Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 395 (2008) 137–157
Theoretical Computer Science
www.elsevier.com/locate/tcs

Fast profile matching algorithms — A survey[☆]

Cinzia Pizzi[a,*,1], Esko Ukkonen[b]

[a] Department of Computer Science, University of Helsinki, Finland
[b] Helsinki Institute for Information Technology, University of Helsinki and Helsinki University of Technology, Finland

Abstract

Position-specific scoring matrices are a popular choice for modelling signals or motifs in biological sequences, both in DNA and protein contexts. A lot of effort has been dedicated to the definition of suitable scores and thresholds for increasing the specificity of the model and the sensitivity of the search. It is quite surprising that, until very recently, little attention has been paid to the actual process of finding the matches of the matrices in a set of sequences, once the score and the threshold have been fixed. In fact, most profile matching tools still rely on a simple sliding window approach to scan the input sequences. This can be a very time expensive routine when searching for hits of a large set of scoring matrices in a sequence database. In this paper we will give a survey of proposed approaches to speed up profile matching based on statistical significance, multipattern matching, filtering, indexing data structures, matrix partitioning, Fast Fourier Transform and data compression. These approaches improve the expected searching time of profile matching, thus leading to implementation of faster tools in practice.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Profile matching; Position-specific scoring matrix; PSSM; Algorithms; Computational complexity

1. Introduction

Many problems in bioinformatics can be modelled as a problem of *motif* search or discovery among a set of sequences, where by a motif we mean a signal responsible for functional or structural biological characterization. Because of the intrinsic variability in genomic sequences, the instances of a motif can be different in composition one from another. Initially, motifs were modelled as simple solid words, and this representation can still be valid in contexts where the solid words can be used as *anchors* to detect more complex patterns [3]. However, it soon rose the need of more flexible representations leading to the proposal of generalized kind of patterns [9,10], scoring matrices [21,26,47] and hidden Markov models [14].

In this paper we will focus on one of these models, the *scoring matrices*, that has been widely accepted as a good compromise between specificity and sensitivity in the motif representation. In literature, the scoring matrices are often referred to as *Position Specific Scoring Matrices* (PSSM), *Position Weight Matrices* (PWM), or *profiles*. There

☆ A work supported by the Academy of Finland grant 211496 and by the EU project Regulatory Genomics.
* Corresponding author.
E-mail addresses: cinzia.pizzi@dei.unipd.it (C. Pizzi), ukkonen@cs.helsinki.fi (E. Ukkonen).
1 Present address: Department of Information Engineering, University of Padova, Italy.

0304-3975/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.01.015

are in fact some differences in the definition of all these scoring matrices, but since it is possible to convert from one representation to another, in the following we will just refer to the general model as a scoring matrix.[2]

Scoring matrices have been used in several different scenarios over both DNA and protein datasets. For example, they have been proved to be a good model, with strong biological motivation, for transcription factor binding sites, when using as a score the information content of the signal [43]. Scoring matrices are widely used also for modelling protein families [21]. However, when modelling protein sequences it is often necessary to include variable length gaps into the model. Then *profile hidden Markov models* [14,15] are an efficient tool for representing motifs with gaps. Because our focus is on algorithms to search matrices, in the following discussion we will restrict our attention on the modelling of ungapped sequences.

A wide literature is available on topics related to scoring matrices. So far, most of the effort has been devoted specifically to two aspects: how to define the score to represent the instances of the motif such that the biological signal is properly modelled, and how to define a threshold to detect in the search only the true instances of the motif. It is undoubted that these two aspects play a crucial role in the definition and effectiveness of the model itself.

However, there is another aspect to consider. Once the scoring matrix is made available, and an appropriate threshold has been fixed, there is the need to find the matches of the matrix by scanning an input sequence or a sequence database. This problem can be referred to as *profile matching* as opposed to the classical *string pattern matching*. Most of the widely used algorithms described in the literature (e.g. [39,44,51]), use a brute-force sliding window approach, and the profile matching problem is still regarded as a not yet satisfactorily well-solved problem in computational biology [20]. In recent years a bunch of advanced algorithms based on score properties [53], indexing data structures [6,7,13], Fast Fourier Transform [41], data compression[17], matrix partitioning [33], filtering algorithms [7,33,38], pattern matching [38], and superalphabet [38] have been proposed to reduce the expected time of computation. The aim of this paper is to survey these methods to give the reader an overview of the state of the art of the topic and possibly stimulate future research in the field.

The outline of the paper is as follows. In Section 2 we will introduce the scoring matrix. Formal definitions of the model will be given, followed by discussion on choices for scores and thresholds. In Section 3 we will formally introduce the profile matching problem and discuss the advanced algorithms developed to solve it. Section 4 will summarize the characteristics of the algorithms and discuss some experiments with online algorithms. Then we will end up the paper with some remarks in the conclusion.

2. The scoring matrix model

Scoring matrices are often called alternately Position Specific Scoring Matrices, Weight Matrices or Profiles. However, different authors use these names to refer to slightly different definitions of the values within the scoring matrix. In the first part of the section we will discuss different type of matrices and try to organize in a clearer way the definitions that can be found in the literature (e.g. [25,40,48,53]). In the second part we will discuss threshold issues related to the problem of finding matches of scoring matrices in a sequence.

2.1. Building the model

Let $S = \{S_1, \ldots, S_N\}$ be a set of strings of length m that represent instances of a given motif (i.e. a biological signal) defined over an alphabet Σ. In biological sequence analysis the size of Σ is typically 4 (DNA) or 20 (amino acid sequences).

A *scoring matrix* is an $m \times |\Sigma|$ matrix M in which the generic position M_{ij} contains the value (an integer or real value) of the score given by the j-th symbol of the alphabet occurring at position i.

The simplest type of scoring matrix is the *count matrix* $C = (C_{ij})$. To obtain a count matrix we need first to align the instances S_k of the motif. Then for each position i we count the number of times that each symbol $s_j \in \Sigma$ appears in that position. This gives C_{ij}. The sum of the values in each row must sum up to N.

Rather than simple counts, we might want to have a probabilistic description of the signal that fits the instances set S. We call such a probabilistic model a *profile*. It gives occurrence probabilities for all alphabet symbols at

[2] It should be noted that there is another class of widely-used scoring matrices, also called *substitution matrices*, that gives generic and not position-specific scores for point mutations. PAM [12] and BLOSUM [23] matrices are examples in this class.

each position of the motif. For example, the maximum likelihood profile is estimated simply as $P = (P_{ij})$, where $P_{ij} = C_{ij}/N$. The sum of each row now sums up to 1. More advanced generation of a profile is a wide field of research in itself, with many different methods described in literature, see e.g. [21,24,28,45,49,54]. We omit their details as they are not relevant for understanding the profile matching algorithms we are surveying.

The count matrix and maximum likelihood profile for the sequences:

S_1:	A	C	G	A	C	T
S_2:	T	C	G	T	G	G
S_3:	A	C	G	T	C	C
S_4:	A	C	C	C	G	A

is shown in Table 1.

2.2. Derivation of scoring matrix from profile and background

The problem of profile matching can be described as the task of deciding whether a sequence window $w = w_1 w_2 \ldots w_m$ (also called a *segment*) of length m matches the signal described by an $m \times |\Sigma|$ profile P or it is a *background signal*.

The background is usually modelled as an i.i.d. model, that is an $m \times |\Sigma|$ profile π in which each row holds the same probability vector giving the background probability distribution of the alphabet symbols.

A match between a sequence and a matrix is decided upon the evaluation of a score that compares the probability to observe the segment in the signal model P and to observe it in the background:

$$\text{Score}(w) = \log \frac{Pr_P(w)}{Pr_\pi(w)} = \sum_{i=1}^{m} \log \frac{P_{i,w_i}}{\pi_{i,w_i}}.$$

This *log-odds score* is also called the *likelihood ratio* of the segments. Using logarithms facilitates avoiding underflow or overflow problems in numeric computations.

Once the background π is fixed, the profile can be translated in a *position-specific scoring matrix* (PSSM), which is also called a *position weight matrix*. The generic element of a PSSM is given by

$$S_{ij} = \log \frac{P_{ij}}{\pi_{ij}}.$$

The score of a segment w is obtained by adding the contributions of the entries in the PSSM that correspond to the symbols of the segment in each given position

$$\text{Score}(w) = \sum_{i=1}^{m} S_{i,w_i}.$$

Examples of a profile and the corresponding position specific scoring matrix, where the background probability is $p_s = 1/4$ for all s in Σ, are shown in Tables 2 and 3.

The occurrences of a signal described by a PSSM in a given sequence are traditionally defined as the sequence segments that have a score greater than or equal to a given threshold.

Table 1
Count matrix and profile estimated from the example sequence set

	A	C	G	T
1	3	0	0	1
2	0	4	0	0
3	0	1	3	0
4	1	1	0	2
5	0	2	2	0
6	1	1	1	1

	A	C	G	T
1	0.75	0	0	0.25
2	0	1	0	0
3	0	0.25	0.75	0
4	0.25	0.25	0	0.5
5	0	0.5	0.5	0
6	0.25	0.25	0.25	0.25

Table 2
An example profile

	A	C	G	T
1	0.65	0.05	0.05	0.25
2	0.05	0.85	0.05	0.05
3	0.05	0.25	0.65	0.05
4	0.25	0.25	0.05	0.45
5	0.05	0.45	0.45	0.05
6	0.25	0.25	0.25	0.25

Table 3
A position-specific scoring matrix

	A	C	G	T
1	0.96	−1.61	−1.61	0
2	−1.61	1.22	−1.61	0 −1.61
3	−1.61	0	0.96	−1.61
4	0	0	−1.61	0.59
5	−1.61	0.59	−1.61	−1.61
6	0	0	0	0

2.3. *Significance thresholding*

There are several aspects to consider when one has to select a suitable threshold for a search. When scanning a sequence to find the matches of the matrix we want to minimize two quantities:

(1) The number of false positives (FP), that is the number of sites reported as matches while they are not (also called *type-I errors*);
(2) the number false negatives (FN), that is the number of sites that are not reported as matches while they actually are (also called *type-II errors*).

The ability of a given threshold to limit the number of FP in the search is defined by its level of *significance*. On the other hand, when we talk about the ability of a given threshold to detect the true matches, hence to limit the number of FN, we refer to the *power* of the threshold. Despite the fact that it would be more accurate to take into consideration both measures to derive a sensitive threshold [40], the common practice is to rely on the level of significance only.

Moreover, the same threshold value is not good for different matrices as then the comparison of the search results becomes difficult. In fact, the scores depend on the matrix dimensions and the way in which they are scaled, so a threshold value that is a good discriminator for a matrix could be too low or too high for another.

A solution to this problem is to derive a significance threshold rather than using a “raw” threshold value. A common measure of significance is the E-value, i.e. the expected number of matches in a sequence with respect to a given background model. Another possibility, which we will elaborate more, is to use the standard approach of statistics that is based on p-values.

Low p-values allow for highly specific analysis in which only statistically very significant segments are retained and few false positives should occur. Higher p-values correspond to higher sensitive analysis, at a cost of decreased speed and possible inclusion of more FP.

The relationship between scores and p-values is called the *quantile function*. This returns, for a given p-value h, the corresponding threshold score T_h such that the m symbols are long strings whose score is $\geq T_h$ have total probability mass equal to h in the background model. More formally, let X be a random variable that represents the segment scores, and $f(x)$ be its mass probability function. The p-value is given by $G(T) = P\{X \geq T\} = \sum_{x=T}^{\infty} f(x)\mathrm{d}x$. The quantile function is $G^{-1}(p)$, i.e. for a given p-value p^*, the corresponding score is $T^* = \lceil G^{-1}(p^*) \rceil$.

To obtain the quantile function the probability mass function of the distribution must be computed. There are several approaches for this purpose. The straightforward method is to apply the scoring matrix to all sequences of length m and tabulate relative frequencies of score. However, the time complexity of this method is $O(m|\Sigma|^m)$.

If the values of the scoring matrix are integers within a fixed range of size R, a pseudo-polynomial time method, based on dynamic programming, computes the probability mass function recursively through each position of the scoring matrix, e.g. [40,46,53]. Initially, the entire probability mass score is assigned to the zero score. Then each

position row of the matrix is scanned, and the probability mass function is updated using the distribution from the previous row and the values of the current position scores. For a given score x one looks back at the probability mass function of the previous row to see which scores could have generated x. When computing the jth update, these are given by $x - M_{j,a}$ for all possible symbols a. This method works under i.i.d hypothesis and takes $O(m^2 R|\Sigma|)$ time. In [53], a variant to assume Markov model dependency is also described.

The computation of the entire probability mass function can be very expensive in terms of time. An alternative approach can be based on lazy evaluation [6,7]. With this method, only a partial distribution need to be computed. The probabilities for decreasing threshold values are summed up till the given p-value is exceeded . Depending on the p-value, this allows speedups between 3 (for p-value 1E-10) and 330 (for p-value 1E-40) with respect to the method described earlier.

3. Profile matching algorithms

With the exponential growth of both alignments blocks databases from which the scoring matrices can be synthesized (e.g. TRANSFAC [34], PRINTS [5], BLOCKS [27], JASPAR [42]) and sequences databases (e.g. Swiss-Prot [8,37]), fast matching between matrices and sequences is becoming a crucial requirement for nowadays tools. There are basically two scenarios to consider: (i) the query is a scoring matrix, the database is a sequence database; (ii) the query is a sequence, the database is a scoring matrices database.

The first scenario occurs, for example, when we want to find new instances of a protein family or a transcription-factor binding site that belong to the motif described by the matrix in a set of sequences. A typical application of the second scenario is the classification of a sequence with unknown function by comparison with known protein families described by matrices kept in an updated database. Similarly, given a promoter sequence, one could search for instances of known binding sites stored as scoring matrices in a database.

Algorithms designed for the first scenario are called *offline algorithms* since the sequence to scan is already available for preprocessing before the matrices are searched for. On the other end, algorithms designed for the second scenario are *online algorithms* that assume to have the sequence to scan available only at query time. Preprocessing of the matrices is anyway allowed to speed the search.

Despite the possible assignment of the query and the database roles to matrices and sequences, the common search procedure is to scan a sequence for segmental scoring according to the values of a matrix entries. Segments are then sorted according to decreasing score. Top scoring segments are considered occurrences of the motif in the sequence, or alternately in the other scenario, the sequence is likely to belong to the family described by the matrix.

Tools are currently available for both types of searches in both protein and DNA domains. BLIMPS [51], fingerPRINTScan [44], matinspector [39], PSI-BLAST [4], PoSSuMSearch [6], EMATRIX [53] are examples.

Most of these tools rely on the brute-force sliding window approach to evaluate segmental score for each possible starting position and then sort the segments according to the achieved score.

In this section we will concentrate on the task of scoring the segments of a sequence S of length n, according to a scoring matrix M of width m, where both the sequence and the matrix are defined over an alphabet Σ.

The problem of *profile matching* is defined as follows: given a sequence $S = s_1 s_2 \ldots s_n$, a scoring matrix $M_{m\times|\Sigma|} = (M_{ij})$, and a threshold T, find all the m symbols long segments $S_h = s_h s_{h+1}, \ldots, s_{h+m-1}$ in the sequence with a score above the threshold T, where the score of S_h is defined by

$$\mathrm{Score}(S_h) = \sum_{i=1}^{m} M_{i,s_{h+i-1}}. \tag{1}$$

As mentioned before, the brute-force approach consists of evaluating from (1) the score for all the segments S_h, that takes $O(nm)$ time. Successive sorting of the hits, that takes $O(n \log n)$ time, is needed if one wants to output the top-scoring segments first. An early attempt to achieve faster profile matching was implemented in the tool BLOCKSEARCH [18,19]. BLOCKSEARCH exploits the characteristic of the motif in the BLOCKS database that often have positions with one allowed (conserved) symbol, and it reports only segments with such conserved singleton symbols. This clearly allows one to avoid the scoring of many segments, but the possibility to miss many true positives is high. In fact, segments can potentially reach a high score even if they have a mismatch in these conserved positions. The presence of singleton positions in the BLOCK database derive from the way it was generated. The design of BLOCKSEARCH hence heavily depends on this characteristic of conservation in the BLOCKS database.

Table 4
Example of lookahead scoring

	A	C	G	T	T_i
1	0.65	0.05	0.05	0.25	**0.15**
2	0.05	0.85	0.05	0.05	**1.00**
3	0.05	0.25	0.65	0.05	**1.65**
4	0.25	0.25	0.05	0.45	**2.10**
5	0.05	0.45	0.45	0.05	**2.55**
6	0.25	0.25	0.25	0.25	**2.80**

The last column holds the values of the partial thresholds for each position

Algorithms that will be described in the rest of the section are not bound to a particular database or a specific way to build the alignment or the scoring matrix, and are therefore more general.

The first real improvements to the problem of profile matching come from a paper of Wu et al. [53]. In their paper three methods for accelerating the segmental scoring (1) are introduced, and they will be discussed in the next three sections.[3]

3.1. Significance filtering

Significance filtering simply means using the p-value based threshold of Section 2.3 in the search. Then the search will only report segments whose score is significantly high. This approach has several advantages: (i) it eliminates the need to store and sort a large number of potential hits; (ii) it reduces the number of false positives; (iii) it allows to compare hits across scoring matrices according to their statistical significance.

Significance filtering has a tradeoff between speed and sensitivity of the analysis. While the brute-force approach returns the value of the scores $\text{Score}(S_h)$ for all the segments S_h, significance filtering does not report any information about segments that have a score below the threshold. This characteristic is common also to the lookahead-based approaches that will be described in the next sections. On the other hand, the FFT and compression-based algorithms (Sections 3.9 and 3.10) provide information about all the segments in the sequences, as much as the brute-force algorithm does.

3.2. Sequential lookahead scoring

The lookahead technique (LA) is at the basis of several algorithms for profile matching. The idea behind it is simple and powerful. Depending on the matrix entries, there is a maximal score that can be added at every step of the score computation. For a segment S_h to be considered a match, its final score $\text{Score}(S_h)$ must be equal or greater than the given threshold T. This means that if the partial score reached at the step $m-1$ is less than $T - Sc_{\max}[m]$, where $Sc_{\max}[m]$ is the maximal value in the m-th row, then we do not need to perform the last comparison because the segment will never reach the threshold. This observation can be extended to all intermediate positions from 1 to m, computing the minimal threshold score T_i that has to be reached at the i-th comparison step. To do this we need to compute the maximum score that can be obtained from position $i+1$ to position m, for every index i:

$$Z^{(i)} = \sum_{k=i+1}^{m} \max_j M_{k,s_j}$$

where the summation is made over the maximal entries of the matrix in each row k. Then the intermediate threshold T_i is given by $T - Z^{(i)}$. If during comparison a segment is found to have a partial score at step i that is less than the intermediate threshold T_i, then the segment can be dropped, and the sliding window can be shifted to the next segment to evaluate.

As an example of how LA scoring works, consider Table 4, where the last column holds the partial thresholds for a final threshold 2.80.

[3] In the original paper the matrix has the roles of row and column inverted wrt our definitions.

Table 5
Example of permutated lookahead scoring

	A	C	G	T	T_i
2	0.05	0.85	0.05	0.05	**0.35**
1	0.65	0.05	0.05	0.25	**1.00**
3	0.05	0.25	0.65	0.05	**1.65**
4	0.25	0.25	0.05	0.45	**2.10**
5	0.05	0.45	0.45	0.05	**2.55**
6	0.25	0.25	0.25	0.25	**2.80**

When matching the string AGTAAC, at the second step the score will be equal to 0.70, below the partial threshold 1.00. The string is discarded without performing any other comparison, and the sliding window shifts to the next position.

Of course, in the worst case $O(mn)$ comparisons are still needed, but experiments based on real datasets proved that in practice many segments are dropped before the end of the window. The number of dropped segments depends on the scoring matrix and on the given threshold, that in turn depends on the given p-value. In the experiments cited in [53], LA performance ranged from 185.2 symbols/s for $p = 10^{-5}$ to 1019.2 symbols/s for $p = 10^{-40}$. On the same datasets, BLIMPS 3.2.5 computed 15.9 symbols/s, and BLOCKSEARCH 2.1 computed 109.0 symbols/s. However, when the partial thresholds have not enough discrimination power, then the extra cost of making a comparison after each character may lead to actual running times worse than those of the naive search.

3.3. Permutated matrix look-ahead

In the sequential lookahead technique, the intermediate thresholds are computed following the order of the matrix rows. However, the aim of lookahead is to drop a "bad" segment as soon as possible, and there is technically no restriction to compute the score in a different order. In fact, it is possible that a different arrangement of the matrix rows will lead to higher intermediate thresholds in the first steps of the comparison, thus leading to an earlier drop of the segment. For example, consider the scoring matrix of the previous section in which the columns and rows have been swapped. The matrix is reported in Table 5.

In this case the second position will be elaborated first. The second position of the string AGTAAC is a G, with score 0.05 below the partial threshold 0.35. The scoring will end at the first step, rather than the second as in the previous example.

In [53], the proposed permutation is the one that has the maximal difference between expected and maximal score:

$$E_i = \sum_a M_{i,a} b_a$$

$$M_i = \max_a M_{i,a}$$

where $b = (b_{a1}, b_{a2}, \dots b_{a|\Sigma|})$ is the vector of background frequencies that can be computed from a large sequence dataset. The permutation is obtained by sorting the rows with respect to the difference $M_i - E_i$ between maximal and expected scores.

Permutated LA scoring was on average 15.6% faster than sequential LA scoring in the experiments reported in [53].

3.4. Indexing techniques and LA

LA and Permutated LA techniques still rely on a sliding window approach, dropping segments whenever intermediate thresholds are not reached. EMATRIX, the tool that implements these approaches, has been developed for a context in which the query is a sequence and the database contains scoring matrices. This allowed the preprocessing of the scoring matrices database to obtain the quantile function to translate p-values in p-thresholds, so that the user just need to provide a p-value and the query sequence.

However, when the query is a scoring matrix, or a set of scoring matrices, and the database contains sequences, it is possible to achieve further speedup by exploiting the properties of indexing data structures such as suffix tree and suffix array, in combination with the LA technique. In this case the idea is to preprocess the input database or a single input sequence to build an index that allows to skip uninteresting parts that share a common prefix.

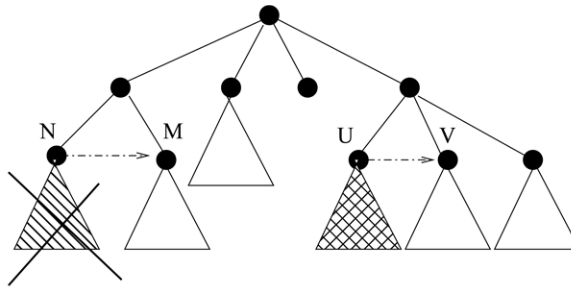


Fig. 1. An example of LA over a suffix tree. If the score at node N is below the partial threshold, then the subtree rooted at N can be skipped completely, and the search move to the next sibling M. If the score of node U is above the threshold T all the leaves that belong to the subtree rooted at U correspond to starting position of matching segments, even if the depth of U were less than m.

The first proposal of such an approach for profile matching can be found in [13] where suffix trees are used. More recently, a space efficient implementation on suffix arrays has been presented in [6,7].

LA with suffix trees

The suffix tree is a compact trie of the suffixes of an input string S, i.e. each leaf in the tree corresponds to a suffix of S. Compactness means that runs of nodes with one child are joined together in a single arc. This implies that every internal node must be a branching node, and as a consequence, since there are n leaves, there must be at most $n - 1$ internal nodes, and $O(n)$ nodes in total. To know whether a word w occurs in the input string S, we have to follow the path from the root labelled with the word characters. If the path exists then the words w occur in S. To know the set of positions in which the word occurs a simple visit of the subtree rooted at the node in which the path ends, or to the next following node if the path ends in the middle of an arc, is to answer the query. In fact, words ending in the middle of an arc share the same starting positions. Word occurrence queries can be answered on a suffix tree in linear time in the length of the word. Suffix trees can be constructed in linear time in the size of the input string [35,50,52].

In [13] LA is implemented over a suffix tree (ST). The scoring of the segments is done by a depth first traversal of the tree, calculating the score for the edge labels. The advantage of using such data structure is twofold, as illustrated in Fig. 1. When the partial score is below the intermediate threshold all the segments corresponding to positions in the subtree of the current node (or the following node if we are in the middle of an arc) can be dropped, and the search can continue from the next sibling. Moreover, if at some point the threshold is reached, all the segments corresponding to the leaves of the subtree of the node (or of the following node if we are in the middle of an arc) are matching segments. The second kind of speedup can be applied directly to matrices with nonnegative elements. For matrices with negative elements it suffices to add the absolute value of the negative minimal element to shift the range of the matrix value in a positive interval, and then apply the second kind of speedup. However, if the user requires to output the exact score with respect to a segment of length m, in case of matching segments it will be necessary to go down the tree until depth m is reached.

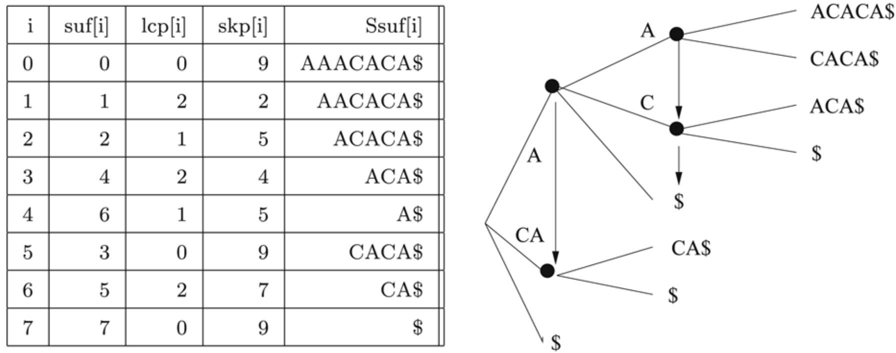
In this way many segments can be ignored or accepted at once, and speedup is easily achieved. Experiments on real datasets described in [13] showed that LA implemented over suffix tree allowed for a speedup factor of between 2.0 and 5.0 over sequential LA and between 2.7 and 10.6 over the brute-force algorithm.

The drawback of using the suffix tree data structure is that it is expensive to store. Each node can have up to $|\Sigma|$ children. If a fixed-size array is used to store the children the space needed to index large set of sequences, which is a typical situation in nowadays biological applications, could be prohibitive. Using a linked list might partially alleviate the problem. The implementation of the algorithm in [13] took $17n$ bytes. A more space-efficient suffix-tree implementation of [31] takes $12n$ bytes.

LA with suffix arrays

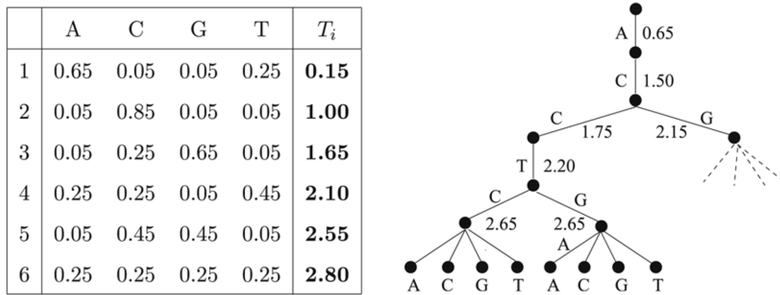
A much more space-efficient solution to speedup profile matching comes from enhanced suffix arrays and is implemented in the PoSSumSearch suite [6,7].

Enhanced suffix arrays (ESA) are as powerful as suffix trees [1] but for profile searching require only $9n$ bytes. Moreover, enhanced suffix arrays can be stored in a file and load into memory when needed. Another advantage over suffix tree is that when scanning the array from left to right, locality can be optimally exploited. The enhanced suffix



i	suf[i]	lcp[i]	skp[i]	Ssuf[i]
0	0	0	9	AAACACA$
1	1	2	2	AACACA$
2	2	1	5	ACACA$
3	4	2	4	ACA$
4	6	1	5	A$
5	3	0	9	CACA$
6	5	2	7	CA$
7	7	0	9	$

Fig. 2. An example of enhanced suffix array and the corresponding suffix tree for the same sequence S = AAACACA$. It can be noted that skipping SA ranges corresponds to skipping to the next simpling in the ST.



	A	C	G	T	T_i
1	0.65	0.05	0.05	0.25	**0.15**
2	0.05	0.85	0.05	0.05	**1.00**
3	0.05	0.25	0.65	0.05	**1.65**
4	0.25	0.25	0.05	0.45	**2.10**
5	0.05	0.45	0.45	0.05	**2.55**
6	0.25	0.25	0.25	0.25	**2.80**

Fig. 3. An example of Aho–Corasick Expansion driven by LA. At each level only the symbols that have a score above the partial threshold are considered for path expansion.

array for profile matching consists of three tables: the classical suffix array tables *suf* (sorted suffixes) and *lcp* (longest common prefix), and a further table *skp* (skip). The suffix array can be computed in linear time [29,30,32], and once the *suf* and *lcp* tables are known, the *skp* table can also be computed in linear time. To perform profile matching as it was done with the suffix tree, one can simulate a depth first traversal of the tree by processing *suf* and *lcp* from left to right [1]. To skip uninteresting ranges corresponding to segments with prefixes that are below the partial threshold the *skp* table is used (for details of computation see [6,7]). It is worth noting that skipping uninteresting ranges in the suffix array corresponds to skipping sibling nodes in the suffix tree, as shown in Fig. 2.

With respect to the implementation on the suffix tree the main gain in using the enhanced suffix array is space saving. Moreover, experiments on real biological datasets showed speedup factors between 4 and 50 times for DNA matrices and up to 1.8 for protein matrices with respect to other popular tools based on suffix tree LA and brute-force approach.

3.5. Multipattern matching based technique

A recent algorithm based on classic pattern matching is described in [38]. Given a matrix M and threshold T, the set of the m symbols long sequences with a score $\geq T$ is fixed. We may explicitly generate all such sequences in a preprocessing phase before the actual search over the input string S, and then use some multipattern search technique of exact string matching to find their occurrences fast from S. The Aho–Corasick multipattern search algorithm can be used here, hence the algorithm is called Aho–Corasick Expansion (ACE). To avoid the generation and checking of all Σ^m words of length m, the lookahead technique can be used to drive the direct generation of valid words only (see Fig. 3).

The set D of valid words is built by generating the prefixes of the sequences in Σ^m, and expanding a prefix that is still alive only if the expanded prefix is a prefix of some sequence that has score $\geq T$. This is repeated with increasing level j, until $j = m$.

For large T, only a small fraction of Σ^m will be included in D. In fact, the above procedure generates D in time $O(|D| + |M|)$ where the $O(|M|) = O(m|\Sigma|)$ time is needed for computing the lookahead bounds L_j from M.

The Aho–Corasick automaton $AC(D)$ for the sequences in D is then constructed [2,11], followed by a postprocessing phase to get the 'advanced' version of the automaton [36] in which the failure transitions have been eliminated. This version is the most efficient for small alphabets like that of DNA. The construction of this automaton needs time and space $O(|D||\Sigma|)$.

The search over S is then accomplished by scanning S with $AC(D)$. The scan will report the occurrences of the members of D in S.

The total running time of ACE is $O(|D||\Sigma| + |M|) = O(|\Sigma|^{m+1} + |M|)$ for preprocessing M to obtain $AC(D)$, and $O(|S|)$ for scanning S with $AC(D)$.

ACE performs very well for matrices up to length 13–14. In databases such as Jaspar they represent a large percentage of the total number of matrices. However, for larger matrices and smaller threshold values the size of the set D, and hence of the automaton, increases substantially, slowing down the performances. Moreover, for some small values of threshold and large matrix length the available memory might not be enough to store the entire complete automaton. In principle, the Aho–Corasick automaton can be built to contain strings coming from different matrices, allowing multiple matrices search at once. Unfortunately, by growing the size of the automaton, memory issues will probably slow down the search, so that the advantage to search more matrices at once would be lost.

3.6. Filtering techniques

Filtering techniques can also be used to speed up profile matching. An approximation of the score can be used for a fast check of the score achievable by a segment. This makes unnecessary to check completely all the segments.

Lookahead filtration algorithm

The lookahead filtration algorithm (LFA) of [38] can be seen as a speed-up version of the lookahead technique (LA) of Section 3.2. In this method, matrix M is preprocessed to get a finite-state automaton F that reports for each h symbols long segment v of S the prefix score of v, that is, the score that is given to v by the first h rows of M. The automaton F has a state for each sequence $u \in \Sigma^h$, and the prefix score of sequence u is associated with state for u. As the states and the transition function τ of F have a very regular structure, they can be implemented as an implicit data structure without an explicit representation of the transitions. The data structure is simply an array of size Σ^h, also denoted by F, whose entry $F(u)$ stores the prefix score of sequence u. Automaton F takes a state transition $\tau(u, a) = v$ by computing the new index v from u and a. This can be done by applying on u a shift operation followed by catenation with a.

The filtration phase is done by scanning with F the sequence S to obtain, in time $O(n)$, the prefix score of every h symbols long segment of S. Let t_i be the score for the segment that starts at s_i. Now, the upper bound used in filtration is $t_i + Z^{(h)}$ where $Z^{(h)}$ is the (also precomputed) lookahead bound for M at h. If $t_i + Z^{(h)} < T$, then the full occurrence of M at i must score less than T, and we can continue the search immediately to next location. If $t_i + Z^{(h)} \geq T$, the full score must be evaluated to see if it really is $\geq T$. This is done by adding to t_i the scores of matching the remaining positions $h + 1, \ldots, m$ of M against $s_{i+h}, \ldots, s_{i+m-1}$.

The filtration automaton F and the lookahead bounds can be constructed in time $O(|\Sigma|^h + |M|)$. Scanning S takes time $O(n)$ plus the time for checking the score at locations picked up by the filter. Checking takes $O(m - h)$ time per such a location. Denoting by r the number of locations to be checked, the total checking time becomes $O(r(m - h)) = O(n(m - h))$. Filtration pays off only if r is (much) smaller than $n = |S|$. Increasing T would obviously decrease r. Similarly, increasing h makes the upper bound tighter and hence decreases r.

Also the LFA algorithm can be easily adapted to multiple matrix search. Each table entry would simply be a vector with the scores corresponding to each different matrix to search.

Filtration by recoding on smaller alphabet

A drawback of some of the algorithms we have seen so far is their loss of performance when bigger alphabet, such as the protein alphabet, need to be used. In [7] a filtering method based on recoding in smaller alphabet is presented. It basically consists in a mapping of symbols of the alphabet $\Sigma = \{a_0, a_1, \ldots, a_k\}$ into symbols of alphabet $\Sigma' = \{b_0, b_1, \ldots, b_l\}$ by the means of a surjective function $\Phi : \Sigma \longrightarrow \Sigma'$ that maps symbol $a \in \Sigma$ to symbol $b \in \Sigma'$. $\Phi^{-1}(b)$ is the character class that corresponds to b. Both the sequence and the matrix need to be transformed according

to Φ:

$$S' = \Phi(s_1)\Phi(s_2)\cdots\Phi(s_n)$$
$$M'_{ib} = \max\{M_{ia} | a \in \Phi^{-1}(b)\}.$$

The matching segments will be those that in S' have a score with respect to M' that is above the same threshold T defined for M. Since the matches in the original sequence are a subset of matches in the transformed sequence, the latter can be used as a filter to detect positions that need to be checked for possible matches of M in S. In [7] the criteria followed to reduce the protein alphabet was driven by the correlation between amino acids defined by the BLOSUM [23] similarity matrix. Performance measures on reduced alphabet showed a speedup of two with respect to the 20 amino acid alphabet when the ESA algorithm was applied.

Filtering similar matrices

In [33] two algorithms are also presented to deal with the case of searching similar matrices. The idea is to exploit the similarity between matrices to design a filtering matrix that will tell if a matrix Mi of the set has a hit in correspondence of some position without computing the score for Mi explicitly. The hypothesis in their paper is that the matrices in the set are *consistent*, that is for each distinct matrix M, N, O, the best alignment between M and N and the best alignment between N and O, allow to deduce the best alignment between M and O. In this context, the optimal alignment between all matrices is simply computed by aligning all matrices with respect to a reference matrix. Examples of a filtering matrix N for a set of matrices $\{M1, M2, \ldots, Mk\}$ are given by:

- arithmetical mean: $N_{ix} = \sum_{j=1}^{k} Mj_{ix}$, $x \in \Sigma$, $i = 1$ to $|M|$
- minimum: $N_{ix} = \min\{M1_{ix}, M2_{ix}, \ldots, Mk_{ix}\}$, $x \in \Sigma$, $i = 1$ to $|M|$
- maximum: $N_{ix} = \max\{M1_{ix}, M2_{ix}, \ldots, Mk_{ix}\}$, $x \in \Sigma$, $i = 1$ to $|M|$.

The first algorithm for searching similar matrices is an exact algorithm. It initially computes the uncertainty threshold between two matrices M and N, defined as the smallest positive natural number x such that if the score of a segment u of length m with respect to N is less or equal to $\alpha - x$, then the score of u with respect to M is less or equal than α.

The uncertainty threshold $U(Mi, N)$ is computed for all matrices Mi in the set with respect to the filtering matrix N. Then the score of the filtering matrix N and the value of $U(Mi, N)$ is used to decide whether the score of Mi needs to be computed explicitly or not.

A lossy filtering algorithm was also presented in [33]. In this case only the score of the filtering matrix is computed. If an occurrence of N exists at position j an occurrence of each Mi is reported at the same position. Of course false positives and false negatives might be present in the output. The choice of the filtering matrix is critical in the determination of the sensitivity and selectivity of the process. However, since it is possible to compute the exact number of false positives and false negatives, the number of errors induced by the filtering can be taken under control.

3.7. Speed up by super-alphabet

It was observed in [38] that the naive brute force algorithm can be made faster by working in a "super-alphabet" instead of the original one (which is typically small in genomic applications). Each q-tuple of the original alphabet is a super-alphabet symbol, for some fixed q. Matrix M is preprocessed to obtain equivalent scoring matrix $\hat{M}$ for super-alphabet symbols: $\hat{M}$ is an $\lceil m/q \rceil \times |\Sigma|^q$ matrix whose entries are defined as

$$\hat{M}_{j,a_1 \ldots a_q} = \sum_{h=1}^{q} M_{(j-1)q+h,a_h}$$

for $j = 1, \ldots, \lceil m/q \rceil$ and for all q–tuples $a_1 \ldots a_q \in |\Sigma|^q$.

The score of each m symbols long segment of S can now be evaluated on $O(m/q)$ steps using shift-and-concatenate technique to find fast the appropriate entries of $\hat{M}$. The total running time of this naive super-alphabet algorithm (NS) becomes $O(nm/q)$, giving a (theoretical) speedup with respect to the naive algorithm by factor q, independently of the threshold T. In practice the larger overhead of algorithm NS makes the speedup smaller. Matrix $\hat{M}$ can be constructed in time $O(m|\Sigma|^q/q)$.

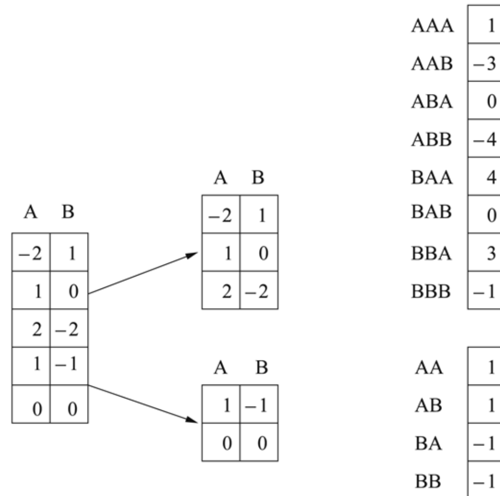


Fig. 4. An example of matrix partitioning in two submatrices, with the corresponding indexing tables.

3.8. Matrix partitioning

While the superalphabet technique NS divides the matrix M into constant-length slices, it is also possible to use variable length slices whose sizes depend on the matrix content [33]. In this algorithm, that we refer to as MP (Matrix Partitioning), the score is precomputed for words of length $i_{k+1} - i_k$ for $k = 1$ to n, where the i_k are the starting positions of the partitions. Note that the LFA algorithm also uses a precomputed version of M but the precomputation is restricted to only one fixed-length slice from the beginning of M.

The solution to the profile matching problem by the MP algorithm is exact. The matrix M is divided into a set of smaller matrices $\{M_1, \ldots, M_l\}$ that are made of contiguous slice of M. Then for each M_k the score for all words in $\Sigma^{i_{k+1}-i_k}$ is computed and store in a table T_k. The reason for considering slices of the matrix is that in such a way relatively small tables are needed to store the precomputed scores. If the scores were stored for all the $|\Sigma|^m$ words of length m, then the space needed would be prohibitive. Fig. 4 shows an example of matrix partitioning.

The score of a segment $u \in \Sigma^m$ is then computed by looking up at the table for the score of each substrings induced by partitioning u in the same segments interval as M:

$$\text{Score}(u, M) = \sum_{k=1}^{l} T_k(h(u[i_k, \ldots, i_{k+1} - 1]).$$

The problem of partitioning the matrix is characterized by two parameters: the number l of submatrices and the increasing sequence of positions $i_1 = 1, \ldots, i_k, \ldots, i_{l+1} = n + 1$. Given the matrix and the size of available memory, the objective is to choose the number and size of the subtables so that the average runtime is minimized. The average number of access for positions $[1 \ldots j]$ can be computed by a dynamic programming algorithm that uses the LA partial thresholds. The optimal partitioning can then be derived by backtracking the structure associated to the average number of access for all positions $[1 \ldots m]$.

The matrix partitioning technique can be easily extended to a set of matrices. In this case each entry of table T_k will be a vector of size equal to the number of matrices. Each cell of the vector will contain the score corresponding to one matrix.

The performance of this algorithm is naturally related to the structure of the matrix. Once the matrix is partitioned in k slices the time needed to compute the score of each segment in the input sequence is given by k look-ups and additions. Since there are $O(n)$ segments for which the score need to be computed the total complexity is $O(kn)$. Comparison with the naive approach NA on a database od DNA transcription factors taken from JASPAR [42] and TRANSFAC [34] databases showed a speed-up factor of 8.

3.9. The Fast Fourier Transform based technique

A completely different approach to profile matching is introduced in [41], where the Fast Fourier Transform (FFT) is used to compute the matching score of every possible alignment between a sequence and a scoring matrix.

It is an old observation that the FFT can be used in classical pattern matching [16]. The idea in [41] is to use the FFT in the same way that it is used to count the number of matches in an alignment between two sequences. The difference consists in replacing one of the sequences with a scoring matrix. Before introducing the algorithm, we will introduce some basic terminology associated with FFTs.

Let $\bar{x} = (x_0, x_1, \ldots x_{n-1})$ and $\bar{y} = (y_0, y_1, \ldots y_{n-1})$ be two real-valued vectors of length n.

Definition. The *cyclic cross-correlation* of $\bar{x}$ and $\bar{y}$ is a vector $\bar{z}_{cyc}(\bar{x}, \bar{y}) = (z_0, z_1, \ldots z_{n-1})$, such that

$$z_j = \sum_{k=0}^{n-1} x_{(j+k)} y_k$$

where the indexes in the expression are taken modulo n.

Definition. The *Discrete Fourier Transform* of a vector $\bar{x} = (x_0, x_1, \ldots x_{n-1})$ is the vector $\bar{X} = (X_0, X_1, \ldots X_{n-1})$, where

$$X_j = x_0 + x_1\omega^j + \cdots x_{n-1}\omega^{j(n-1)}$$

and the complex number $\omega = \exp^{(2\pi \mathrm{i})/n}$ is the nth root of unity. Vector $\bar{X}$ can be computed from $\bar{x}$ using the Fast Fourier Transform (FFT) algorithm, denoted $\bar{X} = \mathrm{FFT}(\bar{x})$. The running time is $O(n \log n)$.

If we indicate with $\bar{y}_R = (y_{n-1}, y_{n-2}, \ldots y_0)$ the *reverse* of the vector $\bar{y} = (y_0, y_1, \ldots y_{n-1})$, and with $\bar{x} \bigodot \bar{y} = (x_0 y_0, x_1 y_1, \ldots x_{n-1} y_{n-1})$ the dot product of the two vectors, we can compute the cyclic cross-correlation through the FFT in time $O(n \log n)$ as follows:

(1) compute $\bar{X} = \mathrm{FFT}(\bar{x})$ and $\bar{Y}_R = \mathrm{FFT}(\bar{y}_R)$
(2) compute the product $\bar{Z}_{cyc}(\bar{X}, \bar{Y}_R) = \bar{X} \bigodot \bar{Y}_R$
(3) compute the inverse of the FFT to obtain the cyclic cross-correlation $\bar{z}_{cyc}(x, y) = FFT^{-1}(Z_{cyc})$.

In profile matching linear alignment is more common than cyclic alignment, hence we need to define the linear cross-correlation.

Definition. The *linear cross-correlation* of two vectors $\bar{x}$ and $\bar{y}$ of length l and m respectively, is a vector $\bar{z}_{lin}(\bar{x}, \bar{y}) = \bar{z}_{cyc}(\bar{x}', \bar{y}') = (z_0, z_1, \ldots z_{n-1})$, where $n = l + m - 1$ and the vector $\bar{x}'$ is obtained from $\bar{x}$ by padding it to the left with $m - 1$ zeros, and $\bar{y}'$ from $\bar{y}$ by padding it to the right with $l - 1$ zeros. The time required to compute the linear cross-correlation is $O(n \log n)$.

FFT can been used for match counts problems (see, for example, [22]). Given two sequences $A = A_0 \ldots A_{l-1}$ and $B = B_0 \ldots B_{m-1}$ defined over an alphabet Σ, for every pair of symbols (a, b) drawn from the alphabet we can compute the number of times that the pair occurs in each possible alignment between A and B. To do that two indicator vectors $\bar{x}_a$, associated with the symbol a and the sequence A, and $\bar{x}_b$, associated with the symbol b and the sequence B, are used. In an indicator vector for a symbol a the element at position i will be set to one if the corresponding symbol in the sequence is equal to a, and zero otherwise. The linear cross-correlation $\bar{z}_{ab}$ between $\bar{x}_a$ and $\bar{x}_b$ will give a vector in which the ith element give the count of the pair (a, b) in the ith alignment between the two sequences. For example, consider two sequences S_1 = CCACCGTCG of length $l = 9$, and S_2 = TCAGTA of length $m = 6$. To compute the number of times the pair (C, A) occurs in each alignment, we have to build the indicator vector for C in S_1 and A in S_2. These vectors are respectively: $\bar{x}_C = (1, 1, 0, 1, 1, 0, 0, 1, 0)$ and $\bar{x}_A = (0, 0, 1, 0, 0, 1)$. The first eight alignments between the two sequences are shown in Table 6.

For example, the seventh alignment in Table 6 corresponds to the following situation in terms of the original sequences:

S_1	C	C	A	C	C	G	T	C	G
S_2			T	C	A	G	T	A	

Table 6
The first eight alignments between $\bar{x}_C = (1, 1, 0, 1, 1, 0, 0, 1, 0)$ and $\bar{x}_A = (0, 0, 1, 0, 0, 1)$

S_1						C	C	A	C	C	G	T	C	G
$\bar{x}'_c$	0	0	0	0	0	1	1	0	1	1	0	0	1	0
$align_0$	0	0	1	0	0	**1**	0	0	0	0	0	0	0	0
$align_1$	0	0	0	1	0	0	**1**	0	0	0	0	0	0	0
$align_2$	0	0	0	0	1	0	0	1	0	0	0	0	0	0
$align_3$	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0	0
$align_4$	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0
$align_5$	0	0	0	0	0	0	0	1	0	0	1	0	0	0
$align_6$	0	0	0	0	0	0	0	0	1	0	0	1	0	0
$align_7$	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0

The number of occurrences of the pair (C, A) is given by the sum of the number of columns in which both vectors hold the value one. Hence, the cross-correlation vector is equal to $(1, 1, 0, 2, 2, 0, 1, 2, 0, 0, 1, 0, 0, 0)$, showing that the third, fourth and seventh alignments are the ones with the highest number of occurrences of the pair.

The interesting cross-correlations are between the same symbol. The sum of the cross-correlations over all the symbols of the alphabet will report the total number of matches for each possible alignment:

$$\bar{z} = \sum_{a \in \Sigma} \bar{z}_{aa}.$$

Let us consider the two sequences S_1 and S_2, and their seventh alignment defined above.

The cross-correlations of each symbol are equal to $\bar{z}_{AA} = 0$, $\bar{z}_{CC} = 1$, $\bar{z}_{GG} = 1$, and $\bar{z}_{TT} = 1$, hence the alignment will have $\sum_{a \in \Sigma} \bar{z}_{aa} = 3$ matches, and this will be the value for the element z_7 of the cross-correlation vector.

The computation for all possible alignments requires $O(|\Sigma| n \log n)$ time.

Profile matching between a scoring matrix M and a sequence $S = s_1 s_2 \ldots s_k$ can be derived as the match count between two sequences. In the jth alignment the score will be

$$z_j = \sum_k M_{k+m-j, s_k}$$

where the index k ranges from $\max\{0, j - (m-1)\}$ to $\min\{m-1, j\}$, and it can be seen as the weighted sum of the indicator vectors of each symbol a in Σ, where the weights are taken from the corresponding column in the scoring matrix.

More formally, let $\bar{x}_a = (I(a, s_1), I(a, s_2), \ldots I(a, s_k))$, where $I(a, s_i) = 1$ if the ith symbol in the sequence is equal to a and $I(a, s_i) = 0$ otherwise. Let $\bar{y}_a = \bar{M}_a = (M_{1,a}, M_{2,a}, \ldots, M_{m,a})$ be the column for symbol a in the scoring matrix M. The linear cross-correlation $\bar{z}_a(\bar{x}'_a, \bar{y}'_a) = (z_{a,0}, z_{a,1}, \ldots, z_{a,n-1})$ is the score due to symbol a. The ith coordinate is given by

$$z_{a,j} = \sum_k I(a, s_k) M_{k+m-j,a}.$$

The sum of $z_{a,j}$ taken over all symbols a in Σ gives z_j, the total score for the jth alignment. Hence, the score vector is given by $\bar{z} = \sum_{a \in \Sigma} \bar{z}_a$.

For each symbol in the alphabet one FFT is needed to obtain $\bar{Z}_a$ from $\bar{z}_a$, and a FFT inversion is needed to obtain the score vector $\bar{z}$ from $\bar{Z} = \sum_{a \in \Sigma} \bar{Z}_a$. The time complexity is then $O(|\Sigma| n \log n)$.

As an example, let us consider the alignment between the sequence S_1 and the transpose of the scoring matrix that is shown in Table 7. The last line shows the indicator vector for the symbol C.

The weighted sum for the symbol C is 1.35, and corresponds to $\bar{z}_{C,7}$, since we are considering the seventh alignment. Similarly we will have that $\bar{z}_{A,7} = 0.65$, $\bar{z}_{G,7} = 0.05$, and $\bar{z}_{T,7} = 0.45$. The sum of these terms gives $\bar{z}_7 = \sum_{a \in \Sigma} z_{a,7} = 2.50$, which is the score of this specific alignment.

Table 7
Alignment between the transpose scoring matrix and the sequence

S_1	C	C	A	C	C	G	T	C	G
A			0.65	0.05	0.05	0.25	0.05	0.25	
C			0.05	0.85	0.25	0.25	0.45	0.25	
G			0.05	0.05	0.65	0.05	0.45	0.25	
T			0.25	0.05	0.05	0.45	0.05	0.25	
$I(C, S_1)$	1	1	0	1	1	0	0	1	0

Table 8
Scoring matrix M and run-length encoding matrix M_{RLE}

	A	C	G	T
1	0.2	0.5	0.1	0.2
2	0.3	0.3	0.2	0.2
3	0.1	0.6	0.2	0.1
4	0.1	0.1	0.4	0.4
5	0.2	0.1	0.2	0.5

	A	C	G	T
1	0.9	1.6	1.1	1.4
2	0.7	1.1	1.0	1.2
3	0.4	0.8	0.8	1.0
4	0.3	0.2	0.6	0.9
5	0.2	0.1	0.2	0.5

3.10. Compression-based techniques

An alternative approach to speed up profile matching, based on sequence compression techniques such as run-length encoding (RLE) and Lempel-Ziv (LZ78) algorithms, was presented in [17]. The idea is to score the compressed sequence, thus reducing the time required for computation by a factor equal to the sequence compression ratio.

Profile matching with RLE

RLE encodes runs of the same symbol a with a pair (l, a), where l is the length of the run, i.e. the number of consecutive symbols a. For example, the sequence $S = acccgttta$ will be represented by the sequence of pairs $(1, a)$ $(3, c)$ $(1, g)$ $(3, t)$ $(1, a)$.

In order to use RLE to speed up profile matching it is necessary to compute in a preprocessing step a telescoping scoring matrix M_{RLE} where the generic entry

$$M_{\text{RLE}}[i][a] = \sum_{k=i}^{m} M_{i,a}$$

represents the contribution that will be added to the score if the part of the segment from position i to m is composed by a run of a symbol a. Once the matrix has been computed, it is possible to know the contribution of a run of a given symbol a starting in any position of the alignment. In fact, if a run of length l is occurring starting at position i, then the contribution of the run to the score is given by $M_{\text{RLE}}[i][a] - M_{\text{RLE}}[i + l][a]$.

For example, consider the scoring matrix M, and its corresponding M_{RLE} in Table 8.

The scoring of the first segment of the sequence $S =$ acccgttta with the brute-force sliding window approach will need a total of five steps. On the other hand, when scoring the compressed representation of S with the M_{RLE} matrix, the run of c of length three will be computed in one step $M_{\text{RLE}}[2, c] - M_{\text{RLE}}[2+3, c] = 1.0$. Hence, the total number of steps for scoring the first segment will drop to three. This is a very simple toy example, but it is clear that for longer runs or for longer-scoring matrices, the probability of having more runs is higher, and hence the saving will also be higher.

Since contribution of runs can be computed in constant time, if l_{avg} is the average length of a run, the number of steps required to score a segment of length m will be on average m/l_{avg}. By adopting this technique the score for all segments can then be computed in $O(nm/l_{\text{avg}})$ time. The preprocessing phase takes $O(m|\Sigma|)$ to build the new matrix and $O(n)$ to encode the input sequence.

Besides the asymptotic complexity, the actual number of operations needs some consideration. In fact, in the brute-force algorithm, only a table lookup and an addition to the partial score is done at each step. With the RLE scoring two lookups, a subtraction and an addition are needed at every step. Moreover, analysis of the composition of some biological sequences reported in [17] showed that the number of runs is close, even though smaller, to the length of the original sequences. Therefore, for sequences with low compression ratio the RLE approach could actually require a higher number of operations than the brute-force approach.

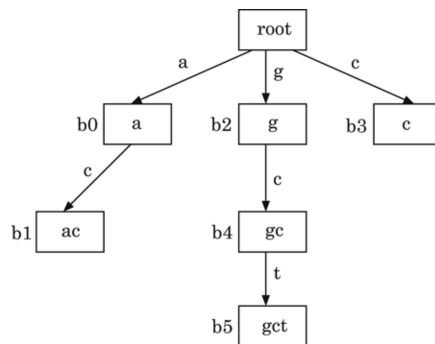


Fig. 5. LZ78 Tree for aacgcgcgct = b0 b1 b2 b3 b4 b5.

Profile matching with LZ78

LZ78 is a dictionary-based compression algorithm[55]. The original sequence is encoded in a sequence of codewords, each of which contains two kind of information: a link to the block corresponding to the longest matching dictionary entry previously found, and a first nonmatching symbol. The dictionary is built gradually by scanning the input sequence. The first character of the string is a codeword (block) itself, and it is added to the dictionary. The input sequence is then scanned, till the maximal matching entry (possibly no entry if the next symbol is scanned for the first time) in the dictionary is found. Then a new codeword is added to the dictionary and the substring will be encoded as a pair consisting of the pointer to the maximal match in the dictionary, and the newly added symbol. Since every block is built upon an already define block, the LZ78 encoding can be represented by a tree (see Fig. 5).

For profile matching, each node of the LZ78 tree needs to be enhanced with two kind of information: the length of the block, and an array that stores at index i the contribution that the block will give if it appears in the sliding window starting at position i. The contribution of each block can be computed by looking up at the array of scores of the father, adding the contribution of the newly added symbol of the current block, and updating its array of scores. Since each block takes constant time to be processed, and the average length of a block is $\log n$, the search time complexity is $O(nm/\log n)$.

In [17] asymptotic analysis, based on the number blocks in biological dataset, showed that LZ78 provides asymptotic speed up of about five times over the brute force algorithm.

4. Discussion and some experiments

Some characteristics of the algorithms described in the previous sections are summarized in Table 9.

The actual performance of these algorithms might indeed depend on implementation details, and on physical constraints such as the available memory. The algorithm to be used also depends on the availability for preprocessing of the sequence database or of the matrices before query time. In Section 4.1 we report some experimental results of the performance of some online algorithms on JASPAR [42] collection of DNA motifs.

For the offline algorithms the original papers describing the PoSSuMsearch suite [6,7] give a performance comparison of techniques that are based on suffix structures. Compared to suffix trees, the enhanced suffix array showed a speed up between 1.5 and 1.8 for proteins. Comparison with the online LA algorithm revealed that on the protein domain LA performed slightly better for large p-values, while for a p-value of 10^{-40} the speed up of the ESA is 1.5. In the DNA domain the speedup factor of the enhanced suffix array ranges from 17 to 196 depending on the value of the MSS threshold [39].

4.1. Experimental comparison of online algorithms

Here we report some experimental performance results for several online algorithms presented in Section 3. The experiments were performed using matrices from the JASPAR database, and a 50 M base long DNA sequence sampled from the human and the mouse genome. We compared the performances of the naive algorithm (NA), the Lookahead algorithm (LA), the Aho–Corasick based algorithm (ACE), the Lookahead Filtration Algorithm (LFA) and the Super-alphabet algorithm (NS). The software was written in C and C++, and it was run on a 3 GHz Intel Pentium processor with 2 GB of main memory, running under Linux.

Table 9
Algorithm summary

	Based on	Type	Time bound
NA	full scoring of all segments	online	$O(mn)$ search
LA	scoring interrupted by partial thresholds	online	$O(m\|\Sigma\|)$ preprocessing, $O(kn)$ search; $k = O(m)$ is the average depth of scoring
ACE	Aho–Corasick pattern matching	online	$O(\|D\|\|\Sigma\| + m\|\Sigma\|)$ preprocessing, $O(n)$ search; D is candidate sequences set, $\|D\| \leq \|\Sigma\|^m$
LFA	filtering by prefix scoring and partial thresholds	online	$O(\|\Sigma\|^h)$ preprocessing, $O(n + (m - h)r)$ search; h is the prefix length, r is average number of full scoring
SM	searching similar matrices by lossless and lossy filtering	online	$O(m\|\Sigma\|N)$ preprocessing for a set of N matrices; stringency of the reference matrix determines the search time for lossless filter and the number of false positives/negatives for lossy filter
NS	transformation into a super-alphabet	online	$O(m\|\Sigma\|^q/q)$ preprocessing, $O(mn/q)$ search; each q tuple of original symbols is a super-alphabet symbol
MP	partition of the matrix in slices of variable length	online	preprocessing with dynamic programming, $O(kn)$ search; $1 \leq k \leq m$
SA	filtering by using reduced alphabet	offline	$O(n + m\|\sigma\|)$ preprocessing to re-code sequence and matrix; speedup factor of 2 wrt ESA search with protein alphabet
ST	indexing by suffix tree	offline	$O(n + m\|\Sigma\|)$ preprocessing, depth–first search guided by LA
ESA	indexing by enhanced suffix array	offline	$O(n)$ preprocessing, $O(n + m + np)$ search; $0 \leq p \leq m$
DC	data compression	offline	$O(n)$ preprocessing, $O(nm/c)$ search ; c is the compression factor
FFT	Fast Fourier Transform	offline	$O(\|\Sigma\|(n + m)\log(n + m))$

Here n denotes the length and Σ the alphabet of the sequence, and m denotes the length of the position specific scoring matrix.

Table 10
Average running times per pattern (in seconds, preprocessing included) of different algorithms when searching for 96 JASPAR patterns of length $m \leq 13$, with varying p-values

	$p = 10^{-5}$	$p = 10^{-4}$	$p = 10^{-3}$	$p = 10^{-2}$
NA	1.033(0.248)	1.028(0.218)	1.109(0.364)	1.447(0.595)
LSA	0.579(0.231)	0.739(0.287)	1.113(0.517)	1.694(0.701)
ACE	**0.071(0.057)**	**0.113(0.073)**	**0.335(0.307)**	1.248(1.224)
LFA	0.339(0.041)	0.347(0.046)	0.446(0.306)	**0.903(0.595)**
NS	0.637(0.853)	0.667(0.153)	0.713(0.307)	1.089(0.539)

The standard deviation is given in parenthesis.

Table 11
Average running times per pattern (in seconds, preprocessing included) of different algorithms when searching for 27 JASPAR patterns of length $m \geq 14$, with varying p-values

	$p = 10^{-5}$	$p = 10^{-4}$	$p = 10^{-3}$	$p = 10^{-2}$
NA	1.779(0.343)	1.954(0.556)	1.954(0.559)	2.631(0.994)
LSA	1.304(0.391)	1.655(0.608)	2.085(0.751)	3.104(1.121)
LFA	**0.516(0.229)**	**0.802(0.576)**	1.109(0.644)	2.187(1.123)
NS	0.853(0.132)	1.003(0.491)	**1.050(0.518)**	**1.764(0.961)**

The standard deviation is given in parenthesis.

Tables 10 and 11 report the summary of results for different p-values from 10^{-2} to 10^{-5}. We did not considered smaller p-values because the number of hits per matrix would be 0 in most cases. We separated the matrices into two groups: those of lengths up to 13 (96 matrices out of 123) and those of length from 14 to 30 (the remaining 27 matrices). This separation was done because for lengths above 14 in some other cases the 1 GB allocated for storing the complete ACE automaton would not be sufficient to store all candidate words, hence we did not report results for ACE in the Table 11.

We add the detailed matrix-per-matrix results in Figs. 6 and 7 for p-value 10^{-4}, and in Figs. 8 and 9 for p-value 10^{-3}.

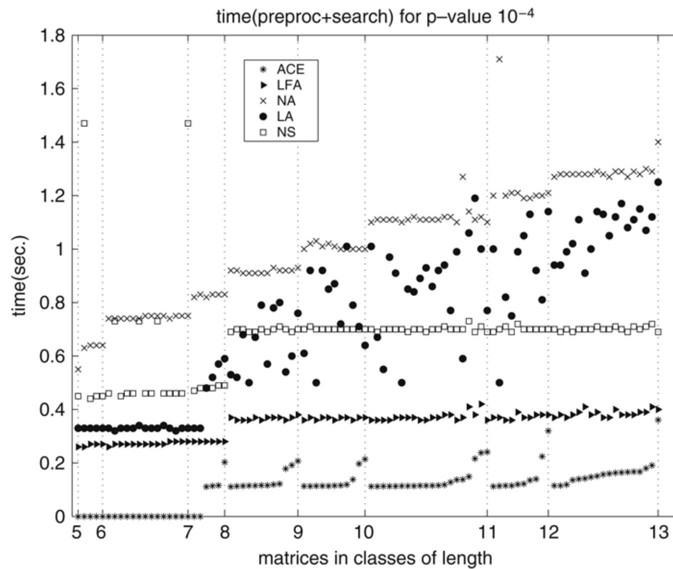


Fig. 6. Results for 96 matrices from JASPAR.

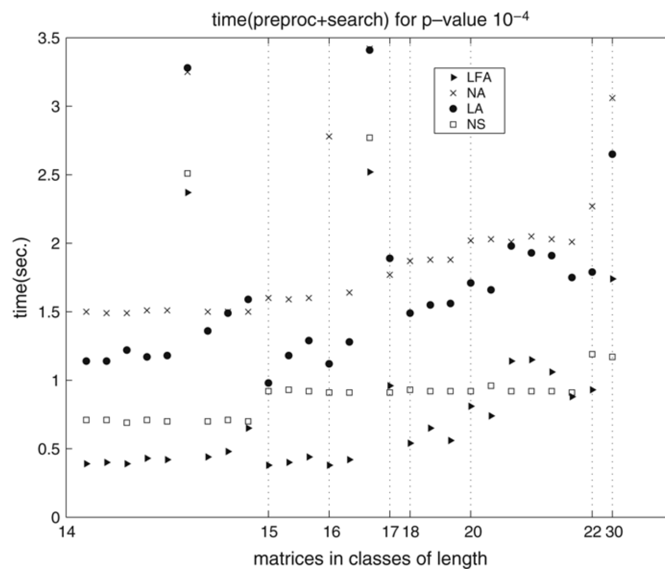


Fig. 7. Results for 27 matrices from JASPAR.

In Figs. 6 and 8 (lengths up to 13) we can notice that the best performing algorithm is ACE, followed from close by the LFA algorithm. Superalphabet(NS) also performed better than lookahead(LA) and naive(NA). For small p-values using ACE for relatively small matrices has the advantage that if no segment generated from the matrix will have a score above the corresponding threshold, ACE will detect immediately this condition when the automaton is empty. This means that no further scanning of the sequence is needed. On the other end, by increasing the p-value the performance of ACE decreases as it was explained before.

In Figs. 7 and 9 the best performing algorithm is LFA. Only for very long matrices the superalphabet outperforms LFA, that anyway can be seen as the more robust algorithm overall.

In [38] a hybrid algorithm (LFACE) was also proposed to combine the ACE and LFA algorithms by using a rule of thumb with respect to the expected size of the automaton to select one algorithm or the other. This combined

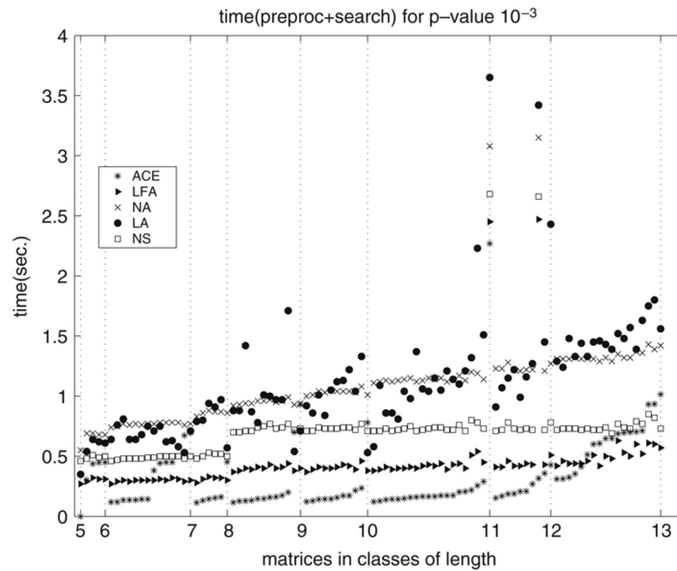


Fig. 8. Results for 27 matrices from JASPAR.

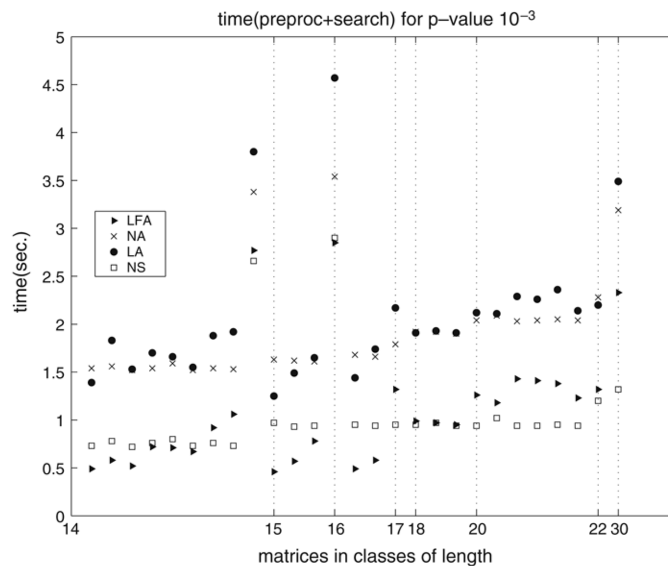


Fig. 9. Results for 96 matrices from JASPAR.

algorithm performs better than the others at basically all lengths and p-values. Some exception can occur for long matrices where superalphabet might be faster, as can be seen in Table 12 that refers to the complete JASPAR database.

It is worth noting also that sometimes the overhead of checking the partial threshold with lookahead can lead to worse performances than the naive algorithm. Moreover, some matrices required a much longer time than other matrices in the same class of lengths with any of the tested algorithms. These intrinsically difficult matrices correspond to the outliers visible in the graphs.

5. Conclusions

Developing good algorithms for profile matching has become more important in recent years, driven by biological applications. Scoring matrices are the model of choice for several applications in both DNA and proteins domains, and

Table 12
Average running times per pattern (in seconds, preprocessing included) of different algorithms when searching for all JASPAR patterns, with varying p-values

	$p = 10^{-5}$	$p = 10^{-4}$	$p = 10^{-3}$	$p = 10^{-2}$
NA	1.197(41.8)	1.232(40.7)	1.283(39.0)	1.708(29.3)
LSA	0.739(57.8)	0.941(53.2)	1.327(37.8)	2.004(25.0)
LFA	0.379(132)	0.448(112)	0.593(84.5)	**1.185(42.3)**
NS	0.686(73.0)	0.742(67.5)	0.788(63.6)	1.239(40.4)
LFACE	**0.147(340)**	**0.258(194)**	**0.505(99.3)**	1.230(40.7)

The average search speed in megabases/s is given in parenthesis.

the task of finding matches between matrices and sequences is a common underlying procedure in many bioinformatics tools.

The exponential growth of databases asks for more efficient algorithms for scoring sequence segments than the brute-force $O(mn)$ algorithm. Several algorithms have been proposed to achieve speedups in real application. Nonetheless, these algorithms present different interesting theoretical approaches to the profile matching problem. In this survey we explored approaches based on score properties, indexing data structures, filtering techniques, pattern matching, matrix partitioning, Fast Fourier Transform and data compression. We also reported some empirical performance results for online algorithms. A more extensive empirical comparison of online and offline methods is a suggested topic for future research in this area.

6. Acknowledgements

We would like to thank the referees for their careful reading of the paper, and their comments that we believe helped us to improve the presentation of the paper substantially.

References

[1] M. Abouelhoda, S. Kurtz, E. Ohlebush, Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms 2 (2004) 53–86.
[2] A.V. Aho, M. Corasick, Efficient string matching: An aid to bibliographic search, Communications of the ACM 18 (1975) 333–340.
[3] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, Basic local alignment search tool, Journal of Molecular Biology 215 (3) (1990) 403–410.
[4] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Anang, W. Miller, D.J. Lipman, Gapped BLAST and PSI-BLAST: A new generation of protein database search programs, Nucleic Acids Research 25 (1997) 3389–3402.
[5] T.K. Attwood, M.E. Beck, PRINTS — a protein motif finger-print database, Protein Engineering 7 (7) (1994) 841–848.
[6] M. Beckstette, D. Strothmann, R. Homann, R. Giegerich, S. Kurtz, PoSSuMsearch: Fast and sensitive matching of position specific scoring matrices using enhanced suffix arrays, in: Proc. of the German Conference in Bioinformatics, in: GI Lecture Notes in Informatics, P-53, 2004, pp. 53–64.
[7] M. Beckstette, D. Strothmann, R. Homann, R. Giegerich, S. Kurtz, Fast index based algoritms and software for matching position sepcific scoring matrices, BMC Bioinformatics 7 (1) (2006) 389.
[8] B. Boeckmann, A. Bairoch, R. Apweiler, M.C. Blatter, A. Estreicher, E. Gasteiger, M.J. Martin, K. Michoud, C. O'Donovan, I. Phan, S. Pilbout, M. Schneider, The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003, Nucleic Acids Research 31 (1) (2003) 365–370.
[9] A. Brazma, I. Jonassen, J. Vilo, E. Ukkonen, Pattern discovery in biosequences, in: Grammatical Inference (ICGI'98), in: Lecture Notes in Articial Intelligence, vol. 1433, Springer-Verlag, 1998, pp. 255–270.
[10] A. Califano, SPLASH: Structural pattern localization analysis by sequential histograms, Bioinformatics 16 (4) (2000) 341–357.
[11] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[12] M. Dayhoff, R. Schwartz, B. Orcutt, A model of evolutionary change in proteins, Atlas of Protein Sequence and Structure 15 (1978) 345–358.
[13] B. Dorohonceanu, C.G. Neville-Manning, Accelerating protein classification using suffix trees, in: Proc. of the 8th International Conference on Intelligent Systems for Molecular Biology, ISMB 2000, 2000, pp. 128–133.
[14] R. Durbin, S.R. Eddy, A. Krogh, G.J. Mitchison, Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic acids, Cambridge University Press, 1998.
[15] S.R. Eddy, Hidden Markov models, Current Opinion in Structural Biology 6 (1996) 361–365.
[16] M. Fischer, M. Paterson, String-matching and other products, complexity of computation, in: R.M. Karp (Ed.), Proc. SIAM-AMS, 1974, pp. 113–125.
[17] V. Freschi, A. Bogliolo, Using sequence compression to speedup probabilistic profile matching, Bioinformatics 21 (10) (2005) 2225–2229.
[18] R. Fuchs, Block search on VAX and alpha computer systems, CABIOS 9 (1993) 587–591.
[19] R. Fuchs, Fast protein block searches, CABIOS 10 (1994) 79–80.

[20] G.H. Gonnet, Some string matching problems from bioinformatics which still need better solutions, Journal of Discrete Algorithms 2 (1) (2004) 3–15.
[21] M. Gribskov, A.D. McLachlan, D. Eisenberg, Profile analysis: Detection of distantly related proteins, Proceedings of the National Academy of Sciences 84 (13) (1987) 4355–4358.
[22] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.
[23] J.G. Henikoff, S. Henikoff, Amino acid substitution matrices from protein blocks, PNAS 89 (1992) 10915–10919.
[24] J.G. Henikoff, S. Henikoff, Using substitution probabilities to improve position specific scoring matrices, Cabios 12 (2) (1996) 135–143.
[25] S. Henikoff, Scores for sequence searches and alignments, Current Opinion in Structural Biology 6 (1996) 353–360.
[26] S. Henikoff, J.C. Wallace, J.P. Brown, Finding protein similarities with nucleotide sequence databases, Methods in Enzymology 183 (1990) 111–132.
[27] J.G Henikoff, E.A. Greene, S. Pietrokovski, S. Henikoff, Increased coverage of protein families with the blocks database servers, Nucleic Acids Research 28 (1) (2000) 228–230.
[28] A. Kel, E. Gossling, I. Reuter, E. Cheremushkin, O. Kel-Margoulis, E. Wingender, Match: A tool for searching transcription factor binding sites in DNA sequences, Nucleic Acid Research 31 (13) (2003) 3576–3579.
[29] D.K. Kim, J.S. Sim, H. Park, K. Park, Linear-time construction of suffix arrays, in: 14th Annual Symposium on Combinatorial Pattern Matching, CPM 2003, in: LNCS, vol. 2676, Springer-Verlag, 2003, pp. 186–199.
[30] P. Ko, A. Aluru, Space efficient linear time construction of suffix arrays, in: Combinatorial Pattern Matching, CPM 2003, in: LNCS, vol. 2676, Springer-Verlag, 2003, pp. 203–210.
[31] S. Kurtz, Reducing the space requirements of suffix trees, Software - Practice and Experience 29 (13) (1999) 1149–1171.
[32] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. 30th International Colloquium on Automata, Languages and Programming, ICALP '03, in: LNCS, vol. 2719, Springer, 2003, pp. 943–955.
[33] A. Liefooghe, H. Touzet, J.-S. Varré, Large scale matching for position weight matrices, in: Combinatorial Pattern Matching, CPM 2006, in: LNCS, vol. 4009, Springer, 2006, pp. 401–412.
[34] V. Matys, E. Fricke, R. Geffers, E. Gossling, M. Haubrock, R. Hehl, K. Hornischer, D. Karas, A.E. Kel, O.V. Kel-Margoulis, D.U. Kloos, S. Land, B. Lewicki-Potapov, H. Michael, R. Munch, I. Reuter, S. Rotert, H. Saxel, M. Scheer, S. Thiele, E. Wingender, TRANSFAC: Transcriptional regulation, from patterns to profiles, Nucleic Acids Research 31 (1) (2003) 374–378.
[35] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM 23 (2) (1976) 262–272.
[36] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings, Cambridge University Press, 2002.
[37] C. O'Donovan, M.J. Martin, A. Gattiker, E. Gasteiger, A. Bairoch, R. Apweiler, High-quality protein knowledge resource: SWISS-PROT and TrEMBL, Brief Bioinformation 3 (3) (2002) 275–284.
[38] C. Pizzi, P. Rastas, E. Ukkonen, Fast search algorithms for position specific scoring matrices, in: Bioinformatics Reserch and Development Conference, BIRD 2007, in: LNBI, vol. 4414, Springer, 2007, pp. 239–250.
[39] K. Quandt, K. Frech, H. Karas, E. Wingender, T. Werner, MatInd and MatInspector: New fast and versatile tools for detection of consensus matches in nucleotide sequences data, Nucleic Acid Research 23 (23) (1995) 4878–4884.
[40] S. Rahmann, T. Müller, M. Vingron, On the power of profiles for transcription factor binding site detection, Statistical Applications in Genetics and Molecular Biology 2 (1) (2003) article 7.
[41] S. Rajasekaran, X. Jin, J.L. Spouge, The efficient computation of position-specific match scores with the Fast Fourier Transform, Journal of Computational Biology 9 (1) (2002) 23–33.
[42] A. Sandelin, W. Alkema, P. Engstrom, W.W. Wasserman, B. Lanhard, JASPAR: An open-access database for eukaryotic transcription factor binding profiles, Nucleic Acids Research 32 (2004) D91–D94.
[43] T.D. Schneider, G.D. Stormo, L. Gold, A. Ehrenfeucht, Information content of binding sites on nucleotide sequences, Journal of Molecular Biology 188 (1986) 415–431.
[44] P. Scordis, D.R. Flower, T. Attwood, FingerPRINTScan: Intelligent searching of the prints motif database, Bioinformatics 15 (10) (1999) 799–806.
[45] K. Sjölander, K. Karplus, M.P. Brown, R. Hughey, A. Krogh, I.S. Mian, D. Haussler, Dirichlet mixture: A method for improved detection of weak but significant proteins sequence homology, CABIOS 12 (4) (1996) 317–345.
[46] R. Staden, Methods for calculating the probabilities for finding patterns in sequences, CABIOS 5 (1989) 89–96.
[47] G.D. Stormo, T.D. Schneider, L.M. Gold, A. Ehrenfeucht, Use of the perceptron algorithm to distinguish translational initiation sites in E.coli, Nucleic Acid Research 10 (1982) 2997–3012.
[48] G.D. Stormo, DNA binding sites: Representation and discovery, Bioinformatics 16 (1) (2000) 16–23.
[49] R. Tatusov, S. Atschul, E. Koonin, Detection of conserved segments in proteins: Iterative scanning of sequence databases with alignmnet blocks, PNAS 91 (25) (1994) 12091–12095.
[50] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[51] J.C. Wallace, S. Henikoff, PATMAT: A searching and extraction program for sequence, pattern and block queries and databases, CABIOS 8 (3) (1992) 249–254.
[52] P. Weiner, Linear pattern matching algorithm, in: 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[53] T.D. Wu, C.G. Neville-Manning, D.L. Brutlag, Fast probabilistic analysis of sequence function using scoring matrices, Bioinformatics 16 (3) (2000) 233–244.
[54] T.D. Wu, C.G. Neville-Manning, D.L. Brutlag, Minimal-risk scoring matrices for sequence analysis, Journal of Computational Biology 6 (2) (1999) 219–235.
[55] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Transactions on Information Theory 24 (1978) 530–536.