EFFICIENT ALGORITHMS FOR HANDLING MOLECULAR WEIGHTED SEQUENCES

Costas S. Iliopoulos,[1] Christos Makris,[2,3] Yannis Panagis,[2,3] Katerina Perdikuri,[2,3] Evangelos Theodoridis,[2,3] and Athanasios Tsakalidis,[2,3]

[1]*Department of Computer Science*
King's College London, Strand, London WC2R2LS
England
csi@dcs.kcl.ac.uk

[2]*Department of Computer Engineering and Informatics*
University of Patras, 26500 Patras
Greece
{makri, panagis, perdikur, theodori}@ceid.upatras.gr

[3]*Research Academic Computer Technology Institute*
61 Riga Feraiou Str., 26221 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.

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 manual 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 [4],[5]. 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 characters of the alphabet. 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 [7], [13]. The weights assigned to each character can be thought of as modeling 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 [10] 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 space needed. 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 [9]. 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.

The structure of the paper is 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

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 three weighted positions. Positions consisting of a single character indicate that this character appears with probability 1.

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 Σ 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 Σ is denoted by Σ^+. 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 ε; 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.

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 word w, as following:

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

For example, if we consider the DNA alphabet Σ = {A,C,G,T} the word w 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 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 subwords.

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

The Suffix Tree

The suffix tree is a fundamental data structure supporting a wide variety of efficient string searching 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 [8].

DEFINITION 2 *We denote by $T(S)$ the suffix tree of S, as the compressed trie of all the suffixes of $S\$,\ \$ \notin \Sigma$. Let $L(v)$ denote the path-label of node v in $T(S)$, which results by concatenating the edge labels along the path from the root to v. Leaf v of $T(S)$ is labeled with index i iff $L(v) = S[i..n]$. 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 [14], [17].

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 generalisation 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 straightforward to give a formal definition as with its ordinary counterpart. A quite informal definition appears below.

DEFINITION 3 *Let S 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 $1/k$. Denote each of them as $S_{i,j}$, where j is the subword rank in arbitrary numbering. We define $WST(S)$ the weighted suffix tree of a weighted sequence S, as the compressed trie of a portion of all the weighted subwords starting within each suffix S_i of $S\$,\ \$ \notin \Sigma$, having a probability of appearance greater than 1/k. Let $L(v)$ denote the path-label of node v in $WST(S)$, which results by concatenating the edge labels along the path from the root to v. Leaf v of $WST(S)$ is labeled with index i if $\exists j > 0$ such that $L(v) = S_{i,j}[i..n]$ and $\pi(S_{i,j}[i \cdots n]) \geq 1/k$, 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 above sequence and $k \geq 1/4$ we have the following possible prefixes for every suffix:

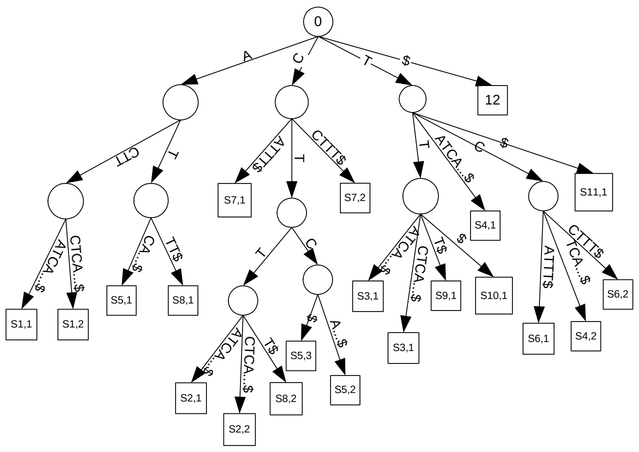

Figure 2. A Weighted Suffix Tree example.

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

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

Construction of the WST

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

Step 1: For each i, ($2 \leq i \leq n$), generate all possible weighted suffixes of the weighted sequence with probability of appearance greater than $1/k$.

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

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 approcah. The exact steps of our methodology for construction are:

Step 1: 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 $1 - 1/k$,

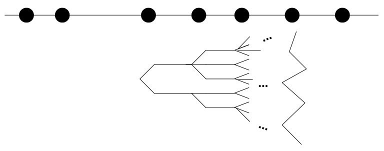

Figure 3. Producing all possible subwords from left to right

- mark position i *gray*, if *at least 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.

Notice that the following holds: at white positions we have only one possible character appearing, thus we can call them *solid* positions, 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. At gray positions, only one character eventually survives, since all the possible characters except 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 also maintain a list B of all black positions.

Step 2: Scan all the positions in B 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 π', π''. The first one measures the actual subword probabilities and the second one is defined by temporarily treating gray positions as white. The generation of a subword stops when it meets a black position and π'' (which skips gray positions) has reached the $1/k$ threshold. We call this position *extended position*. Notice that the actual subword may actually be shorter as π' (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. 4. Notice that only the actual subwords need to be represented with the GST.

Step 3: Having produced all the subwords from every black position, we insert the actual subwords in the generalised suffix tree in the following way. 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 $X' = x[i..\ i + f' - 1]$ is the extended subword of the actual subword $X = x[i..\ i + f - 1]$ ($f \leq f'$) that begins at black position i of the weighted sequence in Fig. 4. Observe the following two facts:

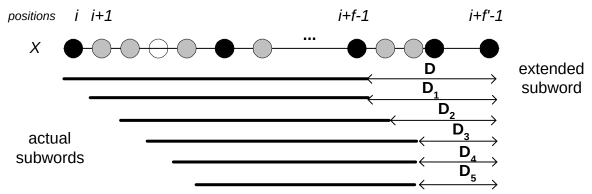

Figure 4. Insertion of subwords in the GST

- There is no need to insert every suffix of X 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 step 2 is executed for i'.
- A suffix of X' can possibly extend to the right of position $i+f-1$, where the actual subword ends, since π' does not take gray positions into account (cf. Fig. 4). No suffix can end though at a position greater than $i+f'-1$, where the extended subword ends.

We have kept every leaf storing a suffix of X', in a list L. Let D_j denote the redundant portion of suffix $X'[i+j..i+f'-1]$ of X' (cf. Fig. 4). After we have inserted the extended subword and the proper suffixes using McCreight's algorithm [14], we have to remove all the D_j's from the GST. Starting from the leaf corresponding to the entire X', we move upwards the tree by D characters. At the current position we eliminate the extra portion of X', storing X. The next redundancy of length D_1 is at the end of $X'[i+1..i+f'-1]$. We locate this suffix using the suffix link. Let $\lambda_d = |D_{d-1}| - |D_d|,\ d > 1$ and $\lambda_1 = D - D_1$. After using the suffix link we also may descend by λ_1 characters. At this position we store the correct suffix (possibly extending it up to λ_1 characters after position $i+f-1$). We continue the elimination procedure for the remaining suffixes of X', as outlined above. The entire process costs at most $\sum_{d>0} \lambda_d = O(D)$, which is the time required to complete the suffix tree construction.

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 process these subsequences separately, according to the previous algorithm.

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 4 *At most* $O\left(|\Sigma|^{\log k/\log(\frac{k}{k-1})}\right)$ *subwords could start at each branching position* i *(*$1 \le i \le n$*) 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 λ 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

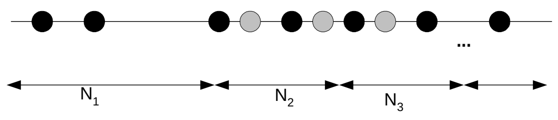

Figure 5. Time cost for step 2

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 $\hat{\pi} = 1 - 1/k$. Assuming that there are no gray positions that could reduce the cumulative probability, $\pi(u[1..\lambda])$ is less or equal to $\hat{\pi}^l$ (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 $\log_2$). For example, typical values of l are $\cong 21.9$ for $k = 20$ and $\cong 1046$ for $k = 200$.

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. ∎

LEMMA 5 *The number of subwords with probability greater than or equal to $1/k$ is at most $O(n)$.*

Proof. If every position i of the weighted sequence is the starting point of a constant number of subwords (Lemma 4), the total number of subwords is $O(n)$. ∎

LEMMA 6 *Step 2 of the construction algorithm takes $O(n)$ time.*

Proof. Suppose that the weighted sequence is divided into windows $N_j, j \geq 1$ (cf. Fig. 5). 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. Step 2 scans the black positions inside N_i. Every black position will generate $O(1)$ subwords (according to Lemma 4) 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, step 2 costs at most $O(l^2(|N_i| + |N_{i+1}|)) = O(|N_i| + |N_{i+1}|)$ time. Summing up the costs for all windows we conclude that step 2 incurs a total of $O\left(\sum (|N_i| + |N_{i+1}|)\right) = O(n)$ cost. ∎

LEMMA 7 *Step 3 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 step 3 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 is $O(l \cdot |N_i| + |N_{i+1}|) = O(|N_i| + |N_{i+1}|)$ and the cost to repair the WST (as we described in step 3) is $O(l \cdot D)$. D is always smaller than $|N_i| + |N_{i+1}|$ thus for window N_i step 3 costs $O(|N_i| + |N_{i+1}|)$ time. Summing the costs for all windows, step 3 yields $O(n)$ time in total. ∎

Based on the previous lemmas we derive the following theorem.

THEOREM 8 *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 5) 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 $O(n)$. ∎

4 Applications

In this section we present three 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.

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 further and thus we report failure. If p contains weighted positions we decompose it into solid patterns each with $\Pr\{\text{occurence}\} > 1/k$ 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 α the output size, with $O(n)$ preprocessing.

Computing the Repeats

A lot of work has been done for identifying the repeats in a word. In [6], [2] and [15], authors have presented efficient methods that find occurrences of squares in a string of length n in time $O(n \log n)$ plus the time to report the detected squares. Moreover in [11] 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. [12] and [16].

Using the WST we can compute in linear time the repeats of a weighted sequence. In particular, we compute the repeats of all subwords u, with $\Pr\{u\} > 1/k, \forall u$. 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 in $LL(v)$, in pairs. This process requires $O(n)$ time by Lemma 5

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

REMARK 1 *Apart from the repeats problem the repetitions detection in weighted molecular sequences can be solved in $O(n \log n + \alpha)$ time, by extending appropriately either of the approaches in [15], [3].*

Longest Common Substring in Weighted Sequences

A classical problem in string analysis is to find the longest common substring of two given strings S_1 and S_2. Here we reformulate the *longest common substring problem* for weighted sequences.

Problem 3. *Given two weighted strings S_1 and S_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 S_1 and S_2 is to build a generalised weighted suffix tree for S_1 and S_2. The path label of any internal node is a substring common to both S_1 and S_2 with probability of appearance greater than $1/k$. The algorithm merely finds the node with greatest string-depth. A preorder traversal of the WST suffices to compute the longest string-depth (for details see [8]). It is easily derived that the above procedure runs in $O(n)$ time.

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*.

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

Using the WST we can compute in $O(n \log n \log n)$ 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. We build WST(S) with parametre k for the sequence S in which every subword appears with probability above $1/k$. We merely have to examine the path to $S[1..n]$. Let s_1 be the leaf storing $S[1..n]$. Let also v be an internal node of WST(S). At each such node v, let s_v denote the string spelled in the path from the root to v.

First we need to perform a depth-first search to construct at each internal node, the leaf-lists of its subtree, as those correspond to occurrences of s inside the indexed string. We organize these lists as simple linked lists, namely $LL(v)$ at a node v. We also need to maintain a gap-tree G, implemented as a van Emde Boas tree [18] over the universe $U = [1, 2, ..., n]$. The gap-tree keeps track of the indices of leaf-lists on the way from s_1 to the root and performs predecessor-successor queries. The algorithm entails moving upwards from s_1 to the root and keeping at each node encountered, the maximum distance, d_{max} of consecutive indices stored at G as well as the maximum index value i_{max}.

In order for an internal path label at node s_v on the path from root to s_1 to form a cover, it must hold that $d_{max} \leq |s_v|$ and $n - i_{max} \leq |s_v|$. More informally, $LL(v)$ of v stores several indices in $[1..n]$. These correspond to repetitive occurrences of s_v in S. Consequently, s_v is a cover whenever the maximum difference between any two starting points of these occurrences is less than the $|s_v|$.

More concretely, we start at s_1 and construct an empty van Emde Boas tree G, insert value 1 and set $d_{max} = n$ and $i_{max} = 1$. At each internal node we must check whether it forms a cover according to the conditions stated above. As moving from a node v to $father(v)$, we insert all the items of the $LL(w)$, $\forall w = \text{sibling}(v)$ to the tree G. After having inserted item i we perform an operation $succ(i)$ and $pred(i)$ in the tree and set $d_{max} = \min\{i - pred(i), succ(i) - i, d_{max}\}$. We also need to check whether $i > i_{max}$ and update d_{max} accordingly.

THEOREM 9 *Computing all covers requires $O(n \log n \log n)$ time.*

Proof. The initialisation procedure takes $O(n)$, to construct the LL lists at the children of each node in path from root to s_1. At each transition from a node v to $father(v)$ a number of insertions need to be made to the gap-tree G. The number of insertions equals the cardinality of each $LL(w)$, $\forall w = \text{sibling}(v)$. Each of these insertions costs $O(\log \log n)$ (see [18]). The item of each LL is only once inserted in G and subsequently left intact. Furthermore, each of the n positions in S occurs only once within a leaf-list, thus inserted only once in G. Each such insertion causes a predecessor and a successor operation in G which are also performed in $O(\log \log n)$ time. Hence, our algorithm incurs a total of $O(n)$ cost for constructing leaf-lists, another $O(n)$ maybe spent during the bottom up traversal of the s_1-to-root path and a total $O(n \log \log n)$ time for performing operations on G, yielding the overall time complexity. ∎

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, repeats finding, least 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 experimental.

Moreover 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.

References

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

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

[3] Brodal G.S., Lyngso R.B., Storm Pedersen C.N., and Stoye J.: Finding Maximal Pairs with Bounded Gap. In Proc. 10th CPM, pp. 134–149, (1999).

[4] Celera Genomics: The Genome Sequence of Drosophila melanogaster. Science, Vol. 287. (2000) 2185–2195

[5] Celera Genomics: The Sequence of the Human Genome. Science, Vol. 291, (2001) 1304–1351.

[6] Crochemore, M.: An Optimal Algorithm for Computing the Repetitions in a Word. Inf. Proc. Lett., Vol. 12. (1981) 244–250.

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

[8] Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York (1997)

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

[10] Iliopoulos, C., Mouchard, L., Perdikuri, K., Tsakalidis, A.,: Computing the repetitions in a weighted sequence, Proceedings of the Prague Stringology Conference (PSC 2003), 91-98.

[11] Kolpakov, R., Kucherov, G.,: Finding maximal repetitions in a word in linear time. In Proc. FOCS99, pp. 596–604, (1999).

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

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

[14] McCreight, E.,M.,: A space-economical suffix tree construction algorithm. J. of the ACM, Vol. 23, (1976) 262–272.

[15] Stoye, J., Gusfield, D.,: Simple and flexible detection of contiguous repeats using a suffix tree. In Proc. 9th CPM, Vol. 1448 of LNCS, (1998) 140–152.

[16] Tsunoda, T., Fukagawa, M., Takagi, T.,: Time and memory efficient algorithm for extracting palindromic and repetitive subsequences in nucleic acid sequences. Pacific Symposium on Biocomputing, Vol. 4, (1999) 202–213.

[17] Ukkonen, E.,: On-line construction of suffix trees. Algorithmica, Vol. 14, (1995), 249–260.

[18] van Emde Boas P., R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, **10**, pp. 99-127, (1977)