New Upper Bounds on Various String Manipulation Problems

Christos Makris[1,3], Yannis Panagis[1,2], Katerina Perdikuri[1,2], Spyros Sioutas[1,2], Evangelos Theodoridis[1,2], Athanasios Tsakalidis[1,2], and Kostas Tsichlas[1,2]

[1] Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece
{makri, perdikur, panagis, sioutas, theodori, tsihlas}@ceid.upatras.gr
[2] Research Academic Computer Technology Institute,
61 Riga Feraiou Str., 26221 Patras, Greece
tsak@cti.gr
[3] Department of Applied Informatics in Management & Finance, Technological Educational Institute of Mesolonghi
Mesolonghi, Greece
makri@ceid.upatras.gr

Abstract. In this chapter we deal with various string manipulation problems which originate from the field of computational biology and musicology. These problems are: "approximate string matching with gaps", "inference of maximal pairs in a set of strings" and "handling of weighted sequences". We provide new upper bounds for solving these problems and for the third we propose a novel data structure, for the representation of the weighted sequences, which inherits most of the properties of the suffix tree.

1 Introduction

String Manipulation Techniques arise in a variety of practical applications such as: word processors, information retrieval systems, molecular sequence databases and music analysis programs. In this chapter we focus on the application of well known Data Structures in solving efficiently string manipulation problems.

In the following paragraphs we will introduce the Approximate Matching Problem with Gaps, the Model Inference Problem in Multiple Strings and the Weighted Suffix Tree, an efficient data structure for solving string manipulation problems in weighted sequences.

The practical importance of these data structuring applications appears in the fields of computerized music analysis and computational biology. The algorithms to be presented can be easily used in the analysis of musical works in order to discover similarities between different musical entities that may lead to establishing a "characteristic signature" [3, 4]. This can be accomplished by noticing that a musical score can be represented as a string and by defining the alphabet to be the set of notes in the chromatic or diatonic notation or the set of intervals that appear between notes.

On the other hand two of the most important goals in computational molecular biology include finding regularities in nucleic or protein sequences, and finding features that are common to a set of such sequences. Both imply inferring patterns, unknown at first, from one or more strings.

In Computational Biology, DNA and Protein sequences can be seen as long texts over specific alphabets. When dealing with DNA sequences the alphabet consists of the four nucleotides, while in the case of protein sequences, the alphabet consists of the twenty amino acids. Those sequences represent the genetic code of living beings. Searching specific sequences over those texts appears as a fundamental operation for problems such as assembling the DNA chain from the pieces obtained by experiments, looking for given DNA chains or determining how different two genetic sequences are.

Regularities in molecular sequences may come under many guises. They may correspond to approximate repetitions randomly dispersed along the sequence, or to repetitions that occur in a periodic or approximately periodic fashion, or else to tandem arrays. The length and number of repeated elements one wishes to be able to identify may be highly variable. Patterns common to a set of sequences may likewise present diverse forms. For various problems in molecular biology, in particular the study of gene expression and regulation, it is important to be able to infer what has been called "structured patterns". Structured patterns allow to identify conserved elements recognized by different parts of the same protein or macromolecular complex, or by various complexes that then interact with each other.

In molecular biology binding site of a regulatory protein can be modeled as a weighted sequence. Each base in a candidate motif instance makes some positive, negative or neutral contribution to the binding stability of the DNA-protein complex. 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. In this chapter we present the weighted suffix tree, a data structure for handling weighted sequences.

In the next paragraph we give the basic definitions to be used in the following sections.

1.1 Basic Definitions

A *string* is a sequence of zero or more symbols drawn from an alphabet Σ. The set of all strings over the alphabet Σ is denoted by Σ^+. A string x of length n is represented by $x_{1..n} = x_1 x_2 \cdots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$, and $n = |x|$ is the length of x. The empty string is the empty sequence (of zero length) and is denoted by ε; we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k and is called *the k^{th} power of* x.

A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$, a *proper prefix* if $u \in \Sigma^+$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$. A string u that is both a proper prefix and a suffix of x is called *a border* of x.

If x has a nonempty border, it is called *periodic*. Otherwise, x is is said to be *primitive*.

Definition 1. *Given a string p called the pattern and a longer string t, called the text, the exact pattern matching problem is to find all occurrences, if any, of pattern p inside the text t.*

According to the above definition, in the problem of exact pattern matching, one is interested in finding all occurrences of a given pattern ("structured" or "non-structured") in a given input sequence. A "non-structured" pattern p is a string of length m, while a "structured" pattern can be defined as an ordered collection of k "boxes" B_i and $k-1$ intervals of distances, called gaps (each one between each pair of successive boxes - see Fig. 1). Each gap g_i could have a minimum min_i and a maximum max_i value , or a fixed length.

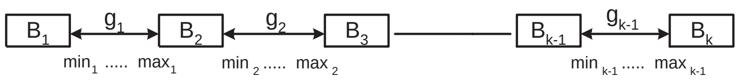

Fig. 1. A structured pattern model.

When we consider the approximate version of this problem we do not require a perfect matching but a matching that is good enough to satisfy certain criteria. The problem of finding substrings in a text similar to a given pattern has been extensively studied in recent years because it has a variety of applications including file comparison, spelling correction, information retrieval, searching for similarities among biosequences and computerized music analysis.

One of the most common variant of the approximate string matching problem is that of finding substrings that match the pattern with at most k-differences. In this case, k defines the approximation extent of the matching (the edit distance with respect to the three edit operations - substitute, insert, delete).

Definition 2. *Given a pattern p, a text t, an integer k and an edit distance function $d()$, the approximate pattern matching problem is to find the set of all text positions j, such that there exists i, with $d(p, t_{i \cdots j}) \leq k$.*

Considering the above problem under a variety of similarity or distance rules (i.e. Hamming distance, etc.) we can define a set of relevant problems. The basic idea in the approximate pattern matching is to locate occurrences of a pattern that can tolerate certain ranges of error. The notion of error is defined either locally or globally as follows.

Definition 3. *Let δ and γ be integers. Two symbols a, b of an alphabet Σ are said to be δ-approximate, denoted as $a =_\delta b$, if and only if $|a - b| \leq \delta$. In that manner two strings x, y are δ-approximate, denoted as $x =_\delta y$, if and only if $|x| = |y|$ and $\forall i, x_i =_\delta y_i$. Also two strings x, y are γ-approximate, denoted as $x =_\gamma y$, if and only if $|x| = |y|$ and $\sum_{i=1}^{|x|} |x_i - y_i| \leq \gamma$. Finally we say that two strings are (δ, γ)-approximate if both conditions are satisfied.*

The error in the first case (δ-approximation) is defined locally for each symbol. In the second case (γ-approximation) the error is defined in a more global sense allowing the uneven distribution of the error to the symbols. Efficient algorithms for approximately matching patterns to text strings are given in [3, 4, 7].

Another set of problems arises when we allow gaps in the approximate matching problem. The problem of pattern matching with gaps was introduced in [5] and is defined as follows.

Definition 4. *Given a pattern p, and a text t, find all occurrences of p in t such that $p_i = t_{j_i}, \forall i \in \{1, \cdots, m\}$, where m is the length of p. Note that p occurs at position j_1 of t with a gap sequence $G = (g_1, g_2, \cdots, g_{m-1})$, with $g_i = j_{i+1} - j_i, \forall i \in \{1, \cdots, m-1\}$ and $j_1 < j_2 < \cdots < j_m$.*

The different versions of the problem of matching with gaps result from the different constraints posed on the structure of the gaps. In all the above cases the pattern p is given. In the case where the pattern p is not given we consider the Model Identification Problem.

Definition 5. *Given a set of strings $S = \{S_1, S_2, \cdots, S_k\}$ we seek a "structured pattern" P that occurs in every $S_i, \forall i \in \{1, \cdots, k\}$. In the special case where the pattern P consists of two identical boxes B (with various restrictions on gaps) we address the Maximal Pairs Identification Problem.*

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 [9].

Definition 6. *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 [13], [14].

In the case of weighted sequences we consider the presence of a set of characters each with a given probability of appearance for a given position of a word w. Thus we define the concept of a weighted word w, as following:

Definition 7. *A weighted word $w = w_1, \ldots, w_n$ 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 $\Sigma = \{$A,C,G,T$\}$ the word w shown in Fig. 2 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

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)			

Fig. 2. Example of a weighted word with three weighted positions. Positions consisting of a single character indicate that this character appears with probability 1.

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 respective 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 substring can be easily extended to accommodate weighted substrings.

2 Approximate Matching with Gaps

In this section we present algorithms for various versions of the problem of approximate matching with gaps. These different versions of the problem, which are extracted by the different constraints posed on the structure of the gaps, were introduced in [5, 6] and are the following: (i) δ-occurrence and (δ, γ)-occurrence with α-bounded gaps, (ii) δ-occurrence minimizing total difference of gaps, (iii) δ-occurrence and (δ, γ)-occurrence with ϵ-bounded-difference gaps and (iv) δ-occurrence of a set of strings with bounded gaps. In the relevant literature ([5, 6]) algorithms for the following problems are also considered: (i) δ-occurrence and (δ, γ)-occurrence with strictly bounded gaps, (ii) δ-occurrence and (δ, γ)-occurrence with unbounded gaps and (iii) δ-occurrence minimizing total sum of gaps. We will not discuss these techniques here.

2.1 δ-Occurrence and (δ, γ)-Occurrence with α-Bounded Gaps

The δ-Occurrence with α-Bounded Gaps problem is defined as follows.

Definition 8. *Given a text string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers α, δ, check whether there is a δ-occurrence of p in t with gaps whose sizes are bounded by constant α.*

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

The problem is solved by employing an incremental procedure based on dynamic programming. Firstly, we define the set of all non-empty prefixes of pattern p to be Prefixes(p), that is Prefixes(p) $= \{\pi_1, \pi_2, \ldots, \pi_m\}$, where $\pi_i = p_{1,\ldots,i}$. The proposed algorithm computes the entries of a matrix $D_{0\ldots m, 0\ldots n}$. The entry $D_{i,j}$ of the matrix designates the position of the last δ-occurrence with α-bounded gaps of prefix π_i before position j in text t, otherwise it has the value 0. The entries of the matrix are computed as follows:

$$D_{i,j} = \begin{cases} j & \text{if } (t_j =_\delta p_i) \text{ and } (j - D_{i-1,j-1} \leq \alpha + 1) \text{ and } (D_{i-1,j-1} > 0) \\ D_{i,j-1} & \text{if } (t_j \neq_\delta p_i) \text{ and } (j - D_{i,j-1} < \alpha + 1) \\ 0 & \text{otherwise} \end{cases}$$

If $D_{i,j} = j$, then there is a match between t_j and p_i while the prefix π_{i-1} has a δ-occurrence at a position given by $D_{i-1,j-1}$ and the formed gap is $\leq \alpha$. If $D_{i,j} = D_{i,j-1}$ then there is no match between t_j and p_i. This means that it is not possible to extend the δ-occurrence of prefix π_{i-1} to π_i and so we store in $D_{i,j}$ the previous value of $D_{i,j-1}$ as long as the gap invariant is not violated. In any other case we store in $D_{i,j}$ the value 0. The boundary conditions of matrix D are as follows:

$$D_{0,0} = 1, D_{i,0} = 0 \text{ and } D_{0,j} = j$$

The above algorithm runs in $O(mn)$ time and uses $O(mn)$ space. In practice the space is linear $O(n)$, since the computation of each row depends only on the previous row. If we want to retrieve a match, we can use the matrix D and perform a trace-back procedure. The time complexity of this procedure is $O(m)$ while the space complexity remains $O(mn)$. Another option would be to use *Hirschberg's divide and conquer technique* [10]. In this case the space complexity is reduced to $O(m + n)$.

In the sequel we are going to explore the problem of computing (δ, γ)-occurrences of a pattern with α-bounded gaps. This problem is defined as follows:

Definition 9. *Given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers α, δ, γ, check if there is a (δ, γ)-occurrence of p with α-bounded gaps in t.*

We will follow an approach similar to the previously used. The computation of D is performed exactly in the same way, except that as we scan each symbol of the text (that is, as we complete each column of matrix D) we maintain for each p_i a *min-queue* ([8]) storing the occurrences of π_{i-1}, such that the gap invariant is not violated. All occurrences stored in this queue satisfy the gap invariant with respect to the current position, while the order key is the approximation error. When we find a δ-occurrence of p_i that extends an occurrence of π_{i-1} to an occurrence of a π_i we add it to the queue of symbol p_{i+1}. When we scan a text symbol we may also delete the first element inserted in the queue since the gap invariant may be violated.

When we encounter a δ-occurrence of p_i we form π_i by letting p_{i-1} be the minimum error δ-occurrence of p_{i-1} among the δ-occurrences stored in the list corresponding to p_i. We also keep the costs of occurrences in an auxiliary matrix C. This matrix is constructed simultaneously with matrix D. When $D(i, j) = 0$ then the corresponding $C(i, j)$ contains the cost of the occurrence of π_i at position j of the text. The cost is the sum of all δ-errors introduced in each symbol of the occurrence of prefixes. When $D(i, j) \neq 0$ then $C(i, j)$ yields the total error of this occurrence. In this way, if there is an occurrence at matrix D (row m of matrix D) then by using matrix C we can deduce whether this is a γ-occurrence as well or not. The time complexity of the algorithm is $O(nm)$ while the space complexity becomes $O(nm + m\alpha) = O(nm)$.

2.2 δ-Occurrence Minimizing Total Difference of Gaps

This problem is formally stated as follows:

Definition 10. *Given a text string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and an integer δ, check if there is a δ-occurrence of p with gaps minimizing the quantity $\sum_{i=1}^{m-2} G_i$, where $G_i = |g_i - g_{i+1}|$.*

To solve this problem we construct a directed acyclic graph (DAG) $H = (V, E)$ by creating for each symbol $p_i, 1 \leq i \leq m$ a node $v_i^j, 1 \leq j \leq n$ whenever $p_i =_\delta t_j$. In this way, for each symbol p_i of pattern p we may create as many as n nodes v_i^j producing totally, at most nm nodes. The construction is made so that the nodes are divided in layers of nodes, where each layer corresponds to the δ-occurrences of a specific symbol p_i of pattern p.

The set of edges E will be constructed as follows. Edges among nodes of the same layer are forbidden. This because we would like the edges to represent all the different δ-occurrences of a pattern p in text t. We introduce a new directed edge between two nodes v_i^j and $v_{i'}^{j'}$ if and only if $i' = i + 1$ and $j' > j$. All nodes that correspond to a δ-occurrence of the symbol p_1, that is nodes v_1^j that lie at the first layer, are connected to a node s ($s \to v_1^j$). All nodes that correspond to a δ-occurrence of the symbol p_m are connected to a node d ($v_m^j \to d$). To each edge we assign a cost proportional to the length of the gap defined by this occurrence. In this way, the edge $e = (v_i^j, v_{i+1}^{j'})$ is given weight $w_e = j' - j$. Edges starting from s or ending at d are given zero weights. It is obvious that in the worst case the edge set will have $O(n^2m)$ size. Concluding, we can compute the set V in $O(nm)$ time while the set E is computed in $O(n^2m)$ time. Thus, the time complexity as well as the space complexity is asymptotically equal to $O(n^2m)$.

From graph H we construct a new graph H'. H' is implemented by contracting two nodes v_i^j and $v_{i+1}^{j'}$ that are connected by the directed edge $e = (v_i^j, v_{i+1}^{j'})$ into a single node v'_e. If the edge $f = (v_{i+1}^{j'}, v_{i+2}^{j''})$ does also exist in H (in H' it is represented by the node v'_f) then in H' we introduce the edge $e' = (v'_e, v'_f)$. The weight of the edge e' in H' is defined as the difference between gap lengths corresponding to weights at edges e and f in H. The new graph H' may have as many as $O(n^2m)$ nodes and $O(n^2m)$ edges. Because of the fact that H' is a DAG (Directed Acyclic Graph) we can compute a shortest path in $O(n^2m)$ time by topologically sorting it. The computation of the shortest path in this graph provides us with the solution to this problem using $O(n^2m)$ space in $O(n^2m)$ time.

The space complexity can be reduced from $O(n^2m)$ to $O(n^2)$ since it is possible to simulate the shortest path computation during the construction of H'. Note that during the construction of H' we need the nodes and edges that correspond to three consecutive symbols p_{i-1}, p_i and p_{i+1} of the pattern p when the current symbol scanned is p_{i+1}. By scanning through the pattern and constructing the required parts of H', one can compute the shortest path to each new node by taking the minimum over the incoming edges.

A similar procedure to the one described above can be used to solve the problem of δ-occurrences and (δ, γ)-occurrences with ϵ-Bounded-Difference Gaps. The time and space complexities remain $O(n^2m)$ and $O(n^2)$ respectively (for more details the reader can refer to [6]).

2.3 δ-Occurrence of a Set of Strings with Bounded Gaps

Let $S_1, S_2, \ldots, S_k \in \Sigma^*$, be a set of k strings. The problem is formed as follows.

Definition 11. *Given a set of strings, a text string $t = t_1, \ldots, t_n$, and an integer δ, check if there is a δ-occurrence of each string S_i in text t such* *that if* $t[l_1 \ldots] =_\delta S_1, t[l_2 \ldots] =_\delta S_2, \ldots, t[l_k \ldots] =_\delta S_k$ *then* $g_1 = l_2 - l_1 \leq \Delta$ *and generally* $g_i = l_{i+1} - l_i \leq \Delta$.

The solution is quite similar to that of subsection 2.2. Define the pattern p to be $p = S_1 S_2 \ldots S_k$. Then, construct matrix D by finding in each row i the δ-occurrence of the substring S_i exactly as we did when we tried to find the δ-occurrence of a single symbol p_i. This can be easily accomplished in time linear to the length of the pattern S_i, by a series of character by character comparisons. Thus, the whole time complexity is equal to $O(n(|S_1| + |S_2| + \cdots + |S_k|))$ and the space consuption is $O(n(|S_1| + |S_2| + \cdots + |S_k)|))$.

3 Model Inference in Multiple Strings

A very interesting problem that is derived from computational molecular biology is the “Model Inference Problem”. In this problem we seek to identify all the regularities (repeated substrings or structures), not known a priori, in a nucleic or protein sequence (Fig. 3).

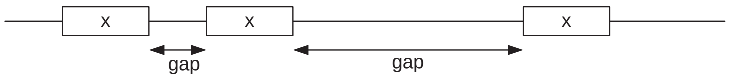

Fig. 3. Inference of Regularities

Towards this goal, the first step that has to be made, is to infer the occurrences of pairs of equal substrings. The space (number of characters) between the end of the first of the equal substrings and the start of the second one is called gap. Gusfield (in [9]) demonstrates a basic technique of finding all maximal pairs in a string of length n, without any restriction on gaps, in time $O(n + a)$ and space $O(n)$, where a is the number of reported pairs. His method is based upon the suffix tree and one of its basic properties that we are going to discuss below. Brodal et al. in [1] extended the algorithm of Gusfield and proposed two new ones that compute all maximal pairs in an input string of length n whose gaps are restricted. When the gaps are lower and upper bounded the algorithm works in $O(n \log n + a)$ time and when the gaps are only lower bounded $O(n+a)$ time is needed, where a is the size of the output. Both algorithms use linear space.

In this section we are going to consider a more general version of this problem where repetitive structures are sought in several strings together and not only in a particular string (Fig. 4). This kind of information points out common features on a set of sequences, with important biological meaning. We consider the problem of identifying the occurrences of maximal pairs and especially in multiple strings and describe the methodology introduced in [12].

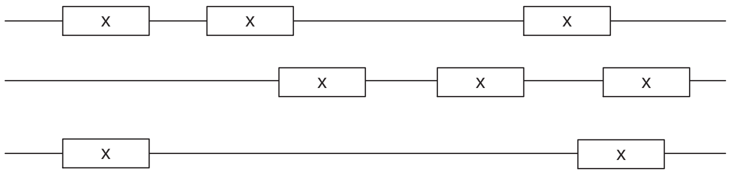

Fig. 4. Regularities in Multiple Strings

3.1 Problem Definitions

Consider that we have a set of strings $S = \{S_1, S_2 \ldots, S_k\}$ where each of these strings is constructed from the alphabet Σ and the total length of the strings is $\sum_{1 \leq i \leq k} |S_i| = n$. Our goal is to find all the maximal pairs that appear simultaneously in all strings S_i $(1 \leq i \leq k)$. Therefore, we will use the suffix tree.

The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels (substrings assigned to edges) on the path from the root to leaf i spells out the suffix of t starting at position i. Two significant details are: (i) for each internal node, each outgoing edge-label has to start by a unique non empty character, (ii) the path-label of a leaf node is the concatenation of the edge-labels on the path from the root to that leaf (see Fig. 5).

Definition 12. *A left-maximal pair in a string t is a pair of identical substrings α and β in t such that the character to the immediate left of α is different from the character to the immediate left of β.*

Definition 13. *A right-maximal pair in defined respectively as the pair that can not be extended to the right.*

Notice that the suffix tree detects easily the right-maximal pairs because at any internal node u the suffixes that lie in different subtrees of u

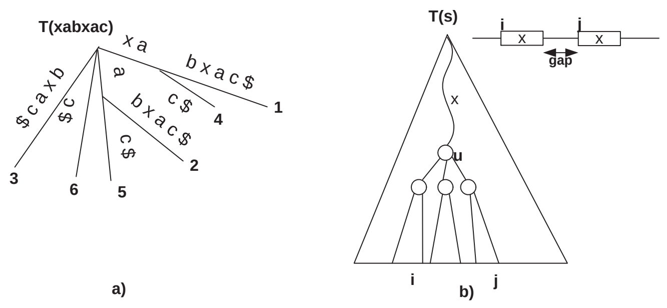

Fig. 5. Suffix tree

start with the same path-label of u and then immediately to the right a different character exists. The later holds because every such suffix corresponds to a path from root to a leaf, touching u, that follows a different edge from u (see (Fig. 5.b).

Definition 14. *A maximal pair in a string t is a pair of identical substrings α and β in t such that the character to the immediate left and right of α are different from the characters to the immediate left and right of β. In other words extending α and β in either direction destroys the equality of the two substrings. A maximal pairs is a left-and-right maximal pair together.*

Remark 1. Having the suffix tree of a string t in our disposal, we can organize in linear time to the length of t the indexes of the suffixes so that the detection of the left-maximal pairs is convenient. At every leaf we keep an array of $|\Sigma|$ lists (one list per each letter of the alphabet - $\{L_1\{\}, L_2\{\}, ..., L_{|\Sigma|}\{\}\}$). These structures are called leaf-lists. Every index i is kept in the list that corresponds to the character at position $i-1$ (immediately to the left). Running a bottom up process in the suffix tree, every internal node u examines all the leaf-lists of its children. The node generates for every possible pair of its children k, l (this guarantees the right-maximality) the maximal pairs, combining all the elements of every sub-list of the first, with the elements of every sub-lists of the other, skipping at the same time sub-lists that correspond to the same character (this guarantees the left-maximality). After reporting the produced maximal pairs, the parent node u merges all the leaf-lists into one, concatenating all the sub-lists that correspond to the same character of the alphabet. Hereafter, the leaf-list is assigned to the parent-node. The merging of leaf-lists at every internal node takes $O(|\Sigma|) = O(1)$ time, therefore the total time needed is $O(n)$ plus the size of the output. Consequently $O(n+a)$ time and linear space $O(n)$ is needed because each of the n indexes occurs only once in a list.

Due to the above remark we can detect the maximal pairs in one string t without caring about the gap between the equal substrings. Trying to find the maximal pairs for either fixed or bounded gaps and at the same time indicating the ones that lie in a set of strings is a bit more complicated.

3.2 Algorithms

Initially we discuss the problem of identification of maximal pairs with arbitrary gaps in a set of strings. We are going to extend Remark 1 to detect when a reported pair from a specific string S_i occurs in all the other strings $S_j(\forall j \neq k)$ as well. Toward this goal we use a generalized suffix tree for all the strings of set S.

A generalized suffix tree *GST(S)* contains all the suffixes of all the strings of S and can be built in linear time to the sum of lengths of the strings. Contrary to a suffix tree, the GST(S) may store more than one indexes at a leaf. Each index belongs to a different string S_i indicating that there are common suffixes among the strings. A similar method to the one described in Remark 1 is used but combining the process of every string together. For this purpose we use new leaf-lists that contains one leaf-list of the Remark 1 for every string S_i (Fig 6).

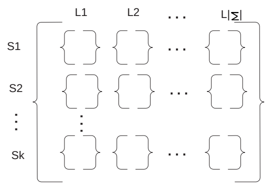

Fig. 6. leaf-list

A bottom up process runs again in the generalized suffix tree, with every internal node u receiving all the leaf-lists of its children. It generates for every possible pair of its children k, l the maximal pairs for each string S_i of the set using the sub leaf-lists that correspond to S_i exactly as described in Remark 1. The produced maximal pairs for every string S_i are not reported immediately because we need to examine if for all the strings $S_i' s$ at least one pair is generated.

In order to achieve this, a temporary array of size k of lists is used to store, in a different list, the produced maximal pairs for every string S_i. In time $O(k)$ can be verified if a list has length equal to zero and consequently if all the strings report at least a pair. Then if such a thing does not hold we report all the lists. After reporting, the concatenating step follows where each of sub leaf-list for each of the strings S_i is concatenated into one sub leaf-list (like Remark 1). The second step takes time $O(|\Sigma|k) = O(1)$. Thus the total space and time needed is linear to the sum of the length of strings ($\sum_{1 \leq i \leq k} |S_i| = n$) plus the size of the output.

Invoking the constraint that the length of the gaps has to be bounded by a constant b, the previous approach should be extended. This constraint further complicates things as can be illustrated in the following example. Consider that the previous bottom up method at an internal node u, the candidate pairs of string S_i are produced taking all the possible combinations of pairs from the lists $S_i \rightarrow L_x\{\}$ and $S_i \rightarrow L_z\{\}, \forall x \neq z$. Now with the new constraint an index i of the first list has not got to be combined with all the indexes of the second one (denoted by j) but only with at most $2b$ indexes that meet the gap constraint($|j - (i + |path\_label|)| \leq b$).

In order to achieve this, the $S_i \rightarrow L_x\{\}$ lists can not be implemented as linear lists, because searching for the proper $j's$ in all the other lists will incur $O(n^2)$ time. If we implement these lists as AVL trees (like in [1]) we trade off merging time (now we do not have to concatenate but merge the lists) for searching time and as we are going to see below, with the following four lemmas, it leads to an $O(n \log n)$ algorithm.

Lemma 1. *Two AVL trees of size at most n and m ,with $n \leq m$, can be merged in $O(\log \binom{n+m}{n})$ time.*

Proof. See [2].

Lemma 2. *Given two sorted list of elements $e_1, e_2, \ldots, e_n$ and $a_1, a_2, \ldots, a_m$ ($n \leq m$) structured in two AVL trees T, T', we can find $q_i = min\{x \in T' | x \geq e_i\}$ for all $1 \leq i \leq n$ in $O(\log \binom{n+m}{n})$ time.*

Proof. The basic idea is to use the merging algorithm of Brown and Tarjan in [2] and instead of performing a real merge, merely find where the element has to be inserted by keeping a pointer.

Lemma 3. *Let T be an arbitrary binary tree T with n leaves. The sum over all internal nodes $u \in T$ of terms $\log \binom{n_1+n_2}{n_1}$, where n_1 and n_2 are the number of leaves in the two subtrees rooted with u (and $n_1 \leq n_2$), is $O(n \log n)$.This lemma is known as the "smaller-half trick".*

Proof. This can be proved by induction to the number of leaves of the binary tree. See [1].

Lemma 4. *The step of producing the candidate pairs and the step of merging at an internal node u with degree d ($d \leq |\Sigma|$) of the generalized suffix tree, can be simulated in a binary tree.*

Proof. As mentioned above for every string S_i and every internal node u the process generates for every possible pair of its children k, l (and this guarantees the right-maximality) the maximal pairs, combining all the elements of every sublist of the first, with the elements of every sub-lists of the other skipping at the same time sub-lists that correspond to the same character (and this guarantees the left-maximality). Assuming that u has degree d, if we transform u into a binary tree with d leaves where every internal edge has empty string label, each of the initial children of u enters the binary tree at a different leaf. If we perform the same process in the binary tree and at every node we just use the existing two leaf-lists from the leaves to the root of the binary tree, we will have considered every possible pair of children k, l of the initial u. For example if u has four children 1,2,3,4 the produced binary tree will have four leaves and height two. Thus, in the bottom up method the pairs of leaf-lists (1,2) and (3,4) are first examined and then merged. In the next level the pair $(1 \cup 2, 3 \cup 4)$ is examined which is equal to pairs (1,3),(1,4), (2,3) and (2,4). So all the possible pairs are formed.

Finally, assuming that we have the generalized suffix tree (GST) T of set S, with at most $\sum_{1 \leq i \leq k} |S_i| = n$ leaves, we can transform T into a binary tree (this process is called binarize) as described in Lemma 4 in linear time, because every node of the GST has at most degree $|\Sigma|$ and thus can be transformed into a binary tree in $O(1)$ time. Consequently if we run the bottom up process at the "binarized" tree and perform at every internal node u the producing and merging steps, using the smaller leaf-list (the one that corresponds to the subtree with the smaller number of leaves) against the other, then Lemmas 1, 2, 3 ensure that overall the process takes $O(n \log n)$ time. Consequently, searching for maximal pairs with bounded gaps in a set of strings can be performed in linear space and $O(n \log n + a)$ time, where a is the size of the output.

4 Weighted Sequences: Data Structures and Algorithms

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).

4.1 Data Structure Description

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 15. *Let S be a weighted sequence. For every suffix starting at position i we define a list of possible weighted substrings 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 substring 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 substrings 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 $L(v) = S_{i,j}$ and $\pi(S_{i,j}) \geq 1/k$, where $j > 0$ denotes the j-th weighted substring 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. 2 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:

– Prefixes for suffix $x_{1\cdots 11}$: $S_{1,1} = ACTT\underline{A}TC\underline{A}TTT$, $\pi(S_{1,1}) = 0.25$, and $S_{1,2} = ACTT\underline{C}TC\underline{A}TTT$, $\pi(S_{1,2}) = 0.25$.

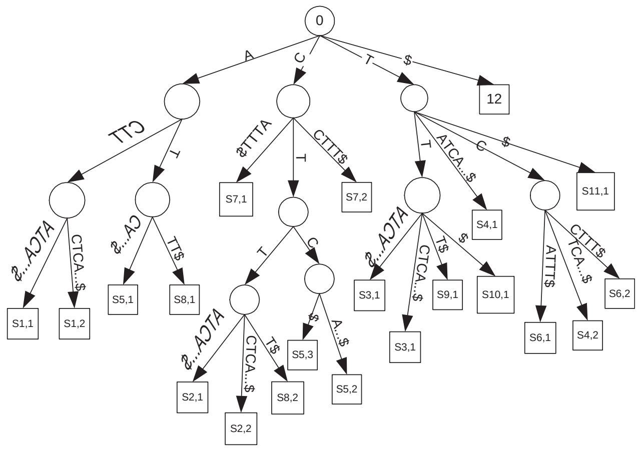

Fig. 7. A Weighted Suffix Tree example.

– Prefixes for suffix $x_{2...11}$: $S_{2,1} = CTT\underline{A}TC\underline{A}TTT$, $\pi(S_{2,1}) = 0.25$, 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 substrings appears in Fig. 7.

4.2 The Construction of Weighted Suffix Tree

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 substrings 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 $1 - 1/k$,
- 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 substring (i.e. $\pi(\mathit{substring}) \geq 1/k$). 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 substrings starting from this position is created. The production of the possible substrings is done as follows: moving rightwards, we extend the current substrings by adding the same single character whenever we encounter a white or gray position, only one possible choice, and creating new substrings at black positions where potentially many choices are provided. The process is illustrated in Fig. 8. At this point we define for every produced substring two cumulative probabilities π', π''. The first one measures the actual substring probabilities and the second one is defined by temporarily treating gray positions as white. The generation of a substring 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 substring may actually be shorter as π' (which incorporates gray positions) may have met the $1/k$ threshold earlier. For every substring we store the difference D of the actual ending position and the extended one as shown in Fig. 9. Notice that only the actual substrings need to be represented with the GST.

Step 3: Having produced all the substrings from every black position, we insert the actual substrings in the generalised suffix tree in the following way. For every substring we initially insert the corresponding extended substring 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 substring of the actual substring $X = X_{i..\,i+f-1}$ $(f \leq$

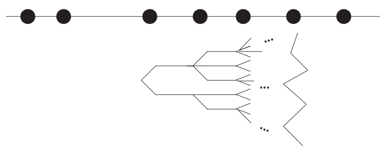

Fig. 8. Producing all possible substrings from left to right

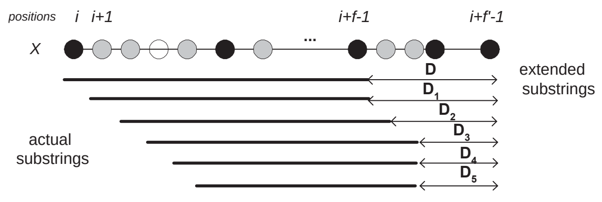

Fig. 9. Insertion of substrings in the GST

f') that begins at black position i of the weighted sequence in Fig. 9. Observe the following two facts:

– 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 substring ends, since π' does not take gray positions into account (cf. Fig. 9). No suffix can end though at a position greater than $i+f'-1$, where the extended substring ends.

We have kept every leaf storing a suffix of X', in a list L. Let D_j denotes the redundant portion of suffix $X'_{i+j..i+f'-1}$ of X' (cf. Fig. 9). After we have inserted the extended substring and the proper suffixes using McCreight's algorithm [13], 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.

4.3 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 5. *At most $O\left(|\Sigma|^{\log k/\log(\frac{k}{k-1})}\right)$ substrings 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 substring 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 substring 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 substring 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 substrings are produced. Hence, every position i of the weighted sequence can be the starting point of at most $|\Sigma|^l$ number of substrings.

Lemma 6. *The number of substrings 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 substrings (Lemma 5), the total number of substrings is $O(n)$.

Lemma 7. *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. 10). Each window contains $l = \log k/\log(\frac{k}{k-1})$ black positions.

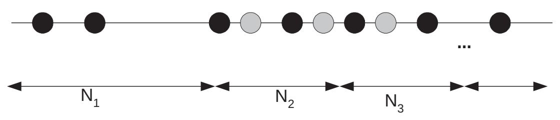

Fig. 10. Time cost for step 2

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 $O(1)$ substrings (according to Lemma 5) 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 substrings 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 8. *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 substrings 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 substrings 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 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 $O(n)$ substrings (by Lemma 6) 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 $O(n)$.

4.4 Applications

The WST is endowed with most of the sequence manipulation capabilities of the Generalized Suffix Tree. Some of its applications are listed below.

Exact pattern matching: There are two versions of the exact pattern matching problem, one when the pattern is unweighted and the other when it is weighted. Pattern matching with unweighted pattern proceeds as with the regular Suffix Tree matching procedure. Thus, if after having spelled the entire pattern from the tree root, we end up in an internal node, all the leaves of its subtree are reported. On the other hand when the pattern is weighted, its non-weighted counterparts are derived and pattern matching reduces to the above case. For a pattern of size m, pattern matching takes $O(m + a)$ time, a the number of reported occurrences.

Finding repetitions in weighted sequences: The WST can be used in order to compute all the repetitions in a given weighted sequence, each repetition having probability of appearance greater than $1/k$. The WST with parametre $1/k$ is constructed and then the repetition finding problem is reduced to a depth-first traversal of the tree, during which a leaf-list is kept for each internal node. If the size of the list exceeds two, this constitutes a repetition. Thus, its elements are reported. Apparently, the problem can be solved in $O(n + a)$ time, where n is the sequence length and a is the answer size (cf. [11]).

Sequence alignment: In the sequence alignment problem we are looking for the best alignment of two sequences S_1 and S_2 which minimises the edit distance of S_1 and S_2 (or in other words maximises the similarity measure of S_1 and S_2). The suffix tree can be used in combination with dynamic programming to produce a hybrid dynamic programming method that is faster than dynamic programming alone(for more details see [9]).

When we consider the sequence alignment problem for two weighted sequences, we have to incorporate the notion of probability in the produced alignment. The edit distance between two weighted sequences is labeled with the probability of appearance of the respective weighted factors. The WST can be used instead of the ordinary suffix tree to efficiently compute the alignment of all pairs of weighted substrings for two given weighted sequences.

Longest common substring of weighted sequences: The Generalized Weighted Suffix Tree is built for two weighted sequences w_1 and w_2. Subsequently, a traversal of the tree that computes the internal node with the greatest depth is required. The string being spelled at this node is the longest weighted subsequence of the two weighted strings. The process can be completed in $O(n_1 + n_2)$ time, n_1,n_2 the sizes of w_1,w_2, respectively.

5 Conclusions

In this chapter we have presented efficient Data Structuring Applications and Algorithms for solving string manipulation problems derived from different research areas.

We initially discussed the Approximate Matching Problem with Gaps, and consequently the Model Inference Problem in Multiple Strings. Finally we presented the Weighted Suffix Tree, an efficient data structure for solving string manipulation problems in weighted sequences.

In our point the above presented techniques can be a useful tool to every researcher who wants to study other string manipulation problems. In the Approximate Matching Problem with Gaps it would be interesting to explore new versions of the problem as well as more efficient algorithms than the previously described. We should point out that the presented algorithms were based mainly on dynamic programming and on the equivalence of some optimization problems on strings to problems on graphs.

In the Model Inference Problem it could be interesting to design efficient algorithms to identify not only pairs of equal substrings in multiple strings, but a collection of equal substrings that can meet some restrictions on gaps. A naive approach (although not sufficient) could be the reporting of maximal pairs and the combination of them building triads, quadruplets and so on.

Finally the Weighted Suffix Tree can solve a variety of string manipulation problems in weighted sequences.

References

1. Brodal, G.S., R. B. Lyngso, C. N. Storm Pedersen and Jens Stoye.: Finding Maximal pairs with bounded gaps. Journal of Discrete Algorithms, Vol. 1(1), pages 77-104, 2000.
2. Brown, M.R., Tarjan, E.: A Fast Merging Algorithm. Journal of the ACM, Vol. 26(2), pages 311–226, 1979.
3. Cambouropoulos, E., Crawford T., Iliopoulos, C., S.: Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects. In Proc. of the AISB'99 Convention (Artificial Intelligence and Simulation of Behavior),pages 42–47, 1999.
4. Cambouropoulos, E., Crochemore, M., Iliopoulos, C., S., Mouchard, L., Pinzon, Y.: Algorithms for computing approximate repetitions in musical sequences. In Proc. of the 10^{th} Australian Workshop on Combinatorial Algorithms, pages 129–144, 1999.
5. Crochemore, M., Iliopoulos, C.,S., Pinzon, Y.,Rytter, W.: Finding Motifs with Gaps. Unpublished manuscript.

6. Crochemore, M., Iliopoulos, C., Makris, Ch., Rytter, W., Tsakalidis, A., Tsichlas, K.: Approximate String Matching With Gaps. Nordic Journal of Computing, Vol. 9(1), pages 54–66,2002.
7. Crochemore M., Iliopoulos C. S. , Lecroq T., and Pinzon Y. J.: Approximate string matching in musical sequences. In M. Balik and M. Simanek, editors, Proceedings of Prague Stringology Club Workshop (PSCW'01), pages 26-36, Czech Technical University, Prague, Czech Republic, 2001.
8. Gajewska, H., Tarjan, R.,E.: Deques with heap order. Information Processing Letters, Vol. 12(4), pages 197–200, 1986.
9. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York, 1997.
10. Hirschberg, D., S.: A linear space algorithm for computing maximal common subsequences. Comm. Assoc. Comput. Mach. Vol. 18(6), pages 341–343, 1975.
11. Iliopoulos, C., Makris, Ch, Panagis, I., Perdikuri, K., Theodoridis, E., Tsakalidis, A. . Efficient Algorithms for Handling Molecular Weighted Sequences, In 3rd IFIP International Conference on Theoretical Computer Science, 2004, to appear.
12. Iliopoulos, C.S., Makris C., Sioutas S., Tsakalidis A., Tsichlas K., : Identifying Occurrences of Maximal Pairs in Multiple Strings. Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching,Lecture Notes In Computer Science, pages 133 - 143, 2002.
13. McCreight, E.,M.,: A space-economical suffix tree construction algorithm. Journal of the ACM, Vol. 23(2), pages 262–272, 1976.
14. Ukkonen, E.,: On-line construction of suffix trees. Algorithmica, Vol. 14(3), pages 249–260, 1995.