Assoc.*, **98**, 2003, 55–66.

[11] Gusfield, D.: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[12] Hughes, J., Estep, P., Tavazoie, S., Church, G.: Computational Identification of cis-regulatory Elements Associated with Groups of Functionally Related Genes in sacharomyces cerevisae, *J. Mol. Biol.*, **296**, 2000, 1205–1214.

[13] Iliopoulos, C., Makris, C., Panagis, I., Perdikuri, K., Theodoridis, E., Tsakalidis, A.: Computing the Repetitions in a Weighted Sequence using Weighted Suffix Trees, *European Conference on Computational Biology (ECCB 2003), Posters' Track*, 2003.

[14] Iliopoulos, C., Mouchard, L., Perdikuri, K., Tsakalidis, A.: Computing the repetitions in a weighted sequence, *Prague Stringology Conference (PSC 2003)*, 2003.

[15] Iliopoulos, C. S., Makris, C., Panagis, Y., Perdikuri, K., Theodoridis, E., Tsakalidis, A.: Efficient Algorithms For Handling Molecular Weighted Sequences, *3rd IFIP Conference on Theoretical Computer Science - IFIP-TCS04* (J.-J. Lévy, E. W. Mayr, J. C. Mitchell, Eds.), Kluwer, 2004.

[16] Iliopoulos, C. S., Perdikuri, K., Theodoridis, E., Tsakalidis, A. K., Tsichlas, K.: Motif Extraction from Weighted Sequences., *String Processing and Information Retrieval, 11th International Conference (SPIRE)*, 2004.

[17] Jensen, L., Knudsen, S.: Automatic Discovery of Regulatory Patterns in Promoter Regions Based on Whole Cell Expression Data and Functional Annotation, *Bioinformatics*, **16**, 2000, 326–333.

[18] Keles, S., van der Laan, M., Dudoit, S., Xing, B., Eisen, M.: Supervised Detection of Regulatory Motifs in DNA Sequences, *Statistical Applications in Genetics and Molecular Biology*, **2**, 2003.

[19] Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time, *FOCS99*, 1999.

[20] Kurtz, S., Schleiermacher, C.: REPuter: fast computation of maximal repeats in complete genomes, *Bioinformatics*, **15**, 1999, 426–427.

[21] Lawrence, C., Altschul, S., Boguski, M., Liu, J., Neuwald, A., Wootton, J.: Detecting Subtle Sequence Signals: A Gibbs Sampling Strategy for Multiple Alignment, *Science*, **262**, 1993, 208–214.

[22] Li, H., Rhodius, V., Gross, C., Siggia, E.: Identification of the binding sites of regulatory proteins in bacterial genomes, *Genetics*, **99**, 2002, 11772–11777.

[23] Liu, Y., Wei, L., Batzoglou, S., Brutlag, D., Liu, J., Liu, X.: A Suite of Web-Based Programs to Search for Transcriptional Regulatory Motifs, *Nucleic Acids Research*, **32**, 2004.

[24] Marsan, L., Sagot, M.: Extracting Structured Motifs Using a Suffix Tree- Algorithms and Application to Promoter Concensus Identification, *RECOMB*, 2000.

[25] McCreight, E.: A space-economical suffix tree construction, *Journal of the ACM*, **23**, 1976, 262–272.

[26] McGuire, A., Hughes, J., Church, G.: Conservation of DNA Regulatory Motifs and Discovery of New Motifs in Microbial Genomes, *Nucleic Acids Research*, **10**, 2000, 744–757.

[27] Ron, D., Singer, Y., Tishby, N.: The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length, *Machine Learning*, **25**, 1996, 117–149.

[28] Roth, D., Hughes, J., Esterp, P., Church, G.: Finding DNA Regulatory Motifs within Unaligned Noncoding Sequence Clustered by whole Genome MRNA Quantitation, *Nature Biotechnology*, **16**, 1998, 939–945.

[29] Stoye, J., Gusfield, D.: Simple and flexible detection of contiguous repeats using a suffix tree, *9th Symposium on Combinatorial Pattern Matching*, 1998.

[30] Sun, Z., Yang, J., Deogun, J.: MISAE: A New Approach for Regulatory Motif Extraction, *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference (CSB 2004)*, 2004.

[31] Tadesse, M., Vannucci, M., Lio, P.: Identification of DNA Regulatory Motifs Using Bayesian Variable Selection Suite, *Bioinformatics*, **20**, 2004, 2553–2561.

[32] Thompson, W., Lawrence, C.: Gibbs Recursive Sampler: Finding Transcription Factor Binding Sites, *Nucleic Acids Research*, **31**, 2003, 3580–3585.

[33] Tompa, M., Sinha, S.: A Statistical Method for Finding Transcription Factor Binding Sites, *Proceedings Int. Conf. Intell. Syst. Mol. Biol.*, 2000.

[34] Tsunoda, T., Fukagawa, M., Takagi, T.: Time and memory efficient algorithm for extracting palindromic and repetitive subsequences, *Pacific Symposium on Biocomputing*, 1999.

[35] Ukkonen, E.: On-line construction of suffix trees, *Algorithmica*, **14**, 1995, 249–260.

[36] Wang, H., Perng, C., Fan, W., Park, S., Yu, P.: Indexing Weighted-Sequences in Large Databases., *International Conference on Data Engineering (ICDE)*, 2003.