VSP International
Science Publishers
P.O. Box 346, 3700 AH Zeist
The Netherlands

Lecture Series on Computer and Computational Sciences
Volume X, 2004, pp. 1-4

Searching for Regularities in Weighted Sequences

M. Christodoulakis, C. Iliopoulos, K. Tsichlas
Department of Computer Science, King's College, Strand, London WC2R 2LS, England
{manolis,csi,kostas}@dcs.kcl.ac.uk

K. Perdikuri
Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece
perdikur@ceid.upatras.gr

Abstract: In this paper we describe algorithms for finding regularities in weighted sequences. A weighted sequence is a sequence of symbols drawn from an alphabet Σ that have a prespecified probability of occurrence. We show that known algorithms for finding repeats in solid sequences may fail to do so for weighted sequences. In particular, we show that Crochemore's algorithm for finding repetitions cannot be applied in the case of weighted sequences. However, one can use Karp's algorithm to identify repeats of specific length. We also extend this algorithm to identify the covers of a weighted sequence. Finally, the implementation of Karp's algorithm brings up some very interesting issues.

1 Introduction

Weighted sequences are used for representing relatively short sequences such as binding sites as well as long sequences such as profiles of protein families (see [2], 14.3). In addition, they are also used to represent complete chromosome sequences ([2], 16.15.3) that have been obtained using a whole-genome shotgun strategy 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.

By keeping all the information the whole-genome shotgun produces, 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.

In this paper we present efficient algorithms for finding repetitions and covers in a weighted sequence. In solid sequences the algorithms of Crochemore [1] and Karp [5] are well known and have a $O(n \log n)$ time complexity. Their difference is that the first algorithm computes repetitions of all possible lengths while the second can compute repetitions of prespecified length.

There was already an attempt [4] to apply Crochemore's algorithm to weighted sequences. However, as we show in this paper the algorithm fails to find repetitions in $O(n \log n)$ time. In fact it needs $O(n^2)$ time to be able to compute all repetitions. However, Karp's algorithm has already been applied for this problem [3] successfully. In this paper we extend this algorithm to compute covers on weighted strings while we experimentally investigate its efficiency.

The structure of the paper is as follows. In Section 2 we give the basic definitions to be used in the rest of the paper. In Section 3 we argue why Crochemore's algorithm is not suitable for weighted sequences while in Section 4 we sketch the algorithm for finding repetitions and covers based on Karp's algorithm. Finally, in Section 5 we provide experimental results.

2 Preliminaries

In this work we concentrate on the identification of repetitions and covers of fixed length in a weighted biological sequence with probability of appearance $\geq 1/k$, where k is a small fixed constant determined by biologists (for example $k \leq 10$). The size of k is chosen small in order to represent the restricted ambiguity in the appearance of several characters in a biological sequence.

Assume an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. A word 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. A factor f of length p is said to occur at position i in the word s if $f = s[i, \cdots i + p - 1]$. A word has a repetition when it has at least two equal factors. A repetition is a cover when each position of the word s belongs in this repetition. A weighted sequence is defined as follows:

Definition 1 *A weighted sequence $s = s_1 s_2 \cdots s_n$ is a set of couples $(q, \pi_i(q))$, where $\pi_i(q)$ is the occurrence probability of character $q \in \Sigma$ at position i. For all positions $1 \leq i \leq n$, $\sum_{q=1}^{\sigma} \pi_i(q) = 1$.*

A factor is *valid* when its probability of occurrence is $\geq \frac{1}{k}$, where k is a small fixed constant. The probability of occurrence of a factor $f = f[1] \ldots f[m]$ occurring at position i in weighted sequence s is the product of probabilities of occurrence of the respective symbols of f in s, i.e. $\prod_{j=1}^{m} \pi_{i+j-1}(f[j])$. A weighted sequence has a repetition when it has at least two identical occurrences of a factor (weighted or not). In biological problems scientists are interested in discovering all the repetitions as well as covers of all possible words having a probability of appearance larger than a predefined constant. In the algorithms we provide we can always find the largest repetition or cover in a weighted sequence by a simple exponential and binary search on possible lengths. As a result we focus only for a prespecified length d.

3 Why Crochemore's Algorithm Fails for Weighted Sequences

This section assumes that the reader is familiar with the algorithm of Crochemore for finding repetitions [1]. The algorithm uses integers to represent factors. The E_i vector holds the factors of length i that start at each position of the text. The algorithm works in stages, each of which corresponds to the computation of repetitions of length larger by one with respect to the previous stage, while at the first stage repetitions of length 1 are computed. At each stage the algorithm chooses small classes to work on, without processing large classes. All classes that have not been processed in stage i, *implicitly* specify longer factors at stage $i + 1$. This is the crucial property of this algorithm that results in an $O(n \log n)$ time complexity.

The problem on weighted sequences is 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 [4] 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 and $o(n^2)$ algorithm.

4 Karp's algorithm

Karp's algorithm computes equivalence classes, similar to Crochemore's, but it computes them using $\log n$ steps of $O(n)$ time each. It has been successfully applied to weighted sequences [3]. The following lemma is the basic mechanism of Karp's 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).*

Based on Karp's algorithm, we will briefly sketch how it is applied to weighted sequences.

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

The equivalence class E_k is represented as a vector $v_1^{(k)}v_2^{(k)}\ldots v_{n-k+1}^{(k)}$ of sets of integers, where each set $v_i^{(k)}$ contains the labels of the equivalence classes of E_k to which each factor starting at position i belongs. The implementation of the algorithm is based on Equation (2) from Lemma 1 while its description in [3] was based on Equation (1). A detailed description of the algorithm follows. The new algorithm will be using $e_a + e_b$ pushdown stores, $P(1), \ldots, P(e_a)$ and $Q(1), \ldots, Q(e_b)$.

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.

4.1 Covers

Covers in weighted sequences fall into two categories: **(a)** allow overlaps to pick different symbols from one single and **(b)** 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.

Assume that we are interested only in length-d covers. This problem is solved in $O(n)$ extra time (thus $O(n \log n)$ total time) for either case:

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

2. Compute the border array of the candidate cover ($O(d)$ time). Scan E_d, like in (1), only now reject occurrences that start at positions other than some border of the previous occurrence.

Obviously we can do this for every class E_i that is computed on the way to create E_d. The computation of type (a) covers is straightforward. On the other hand, type (b) covers face a difficulty:

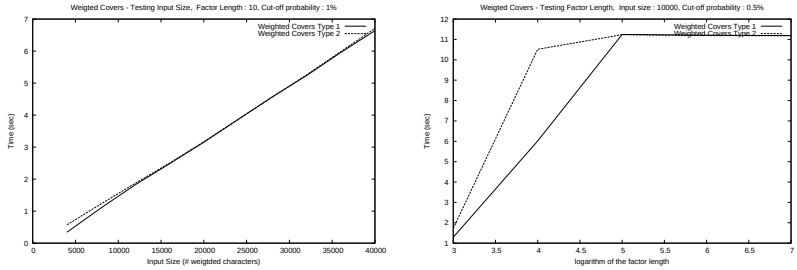

Figure 1: Weighted Covers. Left: The running time with respect to n. Right: The running time with respect to $\log_2 d$.

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.

5 Experimental Results

The algorithms were 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 our algorithms 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.

The running time for locating weighted covers is shown in Figure 1. As expected, weighted covers of type (b) need more time to be computed since, in contrast with type (a) 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 $O(n \log d)$. An interesting aspect of the algorithm is being revealed in the right 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.

References

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

[2] D. Gusfield. Algorithms on strings, trees, and sequences. Cambridge University Press, 1997.

[3] C.Iliopoulos, K. Perdikuri, A. Tsakalidis and K. Tsichlas. The Pattern Matching Problem in Biological Weighted Sequences. *In the Proc. of FUN with Algorithms*, 2004.

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

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