Computation of Repetitions and Regularities on Biological Weighted Sequences*

M. Christodoulakis[1], C. Iliopoulos[1], L. Mouchard[1],
K. Perdikuri[2], A. Tsakalidis[2], and K. Tsichlas[1]**

[1] Department of Computer Science, King's College London,
London WC2R 2LS, England.
{manolis,csi,mouchard,kostas}@dcs.kcl.ac.uk

[2] Computer Engineering & Informatics Dept., University of Patras
26500 Patras, Greece.
{perdikur,tsak}@ceid.upatras.gr

Abstract. Biological Weighted Sequences are used extensively in Molecular Biology as profiles for protein families, in the representation of binding sites and often for the representation of sequences produced by a shotgun sequencing strategy. In this paper we address three fundamental problems in the area of Biological Weighted Sequences: i) Computation of Repetitions, ii) Pattern Matching and iii) Computation of Regularities. To the best of our knowledge, this is the first time these problems are tackled in the relative literature. Our algorithms can be used as basic building blocks for more sophisticated algorithms applied on weighted sequences.

* A preliminary form of the results in this paper were presented in the conferences *Fun with Algorithms* [Iliopoulos et al. 2004b], *CompBionets* [Christodoulakis et al. 2004a] and *ICCMSE* [Christodoulakis et al. 2004b].

** Correspondent Author, Tel.: 0044 (0) 2078481631, FAX: 0044 (0) 20

1 Introduction

The complete sequence of *the* human genome chromosomes is now almost obtained. Various donors of diverse ethnogeographic categories (e.g., African-American, Chinese, Caucasian, etc.) have been enrolled and offered samples that were used to obtain the final reconstructed genome sequence. Despite the fact that natural polymorphism is necessarily faded by this *consensus*, the final sequence corresponds, more or less, to the individual sequences from the donors.

Hopefully, the sequences of the genes, that contain coding information which produces proteins, are usually much more conserved than non-coding regions, and genes are what most of the research teams are looking for. The facts that several (up to six) codons can encode the same amino acid and that different amino acids can have identical chemical properties explain that close (or even distant) DNA sequences can produce protein sequences that fold identically and are able to perform exactly the same task.

In order to represent the various nucleotides that can be found at a given location, the original DNA alphabet $\{A, C, G, T\}$ has been naturally extended to a more complex one, the IUB/IUPAC where a letter represents several nucleotides, e.g. $D \rightarrow A, G, T$ or $M \rightarrow A, C$.

Gene expression is controlled from an important region, upstream from the gene. This region contains several important binding sites where additional proteins, initiating or regulating the transcription, attach themselves to the DNA sequence. Binding sites, are sites in a biological molecule that will come into contact with a site in another molecule permitting the initiation of some biological process (for instance, transcription or translation). These binding sites are usually very specific DNA sequences, that tolerate only very elementary changes,

they are the keyholes of specific keys and any violent alteration prevents the additional protein to bind, and moreover the gene sequence from being transcripted [Stormo 2000]. The sequences that are found between the binding sites might differ from one organism to the other.

To illustrate these concepts, let us consider the regulatory regions of the hemoglobin genes ($\alpha, \beta, \gamma, \zeta$ hemoglobins more precisely) where uppercase letters correspond to the gene sequence and bold lowercase letters correspond to a specific binding site sequence.

Hemoglobin α ... cg**gcactct**tctggtcccc ... atacccaccgATGGTGCT ...

Hemoglobin β ... tgacacaact**gcaacct**ca ... aacagacaccATGGTGCA ...

Hemoglobin γ ... gccgcta**ccgccct**gcgcg ... atgcgcgagtATGGTGCT ...

Hemoglobin ζ ... gct**gcaacct**gcccactcc ... ggcagcgcacATGTCTCT ...

We obtained four sequences we have to align:

Hemoglobin α	g c a c t c t
Hemoglobin β	g c a a c c t
Hemoglobin γ	c c g c c c t
Hemoglobin ζ	g c a a c c t
Consensus	g c a M c c t

As a result, various techniques are used to represent these polymorphisms, using either the extended alphabet we presented before or a more precise encoding that takes into account the relative frequency of each nucleotide. We can consider mainly two techniques, named Position Weight Matrices (PWM for short) where for each position, the probability of each nucleotide is given, and logo sequences [Schneider et al. 1990].

The Position Weight Matrix [Thompson et al. 1994] of a set of strings of length m is a $4 \times m$-matrix that reports the frequency of each nucleotide for all possible locations. In our example, we have:

	1	2	3	4	5	6	7
a	0	0	0.75	0.5	0	0	0
c	0.25	1.00	0	0.5	0.75	1.00	0
g	0.75	0	0.25	0	0	0	0
t	0	0	0	0	0.25	0	1.00

Weighted Sequences are extensively used in Computational Molecular Biology for representing relatively short sequences such as binding sites as well as long sequences such as profiles of protein families (see [Gusfield 1997], 14.3). In addition, they are also used to represent complete chromosome sequences (see [Gusfield 1997], 16.15.3) that have been obtained using a whole-genome shotgun strategy [Venter et al. 2001,Myers et al. 2000] with an adequate cover. The cover is the average number of fragments that appear at a given location. Usually, the cover is large enough so that errors as well as SNPs are clearly spotted and removed by the consensus step. In the case of whole-genome shotgun processes, we would like to dig out information that has been previously undetected after being faded during the consensus step (for example the consensus step wrongly chooses a symbol for a specific position than another). As a result, errors in the genome are not removed by the consensus step but remain and a probability is assigned to them based on the frequency of symbols in each position.

These are some biological examples where weighted sequences can be important. Generally speaking, a weighted sequence could be defined as a *sequence of (symbol, weight) pairs*, $S = \langle (s_1, w_1), (s_2, w_2), \cdots (s_n, w_n) \rangle$, where w_i is the weight of symbol s_i in position i (i.e. occurrence probability of s_i at position i).

Note that this definition is more general than the one that refers to weighted alphabets. In a weighted alphabet the weight of a letter is fixed and does not depend on the position in the string. In our case, the weight is not fixed.

Our goal is to design efficient algorithms for a set of string operations on these sequences. Suppose for example that a biologist wants to find whether a sequence at hand (which may as well be a weighted sequence) occurs in a specific protein family with high probability in order to decide whether a specific protein belongs in a family of proteins. This can be accomplished by pattern matching algorithms on weighted sequences. Weighted sequences have also been used in event management systems [Wang et al. 2003].

In this paper we propose efficient algorithms for a set of problems on weighted sequences taken from the Computational Biology area. Our results can be compared with the Weighted Suffix Tree (WST) [Iliopoulos et al. 2003a]. The WST has all the merits of the usual suffix tree with the difference that its construction is heavily based on the choice of the probability of occurrence $\frac{1}{k}$. It is crucial for its construction that k is a fixed and small constant. When k changes then the suffix tree must be reconstructed from scratch. In addition, for arbitrary k, the size of the WST is exponentially growing and so k must be a very small constant. Our algorithms for pattern matching are more general since they allow for arbitrary k, while at the same time we investigate pattern matching with gaps.

In addition, we use known techniques to discover repetitions and regularities, in particular covers, of weighted sequences. We show that a well known technique by Crochemore [Crochemore 1981] cannot be used efficiently in the case of weighted sequences. In fact, the $O(n \log n)$ time algorithm given in [Iliopoulos et al. 2003b] has a slight error and in fact it is an $O(n^2)$ time al-

gorithm. We use another well known technique [Karp et al. 1972] to compute repetitions of a particular length (as well as covers) in $O(n \log n)$ time.

The paper is organised as follows. In Section 2, we provide basic definitions on weighted sequences. In Section 3 we describe an efficient algorithm for finding repetitions in a weighted sequence while in Section 4 we design an algorithm for locating the covers of a weighted sequence. In Section 5, algorithms for some versions of exact pattern matching problem are presented. Finally, some concluding remarks are given in Section 6.

2 Preliminaries

In this section we provide some definitions needed throughout the paper. In addition, the problem definition is given and two probability measures are introduced.

2.1 Basic Definitions

Let $\Sigma = \{1, 2, \ldots, \sigma\}$ be an alphabet of cardinality $\sigma = |\Sigma|$. A sequence s of length n is represented by $s[1..n] = s[1]s[2]\cdots s[n]$, where $s[i] \in \Sigma$ for $1 \leq i \leq n$, and $n = |s|$ is the length of s. Sequence s is also called a *solid* sequence in order to distinguish them from weighted sequences. An empty sequence is denoted by ε; we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. A factor f of s is a substring of s, that is $f = s[i \ldots j]$. A weighted sequence is defined as follows.

Definition 1. *A weighted sequence $s = s_1 s_2 \cdots s_n$ over an alphabet Σ is a sequence of sets of couples. In particular, each s_i is a set $((1, \pi_i(1)), (2, \pi_i(2)), \ldots, (\sigma, \pi_i(\sigma)))$, where $\pi_i(q)$ is the occurrence probability of character q at position i. For every position $1 \leq i \leq n$, $\sum_{q=1}^{\sigma} \pi_i(q) = 1$.*

We represent each position of the weighted sequence as a vector that contains all the symbols of the alphabet and their corresponding probabilities; if a character does not appear in a specific position then its probability will be zero. An instance of a weighted (sub)sequence p is a (sub)sequence of p where a symbol has been chosen for each position. We will represent each couple $(q, \pi_i(q))$ as $\pi_i(q)_q$, as shown in the example below.

Example 1. Consider the alphabet $\Sigma = \{A, C, G, T\}$. Then

$$s = \begin{pmatrix} 0.3_A & 0.25_A & 0_A & 0.4_A & 0.8_A & 0_A \\ 0_C & 0.25_C & 1_C & 0.2_C & 0.05_C & 0.5_C \\ 0.2_G & 0.5_G & 0_G & 0.2_G & 0.1_G & 0_G \\ 0.5_T & 0_T & 0_T & 0.2_T & 0.05_T & 0.5_T \end{pmatrix} \tag{1}$$

is the Position Weight Matrix that represents a weighted sequence of length 6. Note that the sum of probabilities for each column is 1. In this case $s[1]$ is the set of couples $\{(A, 0.3), (C, 0), (G, 0.2), (T, 0.5)\}$.

Definition 2. *A symbol q occurs at position i of a weighted sequence $s = s[1 \dots n]$ if and only if the probability of occurrence of symbol q at position i is greater than zero, $\pi_i(q) > 0$.*

Example 2. Let s be the weighted sequence defined by the Position Weight Matrix (1). Then symbol A occurs at positions 1, 2, 4, and 5 of s. Similarly, symbol C occurs at 2, 3, 4, 5, and 6.

The following definition clarifies when a solid pattern p occurs in a weighted text t.

Definition 3. *The solid pattern $p = p[1 \dots m]$ occurs at position i of the weighted text $t = t[1 \dots n]$ if and only if $p[j]$ occurs at position $t[i+j-1]$ for all $1 \leq j \leq m$; that is, if and only if $\pi_{i+j-1}(p[j]) > 0$, for all $1 \leq j \leq m$ by Definition 2. We also say that p matches t at position i.*

Example 3. Let t be the weighted sequence defined in Equation (1). Then $p = ACTA$ occurs in t at position 2, since $\pi_2(A) = 0.25$, $\pi_3(C) = 1$, $\pi_4(T) = 0.2$, and $\pi_5(A) = 0.8$.

2.2 Metrics of Information Content on Weighted Sequences

Since each symbol at position i of the text t is assigned a probability of occurrence it is logical to assume that an occurrence of a solid pattern p in the weighted text t must also have a probability of occurrence. In this way, we define how likely is to find an occurrence of p in a specific position of t. In the following we provide two different matching measures that could serve as information content metrics on weighted sequences.

Definition 4. *Let $p = p[1 \dots m]$ be a solid pattern, and $t = t[1 \dots n]$ be a weighted text. Also assume that p matches t at position i. Then the probability of the match can be defined in one of the following ways:*

1. ***Multiplicative Probability*** *is the product of the probabilities of the symbols of t that match p: $P^i_{prod} = \prod_{j=1}^{m} \pi_{i+j-1}(p[j])$*
2. ***Average Probability*** *is the average of the probabilities of the symbols of t that match p: $P^i_{aver} = \frac{\sum_{j=1}^{m} \pi_{i+j-1}(p[j])}{m}$*

Note that the Average Probability Measure (APM) allows for characters with zero probability. In general, the average probability measure is by far less strict than the Multiplicative Probability Measure (MPM) for a given cut-off probability $\frac{1}{k}$. It is easy to see that all matches of the MPM are contained in the set of matches for APM. As a result, by using the APM we are more loose with respect to the matches we are going to get while by using MPM we become much more strict.

Example 4. Let t be the weighted sequence defined in Equation (1) and $p = ACTA$, which occurs in t at position 2. Then the multiplicative probability of

this match is $P^2_{prod} = 0.25 \cdot 1 \cdot 0.2 \cdot 0.8 = 0.04$ and the average probability of the same match is $P^2_{aver} = \frac{0.25+1+0.2+0.8}{4} = \frac{2.25}{4} = 0.56$

The above definitions can be extended to cover the case where both the pattern p and the text t are weighted sequences. In this case we denote by $\pi_{t_i}(q)$ the probability of the symbol q at position i of t, and by $\pi_{p_j}(q)$ the probability of the symbol q at position j of p.

Definition 5. *Let $p = p[1..m]$ and $t = t[1..n]$ be weighted sequences. We say that positions $p[j]$ and $t[i]$* match with respect to symbol q *if $\pi_{p_j}(q) \times \pi_{t_i}(q) > 0$; that is, if q occurs at both $p[j]$ and $t[i]$.*

However, note that $p[j]$ and $t[i]$ may match with respect to many symbols. As a result, we define the match between two positions as follows:

Definition 6. *Let the pattern $p = p[1..m]$ and text $t = t[1..n]$ be weighted sequences. We say that $p[j]$ and $t[i]$* match *if $\sum_{q=1}^{\sigma} \pi_{p_j}(q) \times \pi_{t_i}(q) > 0$; that is, if $p[j]$ and $t[i]$ match with respect to (at least one) q.*

Example 5. Consider $p_j = \begin{pmatrix} 0.3_A \\ 0.1_C \\ 0_G \\ 0.6_T \end{pmatrix}$ and $t_i = \begin{pmatrix} 0_A \\ 0.8_C \\ 0_G \\ 0.2_T \end{pmatrix}$ for some position j of the pattern, and some position i of the text. Then $p[j]$ and $t[i]$ match because there is at least one symbol (precisely two: C and T) that occurs in both of them. Confirm that $\sum_{q \in \{A,C,G,T\}} \pi_{p_j}(q) \times \pi_{t_i}(q) = 0.2 > 0$.

Similarly to Definition 3 we get the following definition of a match, in the case where both the pattern and the text are weighted.

Definition 7. *Let $p = p[1..m]$ and $t = t[1..n]$ be weighted sequences. We say that p* occurs *in (or* matches*) t at position i if and only if*

$$P^i_{w\_aver} = \frac{\sum_{j=1}^{m} \sum_{\forall s \in \Sigma} \pi_{p_j}(s) \times \pi_{t_{i-m+j}}(s)}{m} > 0$$

We call $P^i_{w\_aver}$ the weighted average probability of the match *of p at position i of t. Similarly, we can define the* weighted multiplicative probability of the match *of p at position i of t.*

For the rest of the paper we will assume that p is a solid pattern. For the cases where the algorithm is applicable to weighted patterns as well, it will be explicitly indicated. Moreover, we will be interested in occurrences of the pattern with probability larger than a threshold $\frac{1}{k}$, $k \geq 1$, using the **Multiplicative Probability** metric.

If we do not interpret the weight as a probability then other matching measures may become interesting. For example, the maximum weight matching measure, where the weight of a match between a solid pattern p of length m and a weighted text t at position i is $\max_{1 \leq j \leq m}\{w_{i+j-1}(p[j])\}$. By $w_i(q)$ we represent the weight of symbol q at position i of text t. We believe that our algorithms can be extended to tackle these matching measures too.

3 Computation of repetitions

In solid sequences the algorithms of Crochemore [Crochemore 1981] and Karp [Karp et al. 1972] are well known and have $O(nlogn)$ time complexity for computing repetitions. Their difference is that the first algorithm computes repetitions of all possible lengths while the second can compute repetitions of prespecified length.

However Crochemore's algorithm (henceforth Algorithm `C`) fails to find repetitions in weighted sequences in $O(nlogn)$ time. In fact it needs $O(n^2)$ time to be able to compute all repetitions. However Karp's algorithm (henceforth Algorithm `KMR`) can be applied for computing repetitions in weighted sequences.

3.1 Why Algorithm `C` Fails for Weighted Sequences

This section assumes that the reader is familiar with the algorithm of Crochemore for finding repetitions [Crochemore 1981]. We briefly describe the algorithm, to assist the discussion of the application of this algorithm on weighted sequences later on. Given a string x of length n, the algorithm proceeds in precisely $n/2$ stages, where at stage i a vector E_i of length $O(n)$ is being computed. E_i is a vector of integers (called *classes*), representing factors of length i. A *repetition* of length i can then be identified starting at position j of x if and only if $E_i[j] = E_i[j+i] = \ldots = E_i[j+ki]$, for some k.

As explained above, the algorithm performs $O(n)$ stages. In order to work in $O(n \log n)$ overall time, each vector E_i cannot be computed from scratch; instead, E_i is computed at stage i by modifying a limited number of positions of the vector obtained in the previous stage, E_{i-1}. But since E_{i-1} contains integers representing factors of length $i-1$, and E_i integers representing factors of length

i, should not *all* values of E_i be different from the corresponding values in E_{i-1}? The trick is that the values of E_{i-1} that are not modified in E_i, *implicitly* specify *new* factors, of length i. The choice of which values of E_{i-1} should be modified is based on the number of occurrences of each integer (factor) in E_i, i.e. the values to be modified are for those factors that appear less often.

For example, for some string $x \in \{a, b\}^*$, consider the integer 7 in E_3 representing the factor aba; at the next stage factors of length 4 are computed; thus the class 7 of E_3 will split into "subclasses": those aba that are followed by a (thus giving $abaa$), and those that are followed by b (giving $abab$). If we know that $abaa$ appears more often in x than $abab$ we can leave unmodified all those positions of E_3 that contain 7 and are followed by a (thus, 7 represents at this stage the factor $abaa$) and assign a new integer to those positions in E_{i-1} that contain 7 and are followed by b. Of course, the same procedure should be followed for all classes of E_{i-1}.

The problem on weighted sequences becomes clear: we *cannot* increment factors implicitly; we have to update their probabilities of occurrence at each step so that we know whether these repetitions have a probability $\geq \frac{1}{k}$. As a result, we are obliged to process all classes which leads to an $O(n^2)$. The authors of [Iliopoulos et al. 2003b] did not notice this problem so they claimed that the complexity is $O(n \log n)$, which is wrong. Alternatively, one could try find all the repetitions without computing probabilities, and then compute the probabilities of the actual repetitions. This is no better because the length of each repetition can be $O(n)$ (thus, $O(n)$ multiplications) for each repetition. Moreover, if we don't compute probabilities throughout the algorithm we might end up with

$O(|\Sigma|^n)$ factors. As a result, it seems that adopting this approach for finding repetitions in weighted sequences will probably not lead to an $o(n^2)$ algorithm.

3.2 Algorithm KMR

Algorithm KMR computes equivalence classes, similar to Algorithm C, but it proceeds in $\log n$ stages of $O(n)$ time each. It has been successfully applied to weighted sequences [Iliopoulos et al. 2004b].

Throughout the algorithm, integers (called *equivalence classes*) are used to represent factors. Since any position in a weighted sequence may contain more than one symbols, more than one factors of the same length can start from the same position. Identifying repetitions then comes down to identifying positions that contain *at least one* common factor. Those positions are called *equivalent*. More formally:

Definition 8. *Given a weighted sequence s, positions i and j of s are a-equivalent ($a \in \{1, 2, \ldots, n\}$ and $i, j \in \{1, 2, \ldots, n - a + 1\}$) —written iE_aj— if and only if there exists at least one substring f of length a, that occurs at (starts at) both positions i and j.*

An equivalence relation E_a is represented as an $n-a+1$ vector $v_1^{(a)}v_2^{(a)}\ldots v_{n-a+1}^{(a)}$ of sets of integers, where each $v_i^{(a)}$ contains the labels of the equivalence classes of E_a to which each factor starting at position i belongs.

The following lemma (see Fig. 1) is the basic mechanism of the KMR algorithm:

Lemma 1. *For integers i, j, a, b with $b \leq a$ we have $iE_{a+b}j$ precisely when iE_aj and $i + bE_aj + b$ (1) or, equivalently, when iE_aj and $i + aE_bj + a$ (2).*

To solve the problem of locating repeated substrings of length d over a weighted sequence w, KMR algorithm proceeds as follows:

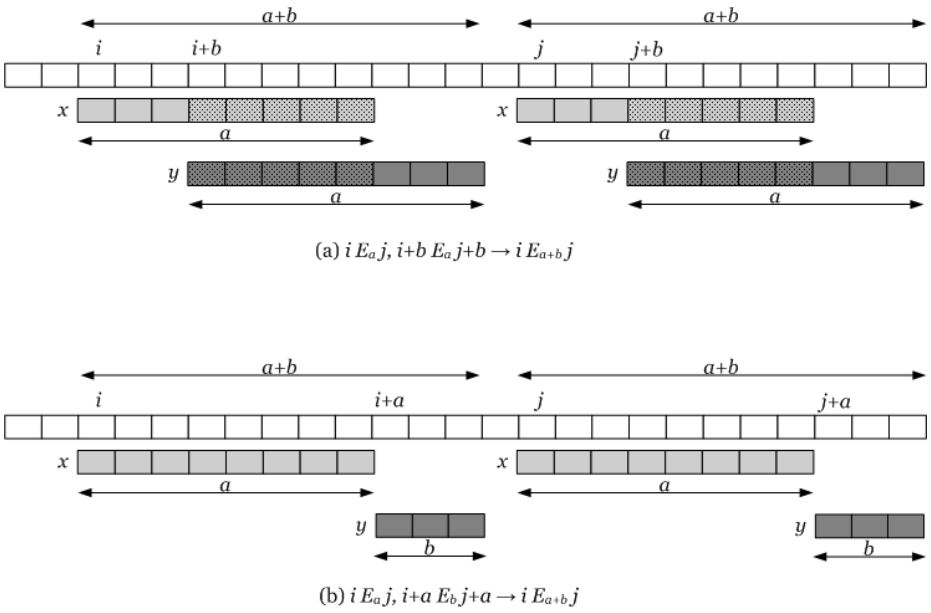

Fig. 1. Constructing the equivalence relation E_{a+b} from E_a and E_b

1. Scan w to construct relation E_1
2. Use Lemma 1 to construct successively, the relations $E_2, E_4, E_8, \ldots, E_r$, where $r = 2^{\lfloor \log_2 d \rfloor}$.
3. Use Lemma 1 to construct the relation E_d from the relations $E_1, E_2, \ldots, E_r$

The first stage of the algorithm –constructing E_1– is straightforward. Constructing new equivalence relations from existing ones (used repeatedly in stages 2 and 3) is what follows. This procedure is using $e_a + e_b$ pushdown stores, $P(1), \ldots, P(e_a)$ and $Q(1), \ldots, Q(e_b)$, where e_a, e_b represent the number of distinct equivalence classes in E_a and E_b respectively.

1. Sort the vector $v^{(a)}$ using the P-pushdown stores; that is, run through $v^{(a)}$, and for each factor x at each position i, push i into $P(x)$. Note that the same position i may be pushed into more than one stacks. So far, having the

same position i in more than one P-stack causes no problem, since these stacks are distinct. But, for the sake of the explanation, let's distinguish them in the following way: we will use i_x to denote the position i when it refers to the factor x starting at i; thus, $i_{x'}$ will denote the same position i but referring to it's second factor x' ($\neq x$).

2. In success, pop each $P(x)$ until it is empty. As the number i_x is popped from $P(x)$, push it into the Q-pushdown stores $Q(y)$ $y \in v_{d+a}^{(b)}$ provided that $d + a \leq n - (a + b - 1)$. Note that there may be more than one factors y of length b starting at position $d + a$. Therefore, position i_x will be pushed into all appropriate Q stacks. However, when another P stack, say $P(x')$, is popped, it is possible that the same position $i_{x'}$ (referring to different factor) will be pushed into the same Q stacks as i_x. Had we not distinguished i_x from $i_{x'}$, we would end up with the same position i appearing more than once in the same stack $Q(y)$, and of course the ambiguity as to which factor starting at position i this refers to, would make it impossible to go on to the third step.

3. Finally, construct $v^{(a+b)}$: Successively pop each Q-stack until empty. Start with a variable class counter c initially set to 1. As each i_x is popped from a given stack $Q(y)$ test whether or not x is equal to x', where x' is the factor represented by position j just previously popped from the same stack; that is, element $j_{x'}$ was previously popped. If this is so, then iE_aj and $i + bE_aj + b$, so $iE_{a+b}j$; therefore, insert c in the set $v_d^{(a+b)}$. Otherwise we have $i + bE_aj + b$ but *not* iE_aj; therefore, insert $c + 1$ into $v_d^{(a+b)}$ and increment c to $c + 1$. When stack $Q(y)$ is exhausted, increment c to $c + 1$ before beginning to pop the next Q-stack. Whenever i_x is the first element from a Q-stack, insert c into $v_d^{(a+b)}$ automatically.

An example of this procedure –constructing E_5 using the precomputed equivalence relations E_3 and E_2– for a weighted string w is shown in Fig. 2. Note that for simplicity the probabilities of the factors have been omitted and, the actual factors (rather than integers representing those factors) appear in $v^{(3)}$ and $v^{(2)}$. Moreover, a table showing the correspondence between the new integers and the factors they represent, is given for the reader to verify the correctness of the algorithm.

3.3 Expected Running Time Analysis

As previously described, in our approach, we use the definition of *Equivalent Families of Repetitions* on a weighted sequence X in order to compute all repetitions of length d. Since X is a weighted sequence we have to bound the size of each equivalence relation E_i. In the following lines we analyze the time complexity of the algorithm using two important lemmas.

Based on the probability threshold k we split the positions of a weighted sequence X into three distinct categories:

1. *White Positions:* are positions that contain a symbol with probability 1. We also call them *solid positions*.
2. *Gray Positions:* are positions that contain a symbol with probability larger than $1 - \frac{1}{k}$. Since the threshold is $\frac{1}{k}$, no other character can appear in this position.
3. *Black Positions:* are positions where two or more characters have probability larger than $\frac{1}{k}$. These positions are also called *branching positions*.

The size of the equivalence relation E_1 is at most $|\Sigma|n$. This is because in each position each character might appear with a non-zero probability. As a result,

Text

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	a	b	a	b	a	b	a	b	b	a
	b					a	b					

1	2	3	4	5	6	7	8	9	10	11
aba	*baa*	*aab*	*aba*	*bab*	*aba*	*bab*	*aba*	*bab*	*abb*	*bba*
	bab	*abb*	*bba*			*baa*	*abb*	*bbb*	*bbb*	
							aaa	*aab*		
							aab	*abb*		

Equival. $v^{(3)}$

Relations

1	2	3	4	5	6	7	8	9	10	11	12
ab	*ba*	*aa*	*ab*	*ba*	*ab*	*ba*	*ab*	*aa*	*ab*	*bb*	*ba*
		ab	*bb*				*aa*	*ab*	*bb*		
								ba			
								bb			

$v^{(2)}$

Step 1 P-stacks

		8_{aba}	10_{abb}		9_{bab}		
	9_{aab}	6_{aba}	9_{abb}		7_{bab}		
	8_{aab}	4_{aba}	8_{abb}	7_{baa}	5_{bab}	11_{bba}	10_{bbb}
8_{aaa}	3_{aab}	1_{aba}	3_{abb}	2_{baa}	2_{bab}	4_{bba}	9_{bbb}

Step 2 Q-stacks

aaa	*aab*	*aba*	*abb*	*baa*	*bab*	*bba*	*bbb*
		9_{bbb}					
		4_{bba}	7_{bab}				
	5_{bab}	2_{bab}	7_{baa}				
	7_{bab}	9_{bab}	8_{abb}				
	7_{baa}	2_{baa}	1_{aba}				
	3_{abb}	9_{abb}	6_{aba}				
	1_{aba}	4_{aba}	8_{aba}				
5_{bab}	6_{aba}	6_{aba}	8_{aab}				
6_{aba}	3_{aab}	9_{aab}	8_{aaa}				

18

aa	*ab*	*ba*	*bb*

Step 3 $v^{(5)}$

1	2	3	4	5	6	7	8	9
6	10	5	9	1	2	3	17	8
18	11	7	13	3	6	4	18	10
					13	15	19	12
					18	16	20	14

1 *babaa* 6 *abaab* 11 *baaba* 16 *baabb*

each class C_f^1, has n positions. For the equivalence relation E_2, the number of possible different classes C_f^2, is $|\Sigma|^2$. Consequently, in the worst case, the size of each class will be n, and as a result $|E_2| \leq |\Sigma|^2 n$. Continuing the same way, the size of the equivalence relation will be at some point $O(|\Sigma|^n n)$ (e.g. when the the repetitions are of size $\frac{n}{4}$). However, we did not take into account the fact that the probability of occurrence $\frac{1}{k}$ is rather large, which means that k is assumed to be small ($k \leq 10$).

Intuitionally, since k is a small fixed constant we cannot have long chains of symbols with small probability. As a result, the number of possible and valid ($pr() \geq \frac{1}{k}$) repetitions will be much smaller as we extend the length of the repetitions.

Lemma 2. *Each possible factor of X may contain up to $\lambda = \left\lfloor \log_{\frac{k-1}{k}} \frac{1}{k} \right\rfloor$ branching (black) positions.*

Proof. Since an arbitrary symbol of Σ participates in a branching position with probability $\leq 1 - \frac{1}{k}$ then for a factor with λ branching positions it holds:

$$(1 - \frac{1}{k})^{\lambda} \geq \frac{1}{k}$$

We would like to bound λ:

$$\lambda \log(1 - \frac{1}{k}) \geq \log(\frac{1}{k}) \Rightarrow \lambda \leq \frac{\log(\frac{1}{k})}{\log(1 - \frac{1}{k})}$$

since $\log(1 - \frac{1}{k})$ is negative.

As a result, all possible factors with probability of occurrence $\geq \frac{1}{k}$ contain at most $\lambda = \left\lfloor \log_{\frac{k-1}{k}} \frac{1}{k} \right\rfloor$ branching positions.

Note that the base $\frac{k-1}{k}$ of the logarithm of λ is the maximum probability assigned to one of the symbols for the specific position. The next lemma shows how

the size of an arbitrary equivalence relation E_i is affected by a single branching position.

Lemma 3. *When a position in an arbitrary equivalence relation E_i is extended by one character that belongs in a branching position, then this position will participate in at most $|\Sigma|$ classes in the equivalence relation E_{i+1}.*

Proof. Assume position j in a class C_f^i of the equivalence relation E_i. This position is extended by a character that participates in a branching position. So, this position will participate in at most $|\Sigma|$ classes in the equivalence relation E_{i+1} since at most $|\Sigma|$ different symbols can participate with a non-zero probability in a branching position.

Both lemmas will be used to prove the following theorem that will be used extensively to prove the time complexities of the algorithms.

Theorem 1. *The size of the equivalence relation E_i, for all possible i, is always bounded by cn, where c is a constant.*

Proof. By Lemma 2, each position in the equivalence relation E_i, $1 \leq i \leq n$, has at most λ branching positions. By Lemma 3 each such branching position will result in at most $|\Sigma|$ different factors. Combining both lemmas we get that a factor with λ branching positions may participate in $|\Sigma|^\lambda$ classes instead of one.

As a result, the maximum size of each equivalence relation will be $\leq |\Sigma|^\lambda n$. Setting $c = |\Sigma|^\lambda$, the theorem follows.

Note that Theorem 1 provides a very rough estimate of the constant c. A more tight analysis will lead to a much smaller constant. Before stating the result of the paper we have to discuss the space complexity of the algorithm. Assuming

that $d = 2^i - 1$, for arbitrary i, we need to store all equivalence relations of size 2^j, for all $j \leq i - 1$ to report the repetitions of length d based on Lemma 1. As a result, the space complexity of the algorithm will be $O(n \log d)$. However, by computing in an online manner the equivalence relations we achieve linear space. That is, if we have computed equivalence relation E_{2^j} then we only need to store this relation (in order to compute the relation $E_{2^{j+1}}$) and the sum we need of all previously computed classes. For example, for length $2^i - 1$, with binary representation $0_i 1_{i-1} 1_{i-2} \ldots 1_2 1_1 1_0$, where the indices are powers of two, we immediately add to the sum the relations E_j where the j-th digit is one in the binary representation of d.

Theorem 2. *The above algorithm computes all repetitions of length d with probability of occurrence $\geq \frac{1}{k}$ in a weighted sequence X of length $|n|$ in $O(n \log d)$ time and linear space.*

Proof. Each step of the algorithm takes $O(n)$ time by Theorem 1. Since there are $O(\log d)$ steps, the complexity of the algorithm is $O(n \log d)$. The linear space is a result of the online computation of the equivalence relation E_d.

3.4 Experimental Results

In this paragraph we provide some experimental results for the computation of repetitions in weighted sequences using the previously defined algorithm. The algorithm was implemented in C++ using the Standard Template Library (STL), and run on a Pentium-4M 1.7GHz system, with 256MB of RAM, under the Red Hat Linux operating system (v9.0). The datasets used for testing the performance of the algorithm consisted of many copies of a small random weighted

sequence. We chose this repeated structure, rather than totally random files, in order to get a fair comparison of the running times.

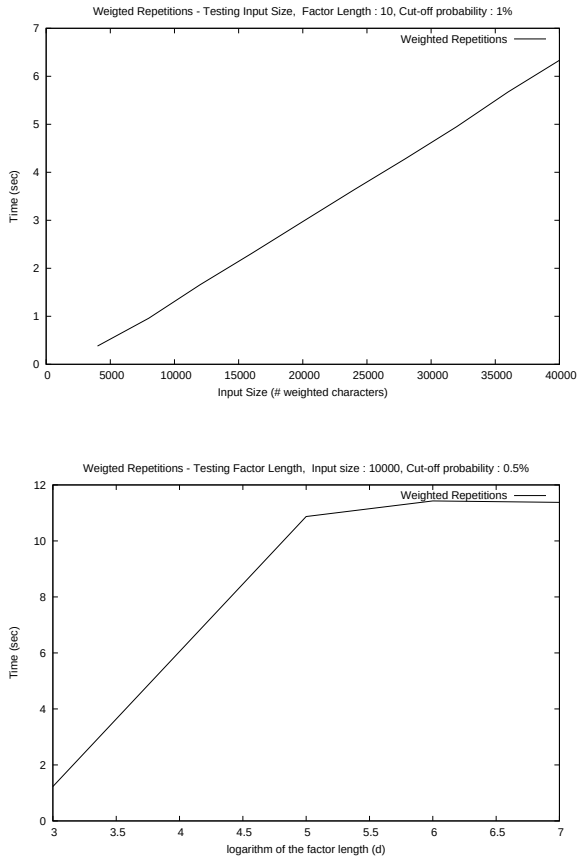

Fig. 3. Weighted Repetitions. Top: The running time with respect to the number of input (weighted) characters, n. Bottom: The running time with respect to the base-2 logarithm of the factor length ($\log_2 d$)

Figure 3 shows the running time of the algorithm for locating weighted repetitions, with respect to the input size, n, and the factor length, d. The former reveals that algorithm runs in linear time with respect to the input size ($O(n)$).

The latter shows a linear increase with respect to the logarithm of the factor length ($O(\log_2 d)$). Thus, the overall running time is $O(n \log d)$.

However, an interesting aspect of the algorithm is being revealed in the bottom graph: the running time tends to become constant for larger values of d. The reason is simple: as the length of the factor increases, there is a point at which the number of factors (with adequate probability) gets to zero (this happens when $d > 32$ in our example). After that point it only takes a singe pass over the vector holding the equivalence classes to realise that there are no factors at all (see $d = 64$ and $d = 128$ in our example).

4 Computation of Regularities

Regularities in a sequence 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. The length and number of repeated elements one wishes to be able to identify may be highly variable. In this section we extend KMR algorithm to compute covers on weighted sequences.

4.1 Computation of Covers

Covers in weighted sequences fall into two categories: **(1)** allow overlaps to pick different symbols from one single position and **(2)** factors that overlap choose the same symbols for overlapping regions. Notice that the first kind of covers allows border-less covers to overlap while the second does not.

For example, consider the weighted sequence

$$t = \begin{pmatrix} 0.3_A & 0.25_A & 0_A & 0.4_A & 0.8_A & 0_A \\ 0_C & 0.25_C & 1_C & 0.2_C & 0.05_C & 0.5_C \\ 0.2_G & 0.5_G & 0_G & 0.2_G & 0.1_G & 0_G \\ 0.5_T & 0_T & 0_T & 0.2_T & 0.05_T & 0.5_T \end{pmatrix} \tag{2}$$

Factor AC depicts the first case: AC occurs at positions 1, 2, 4, and 5. In this case, the occurrence of AC at position 1 picks the letter C at position 2, while the occurrence at position 2 picks up the letter A for position 2. Same happens at the occurrences of AC at positions 4 and 5. (Assuming that probabilities suffice.)

To locate all the length-d covers we first use the algorithm described in Section 3.2 to find all the length-d repetitions. Then by spending $O(n)$ extra time (thus $O(n \log n)$ total time) we can find either type of covers :

Type (1). For every factor occurring at position 1 (there is a constant number of them), scan E_d for occurrences of the same factor; if the distance of consecutive occurrences is always $\leq d$ then a cover has been found.

Type (2). For every factor occurring at position 1, compute its border array ($O(d)$ time). Scan E_d, like in (1), only now reject occurrences that start at positions other than some border of the previous occurrence.

The computation of type (1) covers is straightforward. On the other hand, type (2) covers face a difficulty: how can the border array of a factor be computed, since factors are only represented by integers, and the actual factors (strings) are never stored? Obviously, there is no other way than storing the actual strings that correspond to the numbers that represent factors. The space complexity for this is $O(nd)$, since there are at most $O(n)$ factors of length d. The time complexity remains unaffected by the fact that the actual factors are identified and stored. For example, consider that we will combine the equivalence relations E_a and E_b to obtain E_{a+b}. The identification of a factor in E_{a+b} takes only constant time since it can be constructed by the concatenation of one factor from E_a with one factor from E_b.

4.2 Experimental Results

Similarly, the running time for locating the weighted covers is shown in Figure 4, with respect to the input size (top), and the factor length (bottom). The running time for both types of covers grows asymptotically in the same manner as that of weighted repetitions, $O(n \log d)$.

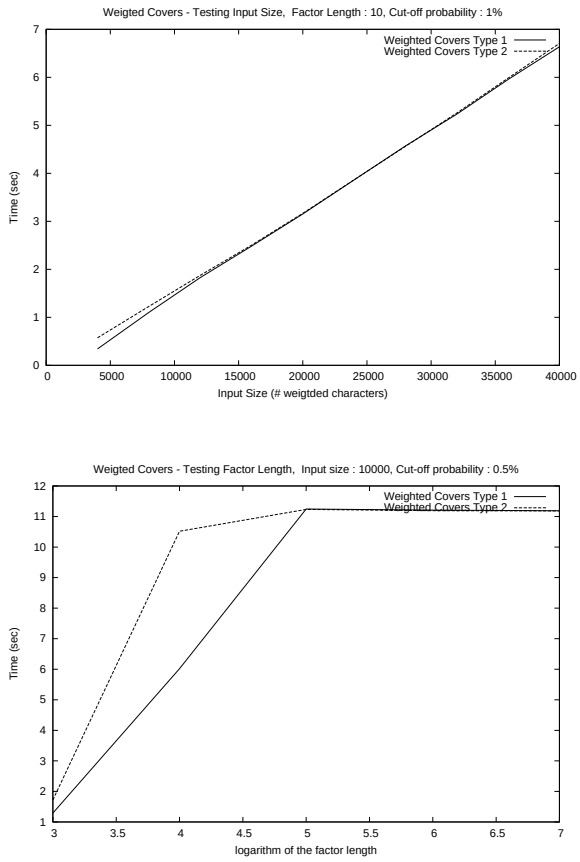

Fig. 4. Weighted Covers. Top: The running time with respect to the number of input (weighted) characters, n. Bottom: The running time with respect to the base-2 logarithm of the factor length ($\log_2 d$)

As expected, weighted covers of type (2) need more time to be computed since, in contrast with type (1) covers, a border array has to be constructed, and overlapping between consecutive occurrences of the same factor needs to be tested. Nevertheless, the asymptotic growth is still $O(n \log d)$.

5 Exact Pattern Matching

In this section we provide algorithms for the problem of exact pattern matching on weighted sequences. The problem we are going to tackle is the following:

Problem 1. Given a solid pattern $p = p[1 \dots m]$, a weighted text $t = t[1 \dots n]$ and an arbitrary constant $k \geq 1$, find all occurrences of p in t with matching probability $\geq \frac{1}{k}$.

The solution to the exact pattern matching problem depends on the matching measure (multiplicative or average) in use. In this section we present the solutions for both measures. In the first paragraph we presents how Algorithm `KMR` presented in 3.2 can be used for the exact pattern matching problem (under the multiplicative measure), while in the rest paragraphs we introduce alternative algorithms for the same problem.

5.1 Algorithm `KMR` for Pattern Matching in a Weighted Sequence

Assume that $m = |p|$. Then, this problem can be viewed as finding the E_m-class of repetitions in the input sequence $p\$X$, where $\$ \notin \Sigma$.

By using the algorithm described in Section 3 we can compute all repetitions of size m in $O((n+m)\log m)$ time. But we are only interested in repetitions (i, j) of the E_m-class such that $i = 1$ and $j > m + 1$. A simple traversal of all classes of E_m in time $O(n)$ will unveil all the desirable repetitions. The space complexity of the algorithm is reduced to linear by the online computation of the equivalence relations. A description of the algorithm follows.

pattern_match(p,X,k)

1 Construct string $s = p\$X$, where $\$ \notin \Sigma$

2 Scan s to construct the relation E_1

3 Use Lemma 1 to construct, successively, the relations $E_2, E_4, \ldots E_r$, $r = 2^{\lfloor log_2 d \rfloor}$

4 IF m is a power of 2 then end: $d = r$;

5 ELSE $d < 2r$, use Lemma 1 to construct E_d from E_r

6 Scan E_d and report pairs (i, j), where $i = 1$ and $j > m + 1$

END pattern_match

Given Theorem 2, the following theorem can be trivially proved:

Theorem 3. *The above algorithm computes the occurences of pattern p of length m in a weighted sequence X of length n in $O((n+m)\log m)$ time and linear space.*

5.2 Average Probability

In this paragraph we are interested in the solution of the pattern matching problem under the average probability measure. In more detail we are interested in finding all occurrences of p in t with average probability $\geq 1/k$, that is $P^i_{aver} > 1/k$. For a small constant k, the construction of the weighted suffix tree cannot be accomplished in linear time, as in [Iliopoulos et al. 2003a], because the number of factors of t is no longer linear. Moreover the use of Karp's algorithm as above is not efficient due to the size of the equivalent classes.

For instance, consider a text t where each position contains exactly two symbols, from an alphabet Σ, each of which has probability 0.5, e.g.

$$t = \begin{matrix} A\ A\ G\ T \cdots C \\ C\ G\ C\ A \cdots A \end{matrix}$$

Then, every possible substring of length m has an average probability 0.5 while the number of such substrings in t is $O(n2^m)$, which results in a suffix tree of

size $O(n2^m)$. Consequently, a small constant k does not allow us to adopt an approach similar to [Iliopoulos et al. 2003a].

An $O(n \log m)$-time algorithm, that works for arbitrary k, is possible based on the Fast Fourier Transform (FFT). First we find the number of matches between the pattern p and the text t for each position of t by using the FFT. Our approach is similar to the one used in [Gusfield 1997]

Let $M(t, p, i)$ be the number of characters of t and p that match when $p[1]$ is aligned with $t[i]$. We allow $p[1]$ to be aligned to the left of $t[1]$. Negative numbers specify the positions to the left of $t[1]$. As a result, the vector $M(t, p)$ stores all values of $M(t, p, i)$ for $-m + 1 \leq i \leq n$.

First, we break this problem into $|\Sigma|$ subproblems, one for each character of the alphabet Σ. Let $M_q(t, p, i)$ be the number of matches of character q when $p[1]$ is aligned to $t[i]$. The $(n + m)$-length vector $M_q(t, p)$ holds all these values. It is straightforward to see that:

$$M(t, p) = \sum_{\forall q \in \Sigma} M_q(t, p) \tag{3}$$

As a result, the problem is reduced to finding the match-count for each character. For each character q construct two bit vectors $\bar{p}_q$ and $\bar{t}_q$, where the i-th position is 1 if q occurs in $p[i]$ and $t[i]$ respectively with non-zero probability, otherwise it is 0. We pad the right end of $\bar{p}_q$ with n zeros and the right end of $\bar{t}_q$ with m zeros. This is necessary for the application of the FFT algorithm [Fischer et al. 1974]. By renumbering the indices of both bit vectors to run from 0 to $n + m - 1$, the number of matches for symbol q at position $t[i]$ is:

$$M_q(t, p, i) = \sum_{j=0}^{n+m-1} \bar{p}_q[j] \times \bar{t}_q[j + i] \tag{4}$$

where the indices are modulo $n + m$. The extra zeros were added so that when the right end of $\bar{p}_q$ is to the right of the end of t then no additional false matches

are counted. This is the cyclic correlation of $\bar{p}_q$ and $\bar{t}_q$, and $M_q(t,p)$ is computed by the FFT algorithm in $O(n \log m)$ operations. As a result the vector $M(t,p)$ by Equation 3 can be computed in $O(|\Sigma| n \log m)$ time. To this point, we have computed the occurrences of p in t without taking into account the probabilities. By scanning $M(t,p)$ we can report all positions that contain m in $O(m+n)$ time.

The problem now is to compute the average probability of each occurrence of p in t. A straightforward computation will lead to a time complexity of $O(mn)$. If the number of occurrences is up to $\frac{n \log m}{m}$ then we compute the average probability straightforwardly in $O(n \log m)$ time. However, if the number of occurrences is larger than $\frac{n \log m}{m}$ then we use the FFT algorithm again to compute the probabilities.

In the same manner as the construction of $\bar{p}_q$ and $\bar{t}_q$ we construct the vectors $\hat{p}_q$ and $\hat{t}_q$ containing the probabilities of occurrence of character q in each position. Then in the same way:

$$\hat{M}(t,p) = \sum_{\forall q \in \Sigma} \hat{M}_q(t,p) \tag{5}$$

where $\hat{M}_q(t,p)$ is the sum of the probabilities of occurrence of p in t for character q.

$$\hat{M}_q(t,p,i) = \sum_{j=0}^{n+m-1} \hat{p}_q[j] \times \hat{t}_q[j+i]$$

This is the cyclic correlation of two vectors and by using the FFT algorithm it can be computed in $O(n \log m)$ time. The computation of $\hat{M}(t,p)$ needs $O(|\Sigma| n \log m)$ time by Equation 5. As a result, by using vectors $M(t,p)$ and $\hat{M}(t,p)$ we can find all occurrences with average probability greater than $\frac{1}{k}$ by just locating positions i in $M(t,p)$ which contain m, and checking whether $\frac{\hat{M}(t,p,i)}{m} > \frac{1}{k}$.

5.3 Multiplicative Probability

In this paragraph we describe an alternative algorithm for the pattern matching problem using the multiplicative probability measure following the methodology presented in 5.2.

Assume that pattern p is a solid sequence and that k is arbitrary. Pattern p has an occurrence at position $t[i]$ with probability $P^i_{prod} = \prod_{j=1}^{m} \pi_{i+j-1}(p[j])$. Applying the logarithm we get:

$$\log(P^i_{prod}) = \sum_{j=1}^{m} \log(\pi_{i-m+j}(p[j]))$$

By this simple trick we got rid of the product and so we can apply the same $O(n \log m)$ algorithm described in 5.2 to find all occurrences. Note that this trick does not work when both the pattern and the text are weighted sequences. In this case, the only known algorithm is the straightforward $O(mn)$ dynamic programming algorithm.

5.4 Pattern Matching with Gaps

We consider the problem of pattern matching on weighted sequences by allowing gaps between occurrences of successive symbols of the pattern p in the text t.

Problem 2. Given a solid pattern $p = p[1 \dots m]$, a weighted text $t = t[1 \dots n]$, a constant $k \geq 1$ and a constant α, find all occurrences of p, allowing the existence of gaps between the occurrences of consecutive symbols of p, in t with probability larger than $1/k$. The gap g_i between the occurrences of $p[i]$ and $p[i+1]$ in t must satisfy: $|g_i| \leq \alpha$.

Example 6. Let $p = TGA$,

$$t = \begin{pmatrix} 0.3_A & 0.25_A & 0_A & 0.4_A & 0.8_A & 0_A \\ 0_C & 0.25_C & 1_C & 0.2_C & 0.05_C & 0.5_C \\ 0.2_G & 0.5_G & 0_G & 0.2_G & 0.1_G & 0_G \\ 0.5_T & 0_T & 0_T & 0.2_T & 0.05_T & 0.5_T \end{pmatrix}$$

$k = 10$, and $\alpha = 1$. Then p occurs in t at position 1, with 1-bounded gaps, since $\pi_1(T) = 0.5$, $\pi_2(G) = 0.5$, and $\pi_4(A) = 0.4$; thus: $P^4_{prod} = \pi_1 \cdot \pi_2 \cdot \pi_4 = 0.1 \geq \frac{1}{k}$. Note the gap (of size 1) between the second and the third symbols of p inside t.

Since gaps are allowed between the occurrences of symbols of p, it is possible that more than one occurrences of p end in a single position i in t. In what follows we will compute for each position i of t only one occurrence of p, namely the occurrence of p at i with the maximum probability, given that this probability is larger than $1/k$.

α-bounded gaps The solution we provide is based on the dynamic programming approach. The basic idea of the algorithm is the computation of continuously increasing prefixes of pattern p in the weighted sequence t. Define the set of all non-empty prefixes of pattern p to be $\Pi(p)$. Formally, $\Pi(p) = \{p[1], p[1..2], p[1..3], \ldots, p[1..m]\}$. We denote by $\Delta(p)$ the set of positions ℓ in the sequence t such that there is an occurrence of p with α-bounded gaps that ends at position ℓ given that the probability of occurrence is greater than $\frac{1}{k}$. Note that when extending the prefix $p[1..i-1]$ to $p[1..i]$, due to an occurrence of character $p[i]$, we choose always the prefix $p[1..i-1]$ with the maximum probability.

Let D be an $(m+1) \times (n+1)$ matrix. Each $D(i,j)$, for $1 \leq i \leq m$ and $1 \leq j \leq n$, will indicate whether there is an occurrence of the prefix $p[1..i]$

ending at position $t[j]$, where the gaps are bounded by α and the probability of occurrence is larger than $1/k$. For this problem the base conditions are:

$$\text{(a) } D(i,0) = 0,\ 0 \leq i \leq m$$

$$\text{(b) } D(0,j) = j,\ 0 \leq j \leq n$$

The base condition $D(i,0) = 0$ is correct since there is no occurrence of prefixes of p in the empty string. The base condition $D(0,j) = j$ is also correct since the empty string is assumed to match each of the characters of t. Before defining the recurrence relation it is necessary to give some notation with respect to probabilities. Assume that we are currently working on cell $D(i,j)$. Attached to this cell is a list of occurrences of prefixes $p[1..i-1]$, of maximum size α (the gap size). By $P^{max}_{i-1,j}$ we denote the occurrence of the prefix $p[1..i-1]$ with the maximum probability (out of those occurrences of $p[1..i-1]$ that appear in the list of the cell $D(i,j)$). The recurrence relation for $D(i,j)$ (without considering the update of the lists) is as follows:

$$D(i,j) = \begin{cases} j & \text{if } (p[i] \text{ in } t[j]) \ \&(j - D(i-1,j-1) - 1 \leq \alpha) \ \& \\ & (D(i-1,j-1) > 0) \ \&(P^{max}_{i-1,j} \times \pi_j(p[i]) \geq \frac{1}{k}) \\ D(i,j-1) & \text{if (previous case does not hold) } \& \\ & (j - D(i,j-1) - 1 < \alpha) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

If $D(i,j) = j$, then there is a match between $t[j]$ and $p[i]$ while the prefix $p[1..i-1]$ may have an occurrence at $D(i-1,j-1)$ and the gap is $\leq \alpha$. In addition, the probability of occurrence for this prefix is $\geq \frac{1}{k}$. If $D(i,j) = D(i,j-1)$, then there is no match between $t[j]$ and $p[i]$ and thus we are not able to extend the

occurrence of prefix $p[1..i-1]$ to $p[1..i]$. As a result, $D(i,j) = D(i,j-1)$, as long as the gap invariant is satisfied. In every other case, $D(i,j) = 0$.

For each position the occurrence with the maximum probability (given that this probability is larger than $1/k$) is given in the m-th row. The only remaining detail are the lists of prefixes that must be maintained during the construction of the matrix. This list must support three operations: 1) insert a new element in the head of the list, 2) delete an element from the tail of the list and 3) find the maximum element among the elements in the list. By using a heap-ordered queue we can support all three operations in constant time. If we want to store the whole matrix so that after its computation we are able to trace it back for occurrences then each cell must have its own list. This means, that when moving from $D(i-1,j)$ to $D(i,j)$ we have to copy the whole list and maybe make changes to its head and tail. This means that the time complexity of the algorithm will be $O(mn\alpha)$ and the same goes for the space complexity.

However, based on the facts that the matrix D is computed column by column and that the only difference between two adjacent lists are only in their head and tail we can reduce the time complexity to $O(mn)$. We use a simplified version of the persistent lists described in [Kaplan et al. 2000]. These lists support the operations of removing an element from the tail of a list, adding an element to its head, identifying the maximum element and copying the list (in fact it records the history of the structure with respect to update operations) in constant amortized time (for worst case constant time complexity refer to [Kaplan 1998]). This means, that over a sequence of n operations the total cost will be $O(n)$.

The algorithm presented in this section refers to the multiplicative probability measure. However, it can easily be adapted for the average probability by adding the maximum probabilities of the characters of the pattern instead of multiplying them. For unbounded gaps we can use the same algorithm by setting $\alpha = n - m + 1$.

6 Conclusions

In the following table the results described in this paper are given.

Problem	Time Complexity
Repetitions	$O(n \log m)$
Covers	$O(n \log m)$
Simple Pattern Matching	$O((n+m) \log m)$
Pattern Matching with Gaps	$O(mn)$

We presented algorithms on several problems on weighted sequences. These sequences seem to model various real life problems. Apart from the use of weighted sequences that was described in the introduction, they also appear in the field of event management for complex networks, where each event has a timestamp [Wang et al. 2003], as well as in DNA micro-array analysis, where expression levels of genes are recorded under different experimental conditions.

Weighted sequences are approximate by definition. However, this approximation measure is not the same as the usual distance metrics, like the Hamming distance or the edit distance. The probabilities in the weighted sequence provide a measure of our uncertainty concerning the data of the sequence. The error introduced by metrics like Hamming distance, provide a measure of the error that really exists between two sequences. As a result, it would be very interesting to design approximate pattern matching algorithms for weighted sequences.

References

[Brodal et al. 2003] Brodal, G.S., Lagogiannis, G., Makris, C., Tsakalidis, A., and Tsichlas, K. 2003. Optimal Finger Search Trees in the Pointer Machine. *Journal of Computer and System Sciences*, 67(2), 381–418.

[Brodal et al. 2000] Brodal, G.S., Lyngso, R., Pedersen, C., and Stoye, J. 2000. Finding Maximal Pairs with Bounded Gap. *Journal of Discrete Algorithms*, 1, 134-149.

[Brown et al. 1979] Brown, M.R., and Tarjan, R.E. 1979. A Fast Merging Algorithm. *Journal of the ACM*, 26(2), 211-226.

[Chazelle et al. 1986] Chazelle, B., and Guibas, L.J. 1986. Fractional Cascading: I. A data structuring technique. *Algorithmica*, 1, 133-162.

[Christodoulakis et al. 2004a] Christodoulakis, M., Iliopoulos, C., Mouchard, L., and Tsichlas, K. 2004. Pattern Matching on Weighted Sequences. *In the Proc. of Algorithms and Computational Methods for Biochemical and Evolutionary Networks (CompBionets).*

[Christodoulakis et al. 2004b] Christodoulakis, M., Iliopoulos, C., Perdikuri, K., and Tsichlas, K. 2004. Searching for Regularities in Weighted Sequences. *In the Proc. of International Conference of Computational Methods in Sciences and Engineering (ICCMSE).*

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

[Fischer et al. 1974] Fischer, M.J, and Paterson, M.S. 1974. String Matching and Other Products. Technical Report, Massachusetts Institute of Technology, Cambridge, MA.

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

[Iliopoulos et al. 2003a] Iliopoulos, C., Makris, C., Panagis, I., Perdikuri, K., Theodoridis, E., and Tsakalidis, A. 2003. Computing the Repetitions in a Weighted Se-

quence using Weighted Suffix Trees. *In Proc. of the European Conference On Computational Biology (ECCB).*

[Iliopoulos et al. 2004a] Iliopoulos, C., Makris, C., Panagis, I., Perdikuri, K., Theodoridis, E., and Tsakalidis, A. 2004. Efficient Algorithms for Handling Molecular Weighted Sequences. *In Proc. of the IFIP TCS.*

[Iliopoulos et al. 2002] Iliopoulos, C., Makris, C., Sioutas, S., Tsakalidis, A., and Tsichlas, K. 2002. Identifying occurrences of maximal pairs in multiple strings. *In Proc. of the 13th Ann. Symp. on Combinatorial Pattern Matching (CPM)*, 133-143.

[Iliopoulos et al. 2003b] Iliopoulos, C., Mouchard, L., Perdikuri, K., and Tsakalidis, A. 2003. Computing the repetitions in a weighted sequence. *In the Proc.of the Prague Stringology Conference (PSC)*, 91-98.

[Iliopoulos et al. 2004b] Iliopoulos, C., Perdikuri, K., Tsakalidis, A., and Tsichlas, K. 2004. The Pattern Matching Problem in Biological Weighted Sequences. *In Proc. of FUN with Algorithms*, 106-117.

[Kaplan 1998] Kaplan, H. 1998. *Purely Functional Lists.* PhD Thesis, Department of Computer Science, Princeton University.

[Kaplan et al. 2000] Kaplan, H., Okasaki, C., and Tarjan, R.E. 2000. Simple Confluently Persistent Catenable Lists. *SIAM Journal on Computing*, 30(3), 965–977.

[Karp et al. 1972] Karp, R., Miller, R., and Rosenberg, A. 1972. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. *In the Proc. of the Symposium on Theory of Computing (STOC).*

[Marsan et al. 2000] Marsan, L., and Sagot, M.-F. 2000. Algorithms for extracting structured motifs using a suffix tree with application to promoter and regulatory site consensus identification. *Journal of Computational Biology*, 7, 345-360.

[McCreight 1976] McCreight, E.M. 1976. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23, 262-272.

[Myers et al. 2000] Myers, E.W., and Celera Genomics Corporation. 2000. The whole-genome assembly of drosophila. *Science*, 287, 2196–2204.

[Pisanti et al. 2003] Pisanti, N., Crochemore, M., Grossi, R., and Sagot, M.-F. 2003. A basis of tiling motifs for generating repeated patterns and its complexity for higher quorum. *In Proc. of the 28th Symp. on Mathematical Foundations of Computer Science (MFCS)*, LNCS vol.2747, 622-632.

[Sagot 1998] Sagot, M.-F. 1998. Spelling approximate repeated or common motifs using a suffix tree.*In Proc. of the 3rd LATIN Symp.*, LNCS vol.1380, 111-127.

[Schieber et al. 1988] Schieber, B., and Vishkin, U. 1988. On Finding lowest common ancestors:simplifications and parallelization. *SIAM Journal on Computing*, 17, 1253-1262.

[Schneider et al. 1990] Schneider, T.D., and Stephens, R.M. 1990. Sequence Logos: A New Way to Display Consensus Sequences. *Nucleic Acids Res.*, 18, 6097–6100.

[Stormo 2000] Stormo, G.D. 2000. DNA binding sites: representation and discovery. *Bioinformatics*, 16(1), 16–23.

[Thompson et al. 1994] Thompson, J.D, Higgins, D.G, and Gibson, T.J. 1994. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, 22, 4673–4680.

[Venter et al. 2001] Venter, J.C., and Celera Genomics Corporation. 2001. The sequence of the human genome. *Science*, 291, 1304–1351.

[Wang et al. 2003] Wang, H., Perng, C.S., Fan, W., Park, S., and Yu, P.S. 2003. Indexing Weighted-Sequences in Large Databases. *In the Proc. of the 19th International Conference on Data Engineering (ICDE).*