Finding significant matches of position weight matrices in linear time

Cinzia Pizzi, Pasi Rastas, and Esko Ukkonen

***Abstract*— Position weight matrices are an important method for modeling signals or motifs in biological sequences, both in DNA and protein contexts. In this paper we present fast algorithms for the problem of finding significant matches of such matrices. Our algorithms are of the on–line type, and they generalize classical multi-pattern matching, filtering, and super-alphabet techniques of combinatorial string matching to the problem of weight matrix matching. Several variants of the algorithms are developed, including multiple matrix extensions that perform the search for several matrices in one scan through the sequence database. Experimental performance evaluation is provided to compare the new techniques against each other as well as against some other on–line and index–based algorithms proposed in the literature. Compared to the brute-force $O(mn)$ approach, our solutions can be faster by a factor that is proportional to the matrix length m. Our multiple-matrix filtration algorithm had the best performance in the experiments. On a current PC, this algorithm finds significant matches (p = 0.0001) of the 123 JASPAR matrices in the human genome in about 18 minutes.**

***Index Terms*— Position weight matrices, position specific scoring matrices, profiles, pattern search, string matching.**

I. Introduction

Position weight matrices [11], [13], [27] are statistical models for sequence signals, such as transcription factor binding sites, in DNA and in other biological sequences. Fast search of significant matches between weight matrices and sequences is a crucial requirement for nowadays biological sequence analysis tools, due to the exponential growth of both DNA and protein sequence databases as well as alignment block databases from which the weight matrices can be synthesized (e.g., TRANSFAC [16], PRINTS [2], BLOCKS [14], JASPAR [22]). High–performance matrix search tools are needed, for example, in genomewide analysis of gene regulation (e.g., [12]).

The algorithms for position weight matrix search can be divided into two groups that substantially differ in their approach, namely the on–line algorithms and the index–based algorithms. The *index–based algorithms* utilize a separately constructed index structure of the target sequence that allows rapid accessing any location of the target. The index may provide a fast search at the cost of possibly large time and space requirement of the index construction. The proposed index–based algorithms use suffix trees [8], [24] or suffix arrays [3] as the index. The *on–line algorithms* perform the search in one left–to–right scan through the target sequence. All new algorithms introduced in this paper are of the on–line type.

The popular on–line algorithms (e.g. [19], [25], [29]) use a straightforward trial and error search in a sliding window. The time requirement becomes $O(mn)$, where m and n are the lengths of the matrix and of the target sequence, respectively. As this is slow, there has been some recent developments of more advanced on–line algorithms based on various ideas. In [30], the *lookahead* approach was introduced which utilizes score properties to devise partial thresholds that allow one to terminate the comparison of symbols as soon as it is clear that no match will occur. More recent proposals include matrix partitioning [15], filtering approaches based on alphabet reduction [4] or on clusters of similar matrices [15], and shift–add technique [21]. Techniques using Fast Fourier Transform [20] and data compression [10] have also been proposed, but besides the interesting theoretical approach it is not clear how efficient they can be in practical applications.

In this paper we present new fast on–line al-

This article is an expanded and revised version of [18].

Cinzia Pizzi is with Department of Information Engineering, University of Padova, Via Gradenigo, 6/A, 35133 Padova, Italy. E-mail: cinzia.pizzi@dei.unipd.it

Pasi Rastas and Esko Ukkonen are with Department of Computer Science, University of Helsinki and Helsinki Institute for Information Technology HIIT, Helsinki University of Technology and University of Helsinki, P.O. Box 68, 00014 University of Helsinki, Finland. E-mail: Firstname.Lastname@cs.helsinki.fi

Supported by the Academy of Finland grants 211496 (From Data to Knowledge) and 7523004 (Algorithmic Data Analysis) and by EU project *Regulatory Genomics*.

gorithms for the problem of finding weight matrix matches that score higher than a given significance threshold. Our algorithms are based on the classical multi-pattern matching, filtration, and super-alphabet techniques originally developed for exact and approximate key–word matching. Each technique is introduced in its basic version among with its variants. For critical analysis of the various approaches, an experimental comparison of all proposed and some old techniques is provided.

The paper is organized as follows. Section II formalizes the problem of weight matrix search, recalls the underlying probabilistic scoring model and how the significance threshold is given as a p-value. We also explains the so-called lookahead approach [30], that is often used in more advanced algorithms, including the ones proposed in this paper.

In Section III we introduce our Aho-Corasick expansion algorithm and its pruned variant, all based on classical pattern matching techniques and lookahead. The Aho–Corasick algorithm [1], [7], [17] is a very fast method for multiple key–word search. To use it for weight matrices, we explicitly generate for a given matrix the sequences that score above the significance threshold and use these qualifying sequences as the key–word set in the Aho–Corasick algorithm. The key–word set may become so large that it cannot be accommodated in the fast storage of the computer, hence slowing down the practical performance of this theoretically appealing on–line algorithm. The same idea of generating the qualifying sequences and applying multiple key–word search (but other than Aho–Corasick) independently appears in [21].

To solve the storage problem of the key–word searching, we introduce in Section IV the filtration approach, a well-known technique utilized in approximate pattern matching in strings (e.g., [28]). Filtration algorithms compute first an upper bound for the matching score of the sliding window. The actual score is evaluated only if the obtained bound exceeds the threshold. In fact, the lookahead scoring search algorithm of [30] already utilizes the filtering idea. The novelty of our method is to make the filtration faster by using a precomputed table of the scores in a fixed–width window of the matrix. This gives fast and robust algorithms that have good performance in practice as they can be tuned to fit the available fast memory by selecting the width of the window appropriately.

Section V finally gives a super–alphabet generalization of the naive search. The fact that this trick speeds up the search is an old observation in string algorithmics (e.g., [6], [9]). The method conceptually uses a super–alphabet consisting of constant–width tuples of the original symbols. This would give a speed–up which is independent of the significance threshold, making this algorithm competitive for low thresholds and long matrices as then the Aho–Corasick and filtration algorithms get slower. The super–alphabet method can be seen as a special case of the variable–width submatrix decomposition approach of [15], with quite different technical implementation.

Extending the preliminary comparisons reported in [18] we present in Section VI an experimental comparison of the proposed algorithms and some other recent on–line or index–based algorithms from [3], [4], [21], [24]. In the experiments, we used matrices from the JASPAR [22] database, consisting of DNA motifs for transcription factor binding sites, and from the PRINTS [2] database of protein motifs. Multiple-matrix filtration algorithms were the best search algorithms for the tested matrices.

A C++ implementation of our algorithms is available under the GNU general public licence at www.cs.helsinki.fi/u/prastas/pssm/.

II. Scoring matrices and weighted pattern search

A. The search problem

The problem we consider is to find good matches of a positionally weighted pattern in a sequence of symbols in some finite alphabet Σ. A *positionally weighted pattern* is a real–valued $m \times |\Sigma|$ matrix $M = (M(j, a))$. In literature, these patterns are also called, e.g., position weight matrices, position specific scoring matrices, and profiles. As a shorthand, we use *pattern* or *matrix* in this paper. The matrix gives a weight (score) for each alphabet symbol a at each position j. We call m the *length* and Σ the *alphabet* of M. Table I gives an example pattern.

Pattern M matches any sequence $u = u_1 \dots u_m$ of length m in alphabet Σ. The quality of the match is indicated by the *match score* $W_M(u)$ of u with respect to M. The score is defined as

$$W_M(u) = \sum_{j=1}^{m} M(j, u_j). \quad (1)$$

Let $S = s_1 s_2 \dots s_n$ be an n symbol long sequence in alphabet Σ. Pattern M gives a match score for any m symbol long segment $s_i \dots s_{i+m-1}$ (called *an m-segment*) of S. We denote the match score of M at location i as

$$w_i = W_M(s_i \dots s_{i+m-1}). \quad (2)$$

TABLE I

A POSITIONALLY WEIGHTED PATTERN IN DNA ALPHABET. THE PATTERN WAS OBTAINED FROM THE COUNT MATRIX GATA-3 (MA0037) OF JASPAR [22] WHICH WAS TRANSFORMED INTO LOG-ODDS MATRIX USING BACKGROUND DISTRIBUTION $q_A = 0.343,\ q_C = 0.187,\ q_G = 0.189,\ q_T = 0.281$ (THE SCORES MULTIPLIED BY 100 AND ROUNDED TO INTEGERS). A PSEUDO-COUNT q_a WAS FIRST ADDED TO THE COUNTS FOR EACH ALPHABET SYMBOL a.

	A	C	G	T
1	14	17	-106	12
2	-416	-231	164	-416
3	103	-416	-232	-264
4	-416	-416	-85	118
5	58	-231	-106	7
6	-36	-132	112	-77

The problem studied in this paper can now be formalized as follows. Given a real–valued *significance threshold* k, the *weighted pattern search problem with threshold* k is to find all locations i of sequence S such that $w_i \geq k$. In addition to the locations i, also knowing the values w_i is of interest in many applications.

A special case of this search problem is the widely studied problem of exact string pattern matching in which one wants to find within sequence S the exact occurrences of a pattern $P = p_1 p_2 \dots p_m$ where each p_j is in Σ (e.g., [7]). The weighted pattern search formalism gives the exact pattern search problem by choosing the weight matrix M as $M(j, p_j) = 1$, and $M(j, v) = 0$ if $v \neq p_j$. Then threshold k selects the locations of S where the symbols of P match the corresponding symbol of S in at least k positions. Choosing $k = m$ gives the locations of the exact occurrences of P in S.

B. Probabilistic pattern model, log-odds scoring, and significance thresholding

In typical applications such as finding putative binding sites of transcription factors in DNA, the weights $M(j, a)$ are in fact log-odds scores of a probabilistic model of a signal to be detected against the background. The signal model is normally given as an $m \times |\Sigma|$ matrix Π such that $\Pi(j, a)$ gives the probability of the symbol a to occur in model position j. These probabilities can be obtained e.g. from the corresponding empirically constructed count matrix [22], possibly with added pseudocounts.

The background is usually modeled as an i.i.d. model, that is, as an $m \times |\Sigma|$ matrix π in which each row holds the same probability vector giving the background probability distribution of the alphabet symbols. We denote the background probabilities as q_a for $a \in \Sigma$. Hence $\pi(j, a) = q_a$ for all j.

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

$$Score(u) = \log \frac{Pr_\Pi(u)}{Pr_\pi(u)} = \sum_{j=1}^{m} \log \frac{\Pi(j, u_j)}{\pi(j, u_j)} = \sum_{j=1}^{m} \log \frac{\Pi(j, u_j)}{q_{u_j}}$$

Once the background π is fixed, say by estimating from the sequence S to be searched, the model Π can be translated into a positionally weighted pattern M:

$$M(j, a) = \log \frac{\Pi(j, a)}{\pi(j, a)}$$

Then the score obtained as (1) equals the above $Score(u)$.

The significance threshold k for the search is normally given indirectly by using the standard approach of statistics that utilizes the p-values to control the confidence of the findings. For given p-value p^*, the corresponding threshold is a value $k = k(p^*)$ such that in the background distribution the probability of sequences u such that $W_M(u) \geq k$, is p^*. Given p^*, the corresponding $k = k(p^*)$ can be evaluated by using a well-known pseudo-polynomial time dynamic programming algorithm [26], [30].

C. The naive algorithm and the lookahead scoring algorithm

Let us recall in this section some well-known search algorithms that will be included in our experimental comparison.

The weighted pattern search problem can be solved for given S, M, and k by evaluating w_i from (1) and (2) for each $i = 1, 2, \dots, n - m + 1$, and reporting locations i such that $w_i \geq k$. We call this the *naive algorithm* (NA). As evaluating each w_i from (1) takes time $O(m)$, the total search time of the naive algorithm becomes $O(mn)$, where m is the length of the pattern and n the length of the sequence S.

The *lookahead scoring algorithm* is an improved version of NA from [30] that precomputes the intermediate score thresholds that each prefix of a candidate segment must meet in order to keep the chance to be a match. The intermediate thresholds

are computed using the *maximal remainder scores*, defined for $0 \leq h \leq m$ as

$$R_h = \sum_{j=h+1}^{m} \max_{a \in \Sigma} M(j, a). \tag{3}$$

Obviously, R_h is the maximum possible score that can be achieved by the suffix starting at position $h+1$ of any candidate m-segment $s_i \dots s_{i+m-1}$. Let the sum of the score up to position h, $W_M(s_i \dots s_{i-1+h}) = \sum_{j=1}^{h} M(j, s_{i-1+j})$, and the maximal remainder scores R_h be below the matching threshold k. Then we know without seeing the remaining symbols that the m-segment cannot be a match as the total score will be less than k for any choice of the sequence beyond the location h.

Utilizing this observation, the lookahead scoring algorithm first tabulates the *intermediate thresholds*

$$T_h = k - R_h \tag{4}$$

for $h = 0, \dots, m$. The actual search at position i of S evaluates the score for $s_i \dots s_{i+m-1}$ incrementally term by term such that the next term $M(h+1, S_{i+h})$ is added to the score only if the accumulated score so far is at least T_h. Otherwise the search is stopped at the current i and resumed at $i+1$.

A further improvement of NA proposed in [30] is called the *permuted lookahead scoring algorithm* (PLS). This algorithm accumulates the scores of the matrix positions in an order which is not necessarily left-to-right but is determined specifically for each matrix to give optimal average performance. The aim is to detect unsuccessful m-segments as early as possible. This is achieved by choosing the matrix positions in decreasing order of the *expected loss* at position j, defined as

$$L_j = \max_{a \in \Sigma} M(j, a) - \sum_{a \in \Sigma} q_a M(j, a) \tag{5}$$

where q_a is the background probability of a.

On average, computing the score at j would increase the difference between maximum possible score and the actual score by L_j. This means that the partial score, when evaluated in this order, will drop below the intermediate threshold T_h for smallest possible h on average. Hence the search takes minimal average time at each location of S. Note that the maximal remainder score R_h has to be evaluated using the same order of matrix locations, determined by decreasing expected loss.

III. Aho–Corasick expansion algorithm

Pattern M and threshold k determine the m symbols long sequences whose matching score is $\geq k$. We may explicitly generate all such sequences in a preprocessing phase before the actual search over S, and then find their occurrences fast from S, using some multipattern search technique of exact string matching. We will use the Aho–Corasick multipattern search algorithm for that purpose while [21] used the q–gram variant of the multipattern backward DAWG matching algorithm.

A. Full version

The preprocessing phase takes M and k, and generates the Aho–Corasick pattern matching automaton as follows. For each sequence $x \in \Sigma^m$, add x to the set D of the *qualifying sequences* if $W_M(x) \geq k$. In practice we avoid generating and checking the entire Σ^m by using the lookahead technique of [30] to limit the search. We build D 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 $\geq k$.

More technically, we use the same trick as in the lookahead scoring algorithm. For a prefix $u_1 \dots u_h$, $h \leq m$, of a full m–segment, let

$$W_M(u_1 \dots u_h) = \sum_{j=1}^{h} M(j, u_j) \tag{6}$$

be the *prefix score* of $u_1 \dots u_h$. Now, if $W_M(u_1 \dots u_{h-1} u_h) < T_h$, we know for sure that no string with prefix $u_1 \dots u_{h-1} u_h$ can have a score $\geq k$ with respect to M, and there is no need to expand $u_1 \dots u_{h-1}$ to $u_1 \dots u_{h-1} u_h$. Otherwise there is a string with prefix $u_1 \dots u_{h-1} u_h$ that has score $\geq k$, and the expansion by u_h is accepted.

We organize this process in the breadth–first order such that all still alive prefixes of length $h-1$ are examined and expanded to the length h by all $u_h \in \Sigma$ such that $W_M(u_1 \dots u_{h-1} u_h) \geq T_h$. This is started with $h = 1$ (only the empty string is alive at the start) and repeated for $h = 2, \dots, m$. The sequences alive at $h = m$ constitute D.

For large k, only a small fraction of Σ^m will be included in D. Denoting the total length of the sequences in D by $|D|$, the above procedure generates D in time $O(|D|+|M|)$ where the $O(|M|) = O(m|\Sigma|)$ time is needed for computing the intermediate thresholds T_h from M.

The standard Aho–Corasick pattern matching automaton $AC(D)$ for the sequences in D is then constructed [1], [7], followed by a postprocessing phase to get the 'advanced' version of the automaton [17] in which the failure transitions have been eliminated. This version is the most efficient for small alphabets like that of DNA. The construction of the automaton needs time and space $O(|D||\Sigma|)$. For each $w \in D$, we associate the score $W_M(w)$ with the state of the automaton that corresponds to w.

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 as well as the precomputed scores of the members detected.

We call this search algorithm the *Aho–Corasick expansion algorithm* (ACE algorithm). The total running time of ACE is $O(|D||\Sigma|+|M|) = O(|\Sigma|^{m+1}+m|\Sigma|)$ for preprocessing M to obtain $AC(D)$, and $O(|S|)$ for scanning S with $AC(D)$. Note that the scanning time does not depend on the matrix length m.

B. Pruned version

Next we develop a pruned variant of the Aho-Corasick expansion that sometimes gives a notably smaller pattern matching automaton than the full version of Section III-A. The variant is based on the following observation. Let sequence v, $|v| < m$, be such that for all $w \in \Sigma^{m-|v|}$, sequence vw qualifies (i.e., $W_M(vw) \geq k$) and hence is in D. Then detecting v at some position i of S indicates that a qualifying sequence must occur at i. This is because independently of what $m-|v|$ symbols long sequence w follows the occurrence of v in S, we know that the total score of vw must be $\geq k$. Hence we can use in the multipattern search only v instead of all sequences vw. To get greatest saving, v should be shortest possible.

In this way we see that to find the occurrences of the original qualifying sequences in D, it suffices to search for sequences in the set

$$D_{pruned} = \{\, v \mid vw \in D \text{ for all } w \in \Sigma^{m-|v|} \text{ and } v \text{ is shortest possible}\}.$$

Set D_{pruned} can be constructed directly, without at first constructing the full D. We use a technique that is similar to the breadth-first search construction of D. In addition to the maximal remainder scores R_h, the construction also uses the *minimal remainder scores* r_h defined for $0 \leq h \leq m$ as

$$r_h = \sum_{j=h+1}^{m} \min_{a \in \Sigma} M(j, a). \tag{7}$$

Hence the minimal remainder score at position h of M is the smallest possible total score that can be obtained from positions $h+1$ to m.

It should be obvious that $v = v_1 \dots v_h$ is in D_{pruned} if and only if $W_M(v) \geq k - r_h$ and $W_M(v') < k - r_h$ for any proper prefix v' of v. To facilitate the construction of D_{pruned} we therefore tabulate the *minimum gain thresholds*

$$t_h = k - r_h.$$

The breadth-first construction then examines the still alive prefixes $u_1 \dots u_{h-1}$ for all u_h in Σ as follows: If $W_M(u_1 \dots u_{h-1}u_h) \geq t_h$, then $u_1 \dots u_{h-1}u_h$ is added to D_{pruned} and made dead. Otherwise, if $W_M(u_1 \dots u_{h-1}u_h) \geq T_h$, then $u_1 \dots u_{h-1}u_h$ is made alive for the next round. The other $u_1 \dots u_{h-1}u_h$ are made dead. Finally, add to D_{pruned} the sequences $u_1 \dots u_m$ that are alive at $h = m$.

As the total length of the sequences in D_{pruned} can for some matrices M be considerably smaller than the length of the sequences of D, the Aho-Corasick pattern matching automaton $AC(D_{pruned})$ can be clearly smaller than $AC(D)$. This may give a faster search because of improved cache memory behaviour. However, this search would only report the occurrence locations of the sequences in D. In applications also the scores of the occurrences are of interest. To be able to report these, the entire m-segment making the occurrence must be scanned and the corresponding total score must be output. This complicates the search and makes the data structures larger as we need to precompute and store the scores for all members of D. In our experiments we observed that the search with $AC(D_{pruned})$ was actually somewhat slower than with $AC(D)$. Therefore the search times of $AC(D_{pruned})$ are not reported in Section VI.

C. Size of Aho–Corasick automata

When deciding which algorithm to use it is useful to evaluate the size of the Aho–Corasick automaton $AC(D)$ directly, without explicitly constructing the automaton. Given M and $k = k(p^*)$, this can be done as follows.

First we note that for uniform background distribution the expected number of qualifying sequences is $\leq p^*|\Sigma|^m$. Hence the number of states of $AC(D)$ is $\leq p^*m|\Sigma|^m$ in this case.

The accurate number of the states (for any background) can be computed by tabulating the number of different sequences reaching each level of the automaton. Let G_1 and G_0 be the largest and smallest

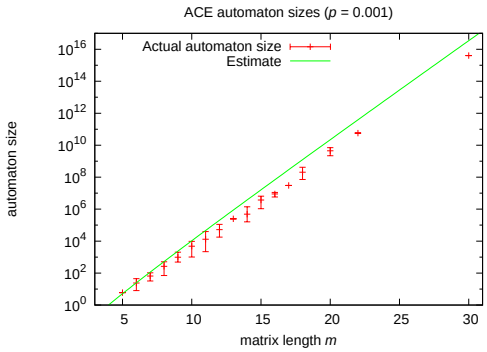

Fig. 1. Vertical bars show the maximum, minimum and average number of states of $AC(D)$ for patterns of each length $4, \ldots, 30$ in the JASPAR database, $p = 0.001$. The solid line gives the size estimate $p^* m 4^m$.

possible partial (prefix) score given by matrix M. Let $C(h, g)$ denote the number of different sequences $u_1 \ldots u_h$ such that M gives them score $= g$, that is, $g = \sum_{j=1}^{h} M(j, u_j)$. Then $C(h, g)$ can be evaluated for $g = G_0, \ldots, G_1$ and $h = 0, \ldots, m$ from

$$C(0,g) = \begin{cases} 1 & \text{if } g = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$C(h,g) = \sum_{a \in \Sigma} C(h-1, g - M(h,a))$$

The number of different prefixes $u_1 \ldots u_h$ of the qualifying sequences in D is now $N_h = \sum_{g \geq T_h} C(h, g)$, where T_h is as in (4). By the properties of the Aho–Corasick trie representing D, the total number of the states of $AC(D)$ then equals $\sum_{j=0}^{m} N_j$.

Fig. 1 gives the sizes of $AC(D)$ for the 123 JASPAR matrices used in the experiments of Section VI.

IV. Filtration algorithms

A. Window–based filtration

While the Aho–Corasick expansion algorithm of Section III scans S very fast, it is not always robust enough for practical use, due to its huge memory requirements: If the matrix length m is large and k is relatively small, set D of the qualifying sequences can become so large that $AC(D)$ cannot be accommodated in the available fast memory. For example, $AC(D)$ for the longest JASPAR matrix ($m = 30$) has more than $4 \cdot 10^{15}$ states when $p = 0.001$ (Fig. 1). This leads to a considerable slow–down of the scanning phase because of secondary memory effects. Moreover, repeating the construction of the Aho–Corasick automaton for different k but the same M seems inefficient.

To tackle these problems we develop in this section algorithms that use the so–called filtration. The size of the filtration machine that scans the sequence S can be selected flexibly to fit the available memory space. The general idea of filtration algorithms is that instead of the accurate value of the score, the search first evaluates an *upper bound* for the score of each m–segment of S. Whenever the upper bound value is $\geq k$, we know that also the score itself may be $\geq k$. However, to eliminate false positives this must be checked separately as the actual score may also be $< k$. But if the upper bound is $< k$, we know for sure that the score itself must be less than k. Checking is not needed, and we can continue immediately to the next segment.

This filtration scheme gives a fast search, provided that the upper bound can be evaluated fast (faster than the accurate value) and the bound is strict enough to keep the number of false positives small. Note moreover, that the filter is lossless, i.e., it always finds all true positives.

To get an upper bound, we will evaluate the score only for a limited window of the matrix M. The window score added with the maximal remainder score for the rest of the matrix clearly is an upper bound for the score of the whole matrix. The evaluation of the window scores can be done using an Aho–Corasick machine or using a simpler technique based on tabulating the scores of all possible sequences for the window. By making the window narrow enough we can get a filtration machine that fits to the available memory.

The position of the window within the matrix is selected such that the filtration accuracy is maximized. This is achieved by using a window position for which the expected loss (5) is maximal. In the checking phase the columns of the matrix outside the window are evaluated in the decreasing order of the expected loss, giving shortest checking on average. In fact, our filtration algorithm is the permuted lookahead scoring algorithm [30] (Section II-C) generalized such that a window of several columns takes the role of the first evaluated column.

An additional useful feature of the algorithms of this section is that the filtration machines for several matrices can be united into a combined filtration machine that performs the pattern search for all matrices in a single scan of the sequence S. In some applications it might not be feasible to store the entire sequence in which case the single–scan approach can

be utilized.

B. Lookahead filtration algorithm

The *Lookahead filtration algorithm* (LF) fixes the details of the filtration scheme as follows. Let S, M, and k be as before, and let h, $h \leq m$, be the *width of the window*. The optimal window gives the maximal total expected loss among all windows of width h. Its position i_0 can be determined as

$$i_0 = \arg_{1 \leq i \leq m-h+1} \max \sum_{j=i}^{i+h-1} L_j.$$

Matrix M is then preprocessed to get a finite–state automaton F that reports for each h symbols long segment v of S the score given to v by the optimal window. The automaton F has a state for each sequence $u = u_1 \dots u_h \in \Sigma^h$. It is convenient to denote this state also by u. The window score

$$W_{M[i_0,i_0+h-1]}(u_1 \dots u_h) = \sum_{j=i_0}^{i_0+h-1} M(j, u_j)$$

of sequence u is associated with state u. The state transition function τ of F is defined for all u in Σ^h and a in Σ as $\tau(u, a) = v$ where $v = u'a$ and u' in Σ^{h-1} is the $h-1$ symbols long suffix of u.

The regular structure of the states and the transition function τ of F makes a very efficient implementation possible. Automaton F is implemented simply as an array of size $|\Sigma|^h$, also denoted by F, whose entry $F(u)$ stores the window score $W_{M[i_0,i_0+h-1]}(u)$ 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 concatenation with a. This *shift–and–concatenate* technique can be conveniently used if $|\Sigma|$ is a power of 2; this is the case for DNA matrices. For other alphabets such as the amino acid alphabet we use arithmetic operations: multiply by $|\Sigma|$, add a (encoded by one of $0, 1, \dots, |\Sigma| - 1$), and subtract $u"|\Sigma|^h$ where $u"$ is the header symbol of u.

The filtration phase is done by scanning the sequence S with F to obtain, in time $O(n)$, the window score of every h symbols long segment of S. Let t_i be the score for such a segment that starts at s_i. (Note that $t_i = F(u)$ where u is the state of F after scanning $s_1 \dots s_{i+h-1}$.) Now, the upper bound used in filtration is $t_i + R_h$ where R_h is the (also precomputed) maximal remainder score (3) for M *outside* the optimal window. If $t_i + R_h < k$, then the full m–segment at the present location must score less than k, and we can continue the search immediately to the next location, without evaluating the full score. On the other hand, if $t_i + R_h \geq k$, the full score must be evaluated to see if it really is $\geq k$. This is done by adding to t_i the scores of matching the remaining positions of M outside the optimal window against the corresponding element of $s_{i-i_0+1}, \dots, s_{i-i_0+m}$. The addition is done in the decreasing order of the expected loss L_j. The checking is terminated as soon as the score accumulated so far plus the maximal remainder score (3) for the rest goes under k.

Note that F does not depend on k. The same filtration automaton F can be used with any threshold k, and hence F needs to be constructed and stored only once.

C. Running time and filtration specificity

To analyse the running time of the LF algorithm, we note first that the filtration automaton F and the maximal remainder scores 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 γ the number of locations to be checked, the total checking time becomes $O(\gamma(m - h)) = O(n(m - h))$. Filtration pays off only if γ is (much) smaller than $n = |S|$. Increasing k or h would obviously decrease γ.

To estimate γ, let us write $\gamma = \delta n$. Then for target sequences taken from the background distribution, δ is proportional to the probability of that the filtration score $t_i + R_h$ is $\geq k$. This depends on M. For the JASPAR matrices we estimated the average values of δ for filtration window width $h = 7$. For $p = 0.01, 0.001, 0.0001,$ and 0.00001, the corresponding values of δ are 0.175, 0.0815, 0.0369, and 0.0178. This also gives average *filtration specificity* $\delta : p$ which is the ratio between the expected number of positive occurrences of the filtration window and the truly positive occurrences of the entire M.

D. Multipattern generalization

The automaton F can be extended to multipattern case, to handle several M simultaneously, as follows. Let $M_1, \dots, M_K$ be the patterns we want to search. The filtration automaton stores in its entry $F(u)$ all the K window scores of u given by the optimal window of each of $M_1, \dots, M_K$. The above time bounds for initialization should be multiplied by K to get time bounds for the generalized algorithm. The algorithm scans S only once but the scanning

becomes slower by a factor which is proportional to the average number of patterns for which the checking phase is entered. We call this algorithm the *Multiple matrix lookahead filtration algorithm* (MLF).

To get an idea of what values of h and K may be feasible for MLF in practice, assume that $|\Sigma| = 4$. Then array F contains about 2.5×10^8 numbers if $h = 9$ and $K = 1000$ or if $h = 10$ and $K = 250$. Storing them takes about 1 gigabytes which is a reasonable main memory size of a current PC. To get fastest possible performance, the array should fit into the CPU's cache memory that is typically much smaller.

E. Aho-Corasick lookahead filtration and multipattern generalization

The idea of the ACE algorithm to use only the qualifying sequences in the search can as well be used for the window search of the filtration method. The qualifying sequences for a given window and threshold k are the sequences whose window score plus the corresponding maximal remainder score is $\geq k$. These sequences can be found using the methods of Section III, adapted to the optimal window. The Aho–Corasick multipattern automaton for these sequences can be put in the role of the finite–state machine F of the LF algorithm. We call this variant the *Aho–Corasick lookahead filtration algorithm* (ACLF). Average size of the ACLF automaton for the Jaspar matrices is 2452 states when $h = 7$ and $p = 0.001$.

As compared to the LF algorithm, ACLF saves space as the qualifying sequences for a window can be sparse.

The multiple matrix version is again possible by combining in an obvious way the Aho–Corasick automata for all matrices. The resulting method is called the *Multiple matrix Aho–Corasick lookahead filtration algorithm* (MACLF). The size of the multipattern automaton for all Jaspar matrices is 21845 states and 170060 window scores (and matrix indices) associated with the states ($h = 7$, $p = 0.001$).

V. Speed–up by super–alphabet

Finally we give a simple 'super–alphabet' generalization of the naive algorithm NA. As the running time of this algorithm does not depend on k, the algorithm has potential of performing relatively well for low k and for long matrices for which the filtration does not work as the lookahead bound tends to be overly conservative.

To define the super–alphabet we fix the integer *width* q of the alphabet. Then each q–tuple of the original alphabet is a super–alphabet symbol.

Matrix M is preprocessed to obtain an 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

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

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

The score of each m symbols long segment of S can now be evaluated in $O(m/q)$ steps using the shift–and–concatenate technique of the previous section to find fast the appropriate entries of $\hat{M}$. We call this method the Naive super–alphabet algorithm (NS).

The search time using algorithm NS becomes $O(nm/q)$, giving a (theoretical) speed–up of algorithm NA by a factor of q, independently of the threshold k. In practice the larger overhead of algorithm NS makes the speed–up smaller. With some care in implementation, matrix $\hat{M}$ can be constructed in time $O(m|\Sigma|^q/q)$.

The submatrix decomposition method of [15] is a generalized super–alphabet technique with varying alphabet width.

VI. Experimental Performance Comparison

A. Experimental set–up

For the experimentation we implemented the well-known base–line algorithms NA (naive algorithm) and PLS (permuted lookahead scoring) as well as our six new algorithms ACE (Aho–Corasick expansion), LF (lookahead filtration), ACLF (Aho–Corasick lookahead filtration) and NS (naive super-alphabet) as well as the multi–matrix variants MLF (multiple matrix lookahead filtration) and MACLF (multiple matrix Aho–Corasick lookahead filtration). The algorithms were implemented in C++. The implementation is available under GNU license at www.cs.helsinki.fi/u/prastas/pssm/.

Moreover, using the software implementation by the original authors, our experiments included: The shift–add (SA) and the enumerative BG algorithm (EBG) and the multimatrix version (MEGB) of EGB by [21]; the index–based (suffix–array) program PoSSuMsearch [3], [4]; the index–based (suffix–tree) program STORM [24]. In the running time tables, POSSUM+ refers to PoSSuMsearch with the index construction time included, while POSSUM- is the mere search time using a preconstructed index. For

program STORM such a separation of the two phases was not possible. Moreover, STORM works only for DNA matrices. The available implementation of the submatrix decomposition method of [15] is obviously not optimized for speed as it took in our experiments at least twice the running time of the base–line algorithm NA. Therefore we do not report detailed timing results for that program.

We collected 123 positionally weighted matrices for DNA from the JASPAR database [22]. The length of these matrices varied from 4 to 30, with average length 10.8. As the target sequence to search the matrix occurrences we used a 50 megabases long DNA sequence. The sequence contained segments taken from the human and the mouse genome. To test the behaviour of the algorithms on very long patterns, we constructed a collection of 13 patterns, each having a length of 100, by concatenating the JASPAR matrices.

We also experimented with amino acid matrices by choosing a random subset (five of each length from 5 to 30, and two of length 31) of 132 matrices from the PRINTS database [2]. The target sequence was a 50 megabases long amino acid sequence that was made by concatenating a sample of SWISS-PROT sequences [5].

The significance threshold k for the experiments was given as a p-value as follows. The original count matrices from the JASPAR and PRINTS databases were first transformed into log-odds scoring matrices using background distribution estimated from the DNA or amino acid sequence and by adding for all alphabet symbols $a \in \Sigma$ a pseudo-count q_a to every original count matrix entry for a; here q_a is the estimated background probability of a. The score that corresponds to a given p-value was then computed using well-known pseudo-polynomial time dynamic programming algorithm [26], [30]. As this method requires an integer-valued matrix, the log-odds scores were first multiplied by 100 and then rounded to integers.

The reported running times are for 3 GHz Intel Pentium IV processor with 2 gigabytes of main memory, running under Linux. The compiler used in the experiments was gcc with parameters "-O3-march=pentium4 -fforce-addr -funroll-loops -frerun-cse-after-loop -frerun-loop-opt -falign-functions=4". We also experimented with 2,13 GHz Core 2 Duo machine, with essentially similar results (slight differences explained by different cache memory size).

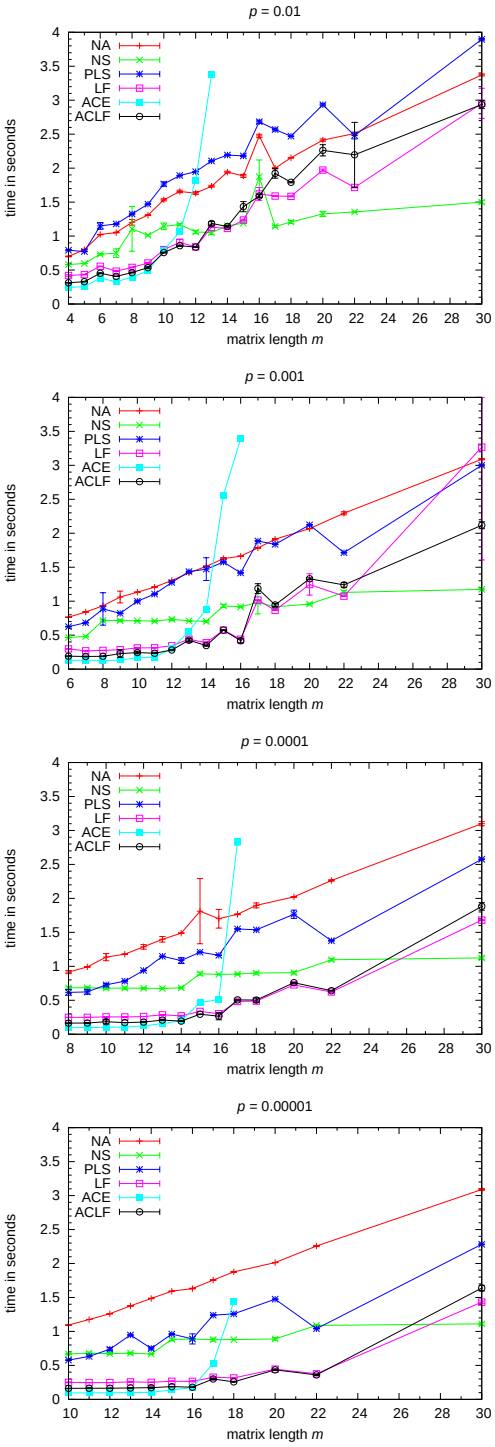

Fig. 2. The dependency of the running time of different algorithms on the pattern length for some p–values. Average running times for DNA patterns of each length $4, \ldots, 30$ in the JASPAR database as well as the standard deviations of 10 runs are shown.

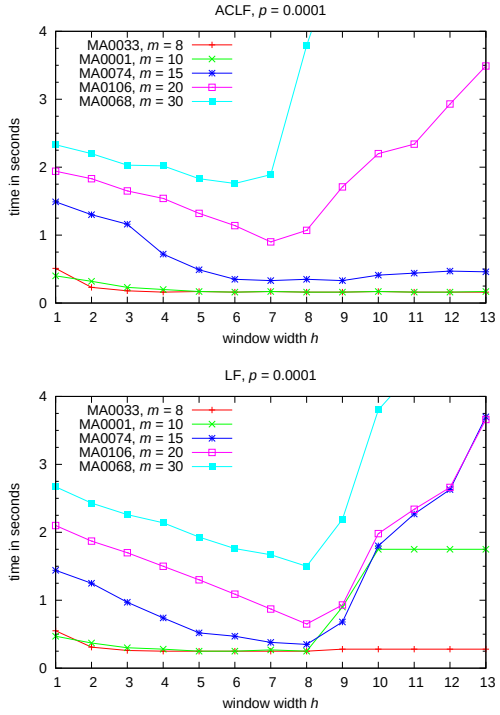

Fig. 3. Dependency on the window width for some individual matrices of different lengths. Running time of algorithms LF and ACLF for 5 example DNA patterns from JASPAR database when h varies from 2 to 13 and p = 0.0001.

B. DNA search

The run time results for searching Jaspar matrices are summarized in Tables II, III, and IV, and in Figures 2, 3, and 4. In algorithms LF, ACLF, MLF, and MACLF we used parameter value $h = 7$ and in algorithm NS we used $q = 7$, if not stated otherwise.

The ACE algorithm is the fastest among our algorithms for short matrices (up to length $m = 8, \ldots, 14$ depending on p); Table II gives the average times for all programs and for all matrices of length $m \leq 15$. For longer matrices the size of the pattern matching automaton of ACE becomes too large to fit the fast cache memory, and the algorithm gets rapidly much slower. Fig. 2 shows the dependency of the running time of some programs on the matrix length m; the collapse of ACE for longer matrices is evident. Similar behaviour was observed for algorithms EBG and MEGB of [21].

The window technique of the filtration algorithms LF and ACLF resolves this problem, giving algorithms that are consistently fast in the entire length range of JASPAR, algorithm ACLF being slightly faster for short matrices and LF for longer ones. As

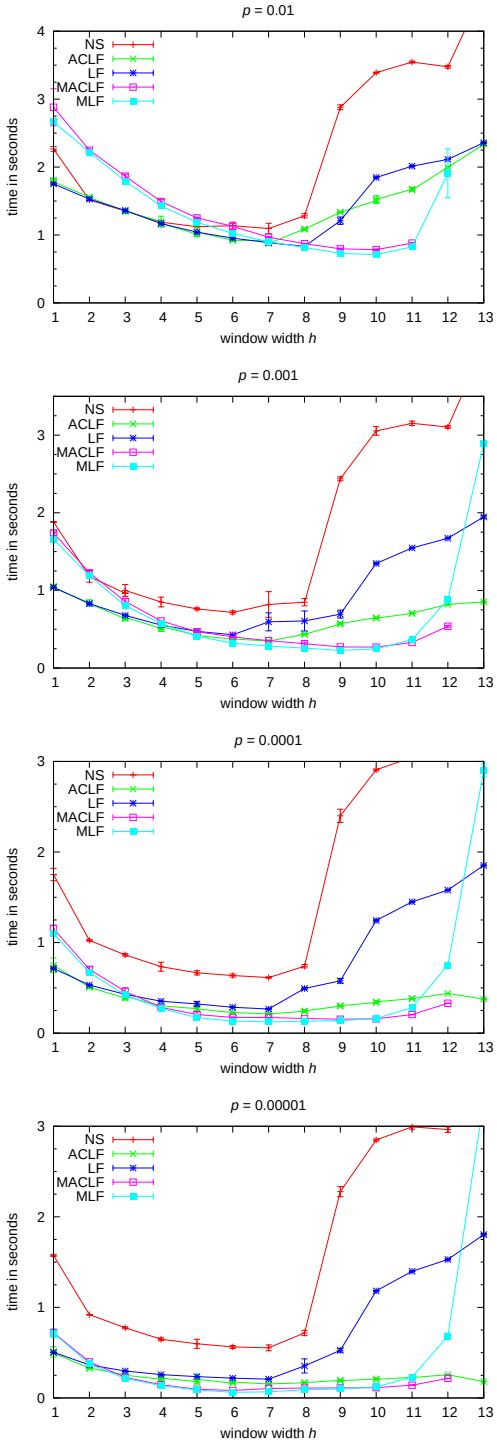

Fig. 4. Average dependency on the window width. Average running time of algorithms LF, ACLF, MACLF, MLF and NS for the 123 DNA patterns from the JASPAR database for some p-values and varying h (= q in the case of NS).

TABLE II

AVERAGE RUNNING TIMES PER PATTERN (IN SECONDS, PREPROCESSING INCLUDED) OF DIFFERENT ALGORITHMS WHEN SEARCHING FOR 108 JASPAR PATTERNS OF LENGTH $m \leq 15$, WITH VARYING P-VALUES. EACH REPORTED TIME IS AN AVERAGE OF 10 RUNS. THE LOWER PANEL IS FOR ALGORITHMS INTRODUCED IN THIS PAPER.

$p =$	10^{-5}	10^{-4}	10^{-3}	10^{-2}
NA	0.846	0.954	1.073	1.440
PLS	0.468	0.665	0.975	1.633
SA	0.864	0.873	0.838	0.819
EBG	0.140	0.199	0.389	1.269
MEBG, $q = 9$	0.015	0.044	0.152	**0.382**
POSSUM+	5.908	5.952	6.348	10.472
POSSUM-	0.018	0.059	0.445	4.579
STORM	0.956	1.041	1.747	-
ACE	0.071	0.105	0.298	1.318
LF	0.178	0.216	0.310	0.754
ACLF	0.117	0.147	0.237	0.715
MLF	0.010	**0.032**	**0.132**	0.680
MACLF	**0.008**	0.039	0.178	0.743
NS	0.551	0.590	0.646	1.002

TABLE III

AVERAGE RUNNING TIMES PER PATTERN (IN SECONDS, PREPROCESSING INCLUDED) OF DIFFERENT ALGORITHMS WHEN SEARCHING FOR ALL 123 JASPAR PATTERNS, WITH VARYING P-VALUES. EACH REPORTED TIME IS AN AVERAGE OF 10 RUNS.

$p =$	10^{-5}	10^{-4}	10^{-3}	10^{-2}
NA	0.987	1.079	1.190	1.572
PLS	0.574	0.771	1.079	1.775
POSSUM+	5.955	6.041	6.524	10.964
POSSUM-	**0.062**	0.148	0.631	5.071
STORM	0.996	1.204	2.173	-
LF	0.208	0.268	0.396	0.886
ACLF	0.153	0.214	0.353	**0.884**
MLF	0.066	**0.126**	**0.280**	0.916
MACLF	0.101	0.171	0.348	0.985
NS	0.537	0.612	0.687	1.043

algorithms ACE, EBG and MEBG were unable to search for all matrices reasonably efficiently, we omit them in further experiments. Algorithm SA is omitted as the available implementation only worked for short matrices.

Fig. 3 illustrates the dependency of the performance of algorithms LF and ACLF on the window width h for some example matrices. Increasing parameter h should improve the filtration performance and hence make the algorithm faster. In the experiment, the performance is seen to improve until $h = 7$ or $h = 8$. After that there is a strong slow-down. This is because the highest level cache memory is getting full. We verified that at $h = 7$ also the number of cache misses starts to grow. Fig. 4 summarizes the dependency on h by giving average execution times of LF, ACLF, MACLF, MLF and NS for all JASPAR matrices for varying h.

TABLE IV

AVERAGE RUNNING TIMES (IN SECONDS, PREPROCESSING INCLUDED) OF DIFFERENT ALGORITHMS FOR LONG DNA PATTERNS ($m = 100$) AND VARYING P-VALUES. EACH REPORTED TIME IS AN AVERAGE OF 10 RUNS.

$p =$	10^{-5}	10^{-4}	10^{-3}	10^{-2}
NA	9.724	10.119	9.901	10.331
PLS	9.463	10.265	11.475	13.147
POSSUM+	72.613	73.355	76.424	84.557
POSSUM-	16.859	17.601	20.670	28.803
STORM	35.138	36.098	40.677	-
LF	8.379	9.199	10.285	12.068
ACLF	7.392	8.074	9.050	10.481
MLF	9.982	10.707	11.905	13.938
MACLF	11.544	11.971	13.133	15.167
NS, $q = 6$	**3.076**	**3.150**	**3.293**	**3.701**

Table III gives the average running times for $h = 7$ ($q = 7$ in NS) for all matrices and for all algorithms that are efficient also for matrices longer than 15. The multiple matrix algorithms MLF and MACLF give the best total performance for small p, while for higher p the difference to LF and ACLF disappears. Fig. 4 also indicates that the multiple matrix algorithms are quite robust against the variation of h.

For the long DNA patterns with $m = 100$, algorithm NS is the best as reported in Table IV. We used $q = 6$ in algorithm NS. Note, however, that we made this experiment only for curiosity. The matrices here were artificially generated as we did not find such long matrices in databases.

TABLE V

AVERAGE RUNNING TIMES PER PATTERN (IN SECONDS, PREPROCESSING INCLUDED) OF DIFFERENT ALGORITHMS WHEN SEARCHING FOR A SUBSET OF PRINTS PATTERNS, WITH VARYING P-VALUES. EACH REPORTED TIME IS AN AVERAGE OF 10 RUNS.

$p =$	10^{-20}	10^{-15}	10^{-10}	10^{-5}
NA	1.408	1.630	1.773	**1.850**
PLS	0.624	1.046	1.606	2.461
POSSUM+	1.498	2.679	4.050	6.074
POSSUM-	1.100	2.281	3.652	5.676
LF, $h = 3$	**0.474**	**0.861**	**1.390**	2.224
ACLF, $h = 2$	0.498	0.887	1.431	2.215
MLF, $h = 4$	0.787	2.077	3.680	4.310
MACLF, $h = 4$	0.853	1.927	2.950	4.772

C. Amino acid search

For amino acid matrices the situation is somewhat different as can be seen in Table V and Fig. 5. As the

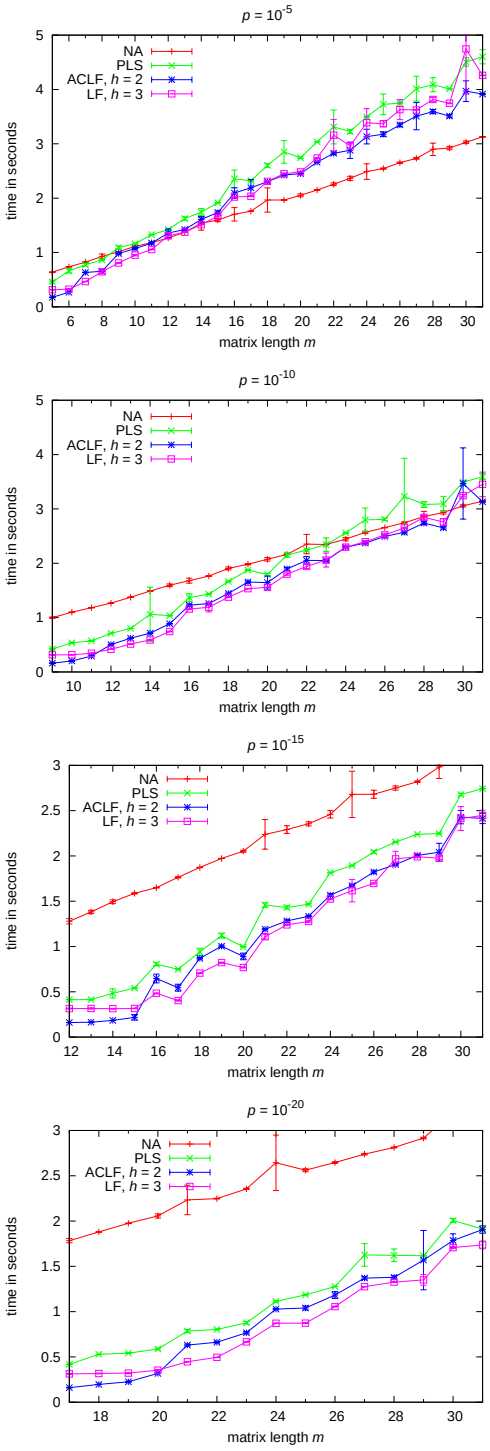

Fig. 5. The dependency of the running time of different algorithms on the pattern length for some p–values. Average running times for some patterns of each length $5, \ldots, 31$ in the PRINTS database are shown.

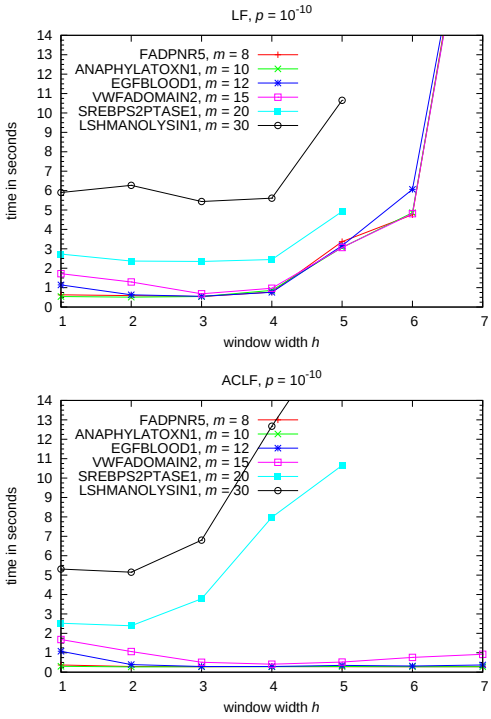

Fig. 6. Running time of algorithms LF and ACLF for 6 example amino acid patterns from PRINTS database when h varies from 1 to 7 and $p = 10^{-10}$.

amino acid alphabet is much larger, filtration can only be efficient if the p-value is very small. Algorithm LF with $h = 3$ gives the best performance but the difference to PLS is clearly smaller than in the case of DNA matrices. Fig. 6 illustrates the dependency on window width for some individual example matrices. As compared to the DNA matrices, the window–based filtration evidently has smaller effect. The multiple matrix algorithms MLF and MACLF had the best performance for window $h = 4$. Somewhat unexpectedly, the speed–up due to filtration did not suffice in the multiple–matrix case to compensate the overhead of the more complicated algorithmic structure.

VII. Discussion

Several novel on–line algorithms were proposed for the position weight matrix search. Six of them were implemented and experimentally compared with some earlier on–line and index–based search programs. Among the new methods, the ACE algorithm is theoretically optimal in the sense that its search speed does not depend on the matrix length. In practice, however, ACE suffers from large memory requirement which makes it competitive only for short matrices.

The multiple matrix versions MLF and MACLF of our filtration algorithm had in our experiments the best overall performance for DNA matrices. In some cases also MEGB of [21] and our LF and ACLF are very fast. For example, algorithm MLF finds in the human genome all matches of the 123 JASPAR matrices on significance level p = 0.0001 in about 18 minutes on a current PC. As a comparison, reading the human genome from disk to the main memory takes about 3 minutes.

It should be emphasized that the practical performance of the algorithms studied here quite strongly depends on the implementation details and on the properties of the memory hierarchy of the computer used. The observed speed differences between different on–line algorithm variants were often relatively small and can vary with the implementation and the computer. However, compared to the naive search (NA) and the permuted lookahead search (PLS) the new algorithms give a clear speed-up in the case of DNA matrices. Compared to PLS, the average speed-up factor was $4, \ldots, 8$ for $p = 0.001, \ldots, 0.00001$, and for individual matrices often much higher. For amino acid matrices the observed speed–up is more modest. For very long DNA matrices the only algorithm showing improved performance is the super–alphabet method (NS).

Choosing between on–line and index–based algorithms is a subtle task with no short answer. Building an index probably pays off if the target sequence (database) is long and stays unchanged, and there are lots of matrix searches to be done. The high speed of the on–line algorithms observed here is sufficient in many situations, noting in addition that on–line algorithms are conceptually simple and easy to implement. In our experiments the index-based programs were observed to perform the pure search (with preconstructed index) very fast for high significance values but the speed rapidly drops for low significances. If the index construction is included in the time comparison, then the index–based algorithms were not competitive with on–line methods for the 50 million symbols long targets used in the experiments. For longer targets the situation may change. However, we leave more extensive experimental study of these issues as a topic for further research.

VIII. Acknowledgement

We would like to thank Janne Korhonen for implementing some of our algorithms and for running the experiments.

References

[1] Aho, A. V. and Corasick, M.: Efficient string matching: An aid to bibliographic search, *Comm. ACM* 18 (1975), 333–340.

[2] Attwood T.K., and Beck, M.E., PRINTS - A Protein Motif Finger-print Database, *Protein Engineering*, 1994; 7(7):841–848.

[3] Beckstette M., Strothmann D., Homann R., Giegerich R., and Kurtz S., PoSSuMsearch: Fast and Sensitive Matching of Position Specific Scoring Matrices using Enhanced Suffix Arrays, in *Proc. German Conference in Bioinformatics*, 2004; GI Lecture Notes in Informatics, vol.: P-53, pages 53–64.

[4] Beckstette M., Strothmann D., Homann R., Giegerich R., and Kurtz S., Fast Index Based Algorithms and Software for Matching Position Specific Scoring Matrices, *BMC Bioinformatics*, 2006 Aug. 24; 7(1):389.

[5] Boeckmann B., Bairoch A., Apweiler R., Blatter M.C., Estreicher A., Gasteiger E., Martin M.J., Michoud K., O'Donovan C., Phan I., Pilbout S., and Schneider M., The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003," in *Nucleic Acids Research*, 2003; 31(1): 365–370.

[6] Boyer, R. S., and Moore, J. S., A fast string searching algorithm. *Commun. ACM* 1977; 20(10):762–772.

[7] Crochemore, M. and Rytter, W., *Text Algorithms*, Oxford University Press 1994.

[8] Dorohonceanu B., and Neville-Manning C.G., Accelerating Protein Classification Using Suffix Trees, in *Proc. 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, 2000; 128–133.

[9] K. Fredriksson, Shift-or string matching with super-alphabets. *Inf. Process. Lett.* 2003; 87(4):201–204.

[10] Freschi V., and Bogliolo A., Using Sequence Compression to Speedup Probabilistic Profile Matching, *Bioinformatics*, 2005; 21(10):2225–2229.

[11] Gribskov M., McLachlan A.D., and Eisenberg D., Profile Analysis: Detection of Distantly related Proteins, *Proc. Natl. Acad. Sci.*, 1987; 84(13):4355–8.

[12] O. Hallikas, K. Palin, N. Sinjushina, R. Rautiainen, J. Partanen, E. Ukkonen & J. Taipale: Genome-wide prediction of mammalian enhancers based on analysis of transcription-factor binding affinity. *Cell* 124 (January 13, 2006), 47–59.

[13] Henikoff S., Wallace J.C., Brown J.P., Finding protein similarities with nucleotide sequence databases, *Methods Enzymol.*, 1990;183:111–132.

[14] Henikoff J.G, Greene E.A., Pietrokovski S., and Henikoff S., Increased Coverage of Protein Families with the Blocks Database Servers, *Nucleic Acids Research*, 2000; 28(1): 228–230.

[15] Liefhooghe, A., Touzet, H. and Varre, J.: Large Scale Matching for Position Weight Matrices. In: *Proc CPM 2006*, LNCS 4006, pp. 401–412, Springer-Verlag 2006.

[16] Matys V., Fricke E., Geffers R., Gossling E., Haubrock M., Hehl R., Hornischer K., Karas D., Kel A.E., Kel-Margoulis O.V., Kloos D.U., Land S., Lewicki-Potapov B., Michael H., Munch R., Reuter I., Rotert S., Saxel H., Scheer M., Thiele S., and Wingender E., TRANSFAC: Transcriptional Regulation, from Patterns to Profiles, *Nucleic Acids Research*, 2003; 31(1):374–378.

[17] Navarro, G. and Raffinot, M., *Flexible Pattern Matching in Strings*, Cambridge University Press 2002.

[18] Pizzi C., Rastas P., and Ukkonen E., Fast Search Algorithms for Position Specific Scoring Matrices, in *Bioin-*

formatics Research and Development Conference (BIRD 2007), LNBI 4414, Springer, 2007; 239–250.
[19] Quandt K., Frech K., Karas H., Wingender E., and Werner T., MatInd and MatInspector: New Fast and Versatile Tools for Detection of Consensus Matches in Nucleotide Sequences Data, *Nucleic Acid Research*, 1995; 23(23):4878–4884.
[20] Rajasekaran S., Jin X., and Spouge J.L., The Efficient Computation of Position-Specific Match Scores with the Fast Fourier Transform, *Journal of Computational Biology*, 2002; 9(1):23–33.
[21] Salmela L. and Tarhio J., Algorithms for weighted matching, in Proc. SPIRE 2007, LNCS 4726, Springer, 2007, 276–286.
[22] Sandelin, A., Alkema, W., Engstrom, P., Wasserman, W.W. and Lanhard, B., JASPAR: an open-access database for eukaryotic transcription factor binding profiles, *Nucleic Acids Research* 32 (2004), D91–D94.
[23] Schneider T.D., Stormo G.D., Gold L. and Ehrenfeucht A., Information Content of Binding Sites on Nucleotide Sequences, *Journal of Molecular Biology*, 1986; 188:415–431.
[24] D. E. Schones, A. D. Smith, and M. Q. Zhang: Statistical significance of cis-regulatory modules *BMC Bioinformatics*, 2007; 8: 19
[25] Scordis P., Flower D.R., and Attwood T., FingerPRINTScan: Intelligent Searching of the PRINTS Motif Database, *Bioinformatics*, 1999; 15(10):799–806.
[26] Staden R., Methods for calculating the probabilities of finding patterns in sequences, *CABIOS*, 1989; 5(2):89–96.
[27] Stormo G.D., Schneider T.D., Gold L.M., and Ehrenfeucht A., Use of the 'Perceptron' Algorithm to Distinguish Translational Initiation Sites in E.coli, *Nucleic Acid Research*, 1982; 10:2997–3012.
[28] Ukkonen, E., Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science* 92 (1992), 191-211.
[29] Wallace J.C., and Henikoff S., PATMAT: a Searching and Extraction Program for Sequence, Pattern and Block Queries and Databases, *CABIOS*, 1992; 8(3):249–254.
[30] Wu T.D., Neville-Manning C.G., and Brutlag D.L., Fast Probabilistic Analysis of Sequence Function using Scoring Matrices, *Bioinformatics*, 2000; 16(3):233–244.

PLACE PHOTO HERE

Cinzia Pizzi received her PhD in Computer Engineering from University of Padova (Italy) in 2005. She then joined as post-doctoral researcher the University of Helsinki (Finland), and next INRIA (France). She is currently at the Department of Information Engineering of University of Padova. Her research interests includes design and analysis of algorithms, and motif discovery in biological sequences and networks.

PLACE PHOTO HERE

Pasi Rastas received his MSc degree in Computer Science in 2004 from University of Helsinki, Finland. His research interests include algorithms in general and in computational biology. He is currently writing his PhD at the University of Helsinki under supervision of Professor E. Ukkonen.

PLACE PHOTO HERE

Esko Ukkonen is since 1985 Professor of Computer Science of the University of Helsinki. He has had visiting positions at the University of California at Berkeley, University of Freiburg, and University of Bielefeld. He has published more than 140 original scientific articles on various areas including algorithms and data structures, combinatorial pattern matching, machine learning, and computational biology. In 1999-2001 he was the chairman of the Finnish Society for Computer Science, in 1999-2004 an Academy Professor of the Academy of Finland, and in 2004-2008 Research Director of the Helsinki Institute of Information Technology (HIIT).