Methodology article

Open Access

Fast index based algorithms and software for matching position specific scoring matrices

Michael Beckstette*[1,2], Robert Homann[1,2], Robert Giegerich[2] and Stefan Kurtz[3]

Address: [1]International NRW Graduate School in Bioinformatics and Genome Research, Center for Biotechnology (CeBITec), Bielefeld University, D-33594 Bielefeld, Germany, [2]Technische Fakultät, Universität Bielefeld, Postfach 100 131, D-33501 Bielefeld, Germany and [3]Zentrum für Bioinformatik, Universität Hamburg, 20146 Hamburg, Germany

Email: Michael Beckstette* - mbeckste@techfak.uni-bielefeld.de; Robert Homann - rhomann@techfak.uni-bielefeld.de; Robert Giegerich - robert@techfak.uni-bielefeld.de; Stefan Kurtz - kurtz@zbh.uni-hamburg.de

* Corresponding author

Published: 24 August 2006

doi:10.1186/1471-2105-7-389

Received: 20 April 2006
Accepted: 24 August 2006

Abstract

Background: In biological sequence analysis, position specific scoring matrices (PSSMs) are widely used to represent sequence motifs in nucleotide as well as amino acid sequences. Searching with PSSMs in complete genomes or large sequence databases is a common, but computationally expensive task.

Results: We present a new non-heuristic algorithm, called *ESAsearch*, to efficiently find matches of PSSMs in large databases. Our approach preprocesses the search space, e.g., a complete genome or a set of protein sequences, and builds an enhanced suffix array that is stored on file. This allows the searching of a database with a PSSM in sublinear expected time. Since *ESAsearch* benefits from small alphabets, we present a variant operating on sequences recoded according to a reduced alphabet. We also address the problem of non-comparable PSSM-scores by developing a method which allows the efficient computation of a matrix similarity threshold for a PSSM, given an E-value or a p-value. Our method is based on dynamic programming and, in contrast to other methods, it employs lazy evaluation of the dynamic programming matrix. We evaluated algorithm *ESAsearch* with nucleotide PSSMs and with amino acid PSSMs. Compared to the best previous methods, *ESAsearch* shows speedups of a factor between 17 and 275 for nucleotide PSSMs, and speedups up to factor 1.8 for amino acid PSSMs. Comparisons with the most widely used programs even show speedups by a factor of at least 3.8. Alphabet reduction yields an additional speedup factor of 2 on amino acid sequences compared to results achieved with the 20 symbol standard alphabet. The lazy evaluation method is also much faster than previous methods, with speedups of a factor between 3 and 330.

Conclusion: Our analysis of *ESAsearch* reveals sublinear runtime in the expected case, and linear runtime in the worst case for sequences not shorter than $|\mathcal{A}|^m + m - 1$, where m is the length of the PSSM and $\mathcal{A}$ a finite alphabet. In practice, *ESAsearch* shows superior performance over the most widely used programs, especially for DNA sequences. The new algorithm for accurate on-the-fly calculations of thresholds has the potential to replace formerly used approximation approaches. Beyond the algorithmic contributions, we provide a robust, well documented, and easy to use software package, implementing the ideas and algorithms presented in this manuscript.

Background

Position specific scoring matrices (PSSMs) have a long history in sequence analysis (see [1]). A high PSSM-score in some region of a sequence often indicates a possible biological relationship of this sequence to the family or motif characterized by the PSSM. There are several databases utilizing PSSMs for function assignment and annotation, e.g., PROSITE [2], PRINTS [3], BLOCKS [4], EMATRIX [5], JASPAR [6], or TRANSFAC [7]. While these databases are constantly improved, there are only few improvements in the programs searching with PSSMs. E.g., the programs *FingerPrintScan* [8], *BLIMPS* [4], and *MatInspector*[9] still use a straightforward $O(mn)$-time algorithm to search a PSSM of length m in a sequence of length n. In [10] the authors presented a method based on Fourier transformation. A different method introduced in [11] employs data compression. To the best of our knowledge there is no software available implementing these two methods. The most advanced program in the field of searching with PSSMs is *EMATRIX* [12], which incorporates a technique called lookahead scoring. The lookahead scoring technique is also employed in the suffix tree based method of [13]. This method performs a limited depth first traversal of the suffix tree of the set of target sequences. This search updates PSSM-scores along the edges of the suffix tree. Lookahead scoring allows to skip subtrees of the suffix tree that do not contain any matches to the PSSM. Unfortunately, the method of [13] has not found its way into a widely available and robust software system. In [14], the development of new, more efficient algorithms for searching with PSSMs is considered an important problem, which still needs better solutions.

In this paper, we present a new, non-heuristic algorithm for searching with PSSMs. With any non-heuristic PSSM searching algorithm, the performance in terms of sensitivity and specificity solely depends on the used PSSM and threshold, i.e. given a PSSM and threshold, all these algorithms give exactly the same results. For the generation of PSSMs from aligned sequences, numerous different methods, were described in literature over the last decades [1,5,15-17]. Rather than improving PSSMs, we focus on improvements in terms of time and space efficiency when searching with PSSMs, independently of their underlying generation method. The overall structure of our proposed search algorithm is similar to the method of [13]. However, instead of suffix trees we use enhanced suffix arrays, a data structure which is as powerful as suffix trees (cf. [18]). Enhanced suffix arrays provide several advantages over suffix trees, which make them more suitable for searching with PSSMs:

- While suffix trees require about $12n$ bytes in the best available implementation (cf. [19]), the enhanced suffix array used for searching with PSSMs only needs $9n$ bytes of space.

- While the suffix tree is usually only computed in main memory, the enhanced suffix array is computed once and stored on file. Whenever a PSSM is to be searched, the enhanced suffix array is mapped into main memory which requires no extra time.

- While the depth first traversal of the suffix tree suffers from the poor locality behavior of the data structure (cf. [20]), the enhanced suffix array provides optimal locality, because when searching with PSSMs it is sequentially scanned from left to right.

One of the algorithmic contributions of this paper is a new technique that allows to skip parts of the enhanced suffix array containing no matches to the PSSM. Due to the skipping, our algorithm achieves an expected running time that is sublinear in the size of the search space (i.e., the size of the nucleotide or protein database). As a consequence, our algorithm scales very well for large data sizes.

Since the running time of our algorithm increases with the size of the underlying alphabet, we developed a filtering technique, utilizing alphabet reduction, that achieves better performance especially on sequences/PSSMs over the amino acid alphabet.

When searching with a PSSM, it is important to determine a suitable threshold for a PSSM-match. Usually, the user prefers to specify a significance threshold (i.e., an E-value or a p-value) which has to be transformed into an absolute score threshold for the PSSM under consideration. This can be done by computing the score distribution of the PSSM, using well-known dynamic programming (DP, for short) methods, e.g., [12,21-23]. Unfortunately, these methods are not fast enough for large PSSMs. For this reason, we have developed a new, lazy evaluation algorithm that only computes a small fraction of the complete score distribution. Our algorithm speeds up the computation of the threshold by factor of at least 3, compared to standard DP methods. This makes our algorithm applicable for on-the-fly computations of the score thresholds.

The new algorithms described in this paper are implemented as part of the *PoSSuM* software distribution. This is available free of charge for non-commercial research institutions. For details, see [24]. Parts of this contribution appeared as [25] in proceedings of GCB2004.

Results

PSSMs and lookahead scoring: LAsearch

A PSSM is an abstraction of a multiple alignment of related sequences. We define it as a function $M : [0, m - 1] \times \mathcal{A} \to \mathbb{R}$, where m is the length of M and $\mathcal{A}$ is a finite alphabet. Usually M is represented by an $m \times |\mathcal{A}|$ matrix, see Figure 1 for an example. Each row of the matrix reflects the frequency of occurrence of each amino acid or nucleotide at the corresponding position of the alignment. From now on, let M be a PSSM of length m and let $w[i]$ denote the character of w at position i for $0 \leq i < m$. Further on, $w[i..j]$ denotes the string starting at position i and ending at position j. We define $sc(w,M) := \sum_{i=0}^{m-1} M(i,w[i])$ for a sequence $w \in \mathcal{A}^m$ of length m. $sc\ (w, M)$ is the *match score of w* w.r.t. M. The *score range* of a PSSM is the interval $[sc_{\min}(M), sc_{\max}(M)]$ with

$sc_{\min}(M) := \sum_{i=0}^{m-1} \min\{M(i,a) \mid a \in \mathcal{A}\}$ and

$sc_{\max}(M) := \sum_{i=0}^{m-1} \max\{M(i,a) \mid a \in \mathcal{A}\}$. Given a sequence S of length n over alphabet $\mathcal{A}$ and a score threshold th, the *PSSM matching problem* is to find all positions $j \in [0, n - m]$ in S and their assigned match scores, such that $sc\ (S[j..j + m - 1], M) \geq th$.

A simple algorithm for the PSSM matching problem slides along the sequence and computes $sc\ (w, M)$ for each $w = S[j..j + m - 1]$, $j \in [0, n - m]$. The running time of this algorithm is $O(mn)$. It is used e.g., in the programs *FingerPrintScan* [8], *BLIMPS* [4], *MatInspector*[9], and *MATCH* [17].

In [12], lookahead scoring is introduced to improve the simple algorithm. Lookahead scoring allows to stop the calculation of $sc\ (w, M)$ when it is clear that the given overall score threshold th cannot be achieved. To be more precise, we define $pfxsc_d(w,M) := \sum_{h=0}^{d} M(h,w[h])$,

$max_d := \max\{M(d, a) \mid a \in \mathcal{A}\}$, and $\sigma_d := \sum_{h=d+1}^{m-1} max_h$ for any $d \in [0, m - 1]$. $pfxsc_d(w, M)$ is the *prefix score of depth d*. σ_d is the maximal score that can be achieved in the last $m - d - 1$ positions of the PSSM. Let $th_d := th - \sigma_d$ be the *intermediate threshold* at position d. The correctness of lookahead scoring, not shown in [12], is implied by the following Lemma:

Lemma 1 The following statements are equivalent:

(1) $pfxsc_d(w, M) \geq th_d$ for all $d \in [0, m - 1]$,

(2) $sc(w, M) \geq th$.

Proof: (1)⇒(2): Suppose that (1) holds. Then $\sigma_{m-1} = \sum_{h=m}^{m-1} max_h = 0$ and

$$sc(w,M) = \sum_{h=0}^{m-1} M(h,w[h]) = pfxsc_{m-1}(w,M) \geq th_{m-1} = th - \sigma_{m-1} = th.$$

(2)⇒(1): Suppose that (2) holds. Let $d \in [0, m - 1]$. Then

$$sc(w,M) = \sum_{h=0}^{m-1} M(h,w[h]) = \sum_{h=0}^{d} M(h,w[h]) + \sum_{h=d+1}^{m-1} M(h,w[h])$$
$$= pfxsc_d(w,M) + \sum_{h=d+1}^{m-1} M(h,w[h])$$

A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y	th_d	σ_d
-19	**92**	-45	-49	-30	-36	-38	-12	-41	-21	-22	-40	-46	-44	-44	-30	-25	16	-35	-34	**2**	**398**
5	-17	17	**22**	-28	-15	-7	-23	-8	-27	-21	21	18	-7	-13	-9	9	-19	-33	-25	**24**	**376**
7	-8	-29	-28	2	-25	-10	25	-23	-4	-5	-25	-32	-26	-25	-18	13	22	-11	**36**	**60**	**340**
-29	**99**	-55	-61	-42	-45	-47	-31	-52	-34	-36	-49	-56	-55	-55	-38	-35	-29	-44	-46	**159**	**241**
-14	-22	14	**22**	-28	9	-8	-26	15	-27	-20	-7	-26	-3	31	-13	5	-23	-30	-24	**181**	**219**
-25	-34	-25	-16	-37	-30	-15	-36	45	-34	-26	-18	-35	-9	**49**	-25	-26	-33	-39	-31	**230**	**170**
7	-8	-25	-24	-19	-23	-22	4	-15	-10	-8	-19	-29	-21	11	-13	**31**	31	-31	-22	**261**	**139**
-34	-27	-44	-43	50	-41	-8	-16	-38	-14	-17	-39	-51	-40	-36	-39	-35	-21	-1	**56**	**317**	**83**
7	**40**	-16	-14	-9	-14	-6	-17	14	-20	-15	-10	-24	-11	12	15	9	-13	-16	20	**357**	**43**
-7	**43**	16	-7	-27	-15	-9	-24	-5	-26	-18	-6	-25	25	13	25	-8	-21	-30	-24	**400**	**0**

Figure 1

Amino acid PSSM. Amino acid PSSM of length $m = 10$ of a zinc-finger motif. If the score threshold is $th = 400$, then only substrings beginning with *C* or *V* can match the PSSM, because all other amino acids score below the intermediate threshold $th_0 = th - \sigma_0 = 400 - 398 = 2$. That is, lookahead scoring will skip over all substrings which begin with amino acids different from *C* and *V*. Here σ_d, $d \in [0, m - 1]$ denotes the maximal score that can be achieved in the last $m - d - 1$ positions of the PSSM as defined in the text.

Hence $sc(w, M) \ge th$ implies $pfxsc_d(w,M) + \sum_{h=d+1}^{m-1} M(h,w[h]) \ge th$. Since $M(h, w[h]) \le max_h$ for $h \in [0, m - 1]$, we conclude

$$\sum_{h=d+1}^{m-1} M(h,w[h]) \le \sum_{h=d+1}^{m-1} max_h = \sigma_d$$

and hence

$$pfxsc_d(w,M) \ge th - \sum_{h=d+1}^{m-1} M(h,w[h]) \ge th - \sigma_d = th_d.$$

The Lemma suggests a necessary condition for a PSSM-match which can easily be exploited: When computing $sc(w, M)$ by scanning w from left to right, one checks for $d = 0,1,\ldots, m - 1$, if the intermediate threshold th_d is achieved. If not, the computation can be stopped. See Figure 1 for an example of intermediate thresholds and their implications.

The lookahead scoring algorithm (herein after called *LAsearch*) runs in $O(kn)$ time, where k is the average number of PSSM-positions per sequence position actually evaluated. In the worst case, $k \in O(m)$, which leads to the worst case running time of $O(mn)$, not better than the simple algorithm. However, k is expected to be much smaller than m, leading to considerable speedups in practice.

Our reformulation of lookahead scoring and its implementation is the basis for improvements and evaluation in the subsequent sections.

PSSM searching using enhanced suffix arrays: ESAsearch

The enhanced suffix array for a given sequence S of length n consists of three tables suf, lcp, and skp. Let \$ be a symbol in $\mathcal{A}$, larger than all other symbols, which does not occur in S. suf is a table of integers in the range 0 to n, specifying the lexicographic ordering of the $n + 1$ suffixes of the string $S\$$. That is, $S_{\mathrm{suf}[0]}, S_{\mathrm{suf}[1]}, \ldots, S_{suf[n]}$ is the sequence of suffixes of $S\$$ in ascending lexicographic order, where $S_i = S[i..n - 1]\$$ denotes the i-th nonempty suffix of the string $S\$$, for $i \in [0, n]$. See Figure 2 for an example. suf can be constructed in $O(n)$ time [26] and requires $4n$ bytes.

lcp is a table in the range 0 to n such that lcp[0] := 0 and lcp[i] is the length of the longest common prefix of $S_{\mathrm{suf}[i-1]}$ and $S_{\mathrm{suf}[i]}$, for $i \in [1, n]$. See Figure 2 for an example. Table lcp can be computed in linear time given table suf [27]. In practice PSSMs are used to model relatively short, local motifs and hence do not exceed length 255. For searching with PSSMs we therefore do not access values in table lcp larger than 255, and hence we can store lcp in n bytes.

skp is a table in the range 0 to n such that skp[i] := min($\{n + 1\} \cup \{j \in [i + 1, n] \mid \mathrm{lcp}[i] > \mathrm{lcp}[j]\}$). In terms of suffix trees, skp[i] denotes the lexicographically next leaf that does not occur in the subtree below the branching node corresponding to the longest common prefix of $S_{\mathrm{suf}[i-1]}$ and $S_{\mathrm{suf}[i]}$. Figure 2 shows this relation. Table skp can be computed in $O(n)$ time given suf and lcp. For the algorithm to be described we assume that the enhanced suffix array for S has been precomputed.

In a suffix tree, all substrings of S of a fixed length m can be scored with a PSSM by a depth first traversal of the tree. Using lookahead scoring, one can skip certain subtrees that do not contain matches to the PSSM. Since suffix trees have several disadvantages (see the introduction), we use enhanced suffix arrays to search PSSMs. Like in other algorithms on enhanced suffix arrays (cf. [18]), one simulates a depth first traversal of the suffix tree by processing the arrays suf and lcp from left to right. To incorporate lookahead scoring while searching we must be able to skip certain ranges of suffixes in suf. To facilitate this, we use table skp. We will now make this more precise.

For $i \in [0, n]$, let $v_i = S_{\mathrm{suf}[i]}$, $l_i = \min\{m, |v_i|\} - 1$, and $d_i = \max(\{-1\} \cup \{d \in [0, l_i] \mid pfxsc_d(v_i, M) \ge th_d\})$. Now observe that $d_i = m - 1 \Leftrightarrow pfxsc_{m-1}(v_i, M) \ge th_{m-1} \Leftrightarrow sc(v_i, M) \ge th$. Hence, M matches at position j = suf[i] if and only if $d_i = m - 1$. Thus, to solve the PSSM searching problem, it suffices to compute all $i \in [0, n]$ satisfying $d_i = m - 1$. We compute d_i along with $C_i[d] = pfxsc_d(v_i, M)$ for any $d \in [0, d_i]$. d_0 and C_0 are easily determined in $O(m)$ time. Now let $i \in [1, n]$ and suppose that d_{i-1} and $C_{i-1}[d]$ are determined for $d \in [0, d_{i-1}]$. Since v_{i-1} and v_i have a common prefix of length lcp[i], we have $C_i[d] = C_{i-1}[d]$ for all $d \in [0, \mathrm{lcp}[i] - 1]$. Consider the following cases:

- If $d_{i-1} + 1 \ge \mathrm{lcp}[i]$, then compute $C_i[d]$ for $d \ge \mathrm{lcp}[i]$ while $d \le l_i$ and $C_i[d] \ge th_d$. We obtain $d_i = d$.

- If $d_{i-1} + 1 < \mathrm{lcp}[i]$, then let j be the minimum value in the range $[i + 1, n + 1]$ such that all suffixes $v_i, v_{i+1},\ldots,v_{j-1}$ have a common prefix of length $d_{i-1} + 1$ with v_{i-1}. Due to the common prefix we have $pfxsc_d(v_{i-1}, M) = pfxsc_d(v_r, M)$ for all $d \in [0, d_{i-1} + 1]$ and $\mathrm{r} \in [i, j - 1]$. Hence $d_{i-1} = d_r$ for $\mathrm{r} \in [i, j - 1]$. If $d_{i-1} = m - 1$, then there are PSSM matches at all positions suf[r] for $r \in [i, j - 1]$. If $d_{i-1} < m - 1$, then there are no PSSM matches at any of these positions. That is, we can directly proceed with index j. We obtain j by following a chain of entries in table skp: compute a sequence of values $j_0 = i$, $j_1 = \mathrm{skp}[j_0],\ldots,j_k = \mathrm{skp}[j_{k-1}]$ such that $d_{i-1} + 1 < \mathrm{lcp}[j_1],\ldots,d_{i-1} + 1 < \mathrm{lcp}[j_{k-1}]$, and $d_{i-1} + 1 \ge \mathrm{lcp}[j_k]$. Then $j = j_k$.

i	suf[i]	lcp[i]	skp[i]	$S_{suf[i]}$
0	1	0	12	aaaaccacac$
1	2	3	2	aaaccacac$
2	3	2	3	aaccacac$
3	7	1	6	acac$
4	4	2	6	accacac$
5	9	2	6	ac$
6	0	0	12	caaaaccacac$
7	6	2	9	cacac$
8	8	3	9	cac$
9	5	1	11	ccacac$
10	10	1	11	c$
11	11	0	12	$

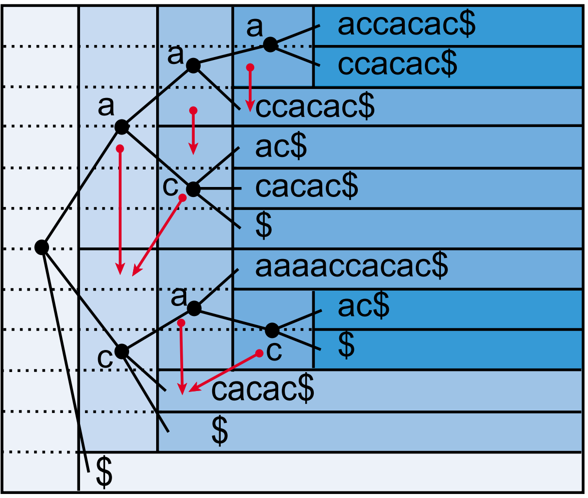

Figure 2

Relationship between enhanced suffix array and suffix tree. The enhanced suffix array consisting of tables suf, lcp, skp (left) and the suffix tree (right) for sequence *S* = caaaaccacac. Some skp entries are shown in the tree as red arrows: If skp[*i*] = *j*, then an arrow points from row *i* to row *j*. For clarity, suffixes corresponding to suf[*i*] are given in table $S_{\text{suf}[i]}$.

These case distinctions lead to the program *ESAsearch* (see Figures 3, 4).

We illustrate the ideas of algorithm *ESAsearch*, formally described above, with the following example. Let *M* be a PSSM of length $m = 2$ over alphabet $\mathcal{A} = \{a, c\}$ with $M(0, a) = 1$, $M(0, c) = 3$, $M(1, a) = 3$, and $M(1, c) = 2$. For a given threshold of $th = 6$ we obtain intermediate thresholds $th_0 = 3$ and $th_1 = 6$. To search with *M* in the enhanced suffix array for sequence *S* = caaaaccacac as given in Figure 2, we start processing the enhanced suffix array suf top down by scoring the first suffix $S_{\text{suf}[0]}$ = aaaaccacac$ with *M* from left to right. For the first character of $S_{\text{suf}[0]}$ we obtain a score of $pfxsc_0(S_{\text{suf}[0]},M)= M(0, a) = 1$ which is below the first intermediate threshold $th_0= 3$. Hence we set $d_0 = -1$ and notice that we can skip all suffixes of *S* that start with character 'a'. Further on, with a lookup in lcp[1] = 3 we find that $S_{\text{suf}[1]}$ and $S_{\text{suf}[0]}$ share a common prefix of length 3 and $d_0 + 1 = -1 + 1 <$ lcp[1] = 3 (second case described above). The next suffix that may match *M* with $th = 6$ is $S_{\text{suf}[6]}$ = caaaaccacac$. Suffixes $S_{\text{suf}[1]}, S_{\text{suf}[2]}, \ldots S_{\text{suf}[5]}$ can be skipped since they all share a common prefix with $S_{\text{suf}[0]}$ of at least length 1. That is, they begin all with character 'a' and would also miss the first intermediate threshold $th_0 = 3$ when scored. We find $S_{\text{suf}[6]}$ by following a chain of entries in table skp: skp[1] = 2, skp[2] = 3, and skp[3] = 6. When scoring $S_{\text{suf}[6]}$ we compute $pfxsc_0(S_{\text{suf}[6]}, M) = M(0,c) = 3$ and $pfxsc_1(S_{\text{suf}[6]}, M) = M(0, c) + M(1, a) = 6$ and store them for reuse in C[0] and C[1]. Since $d_6 = 1 = m - 1 = 1$ holds, we report suf[6] = 0 with score $sc(S_{\text{suf}[6]}, M) = pfxsc_1(S_{\text{suf}[6]},M) = 6$ as a matching position. With lookups in lcp[7] = 2 and lcp[8] = 3 we notice that $S_{\text{suf}[7]}$ and $S_{\text{suf}[8]}$ share a common prefix of at least two characters with $S_{\text{suf}[6]}$. Hence we report suf[7] = 6 and suf[8] = 8 with score C[1] = 6 as further matching positions. We proceed with the scoring of $S_{\text{suf}[9]}$. Since lcp[9] = 1 holds, we obtain the score for the first character 'c' from array *C* with $pfxsc_0(S_{\text{suf}[9]}, M)$ = C[0]. After scoring the second character of $S_{\text{suf}[9]}$, $pfxsc_1(S_{\text{suf}[9]}, M) = 5 < th_1 = 6$ holds and we miss the second intermediate threshold and continue with the next suffix. The last two suffixes $S_{\text{suf}[10]}$ and $S_{\text{suf}[11]}$ in suf do not have to be considered since their lengths are smaller than to $m = 2$ (not counting the sentinel character $) and therefore they cannot match *M*. We end up with matching positions 0, 6, and 8 of *M* in *S* with match score 6. To find these matches, we processed the enhanced suffix array suf top down and scored suffixes from left to right, facilitating the additional information given in tables lcp and skp to avoid rescoring of characters of common prefixes of suffixes and to skip suffixes that cannot match *M* for the given threshold.

Analysis

The C_i arrays can be stored in a single $O(m)$ space array *C* as any step *i* only needs the C_i specific to that step. C_i solely depends on C_{i-1}, and $C_i[0..d - 1] = C_{i-1}[0..d - 1]$ holds for a certain $d<m$, i.e., the first *d* entries in C_i are known from the previous step, and thus *C* can be organized as a stack.

```
Algorithm 1: ESAsearch
input  : An enhanced suffix array for sequence $S\$$ consisting of tables suf, lcp and skp,
         a PSSM $M$ of length $m$, a threshold $th$, and intermediate thresholds $th_d$,
         $0 \le d < m$.
output: All matching positions of $M$ in $S$ and their associated matchscores.

depth ← 0;
i ← 0;
while $i < n$ do
    if $n - m < \mathsf{suf}[i]$ then
        while $(n - m < \mathsf{suf}[i]) \wedge (i < n)$ do
            $i \leftarrow i + 1$;
            $depth \leftarrow min\{depth, \mathsf{lcp}[i]\}$;
        end
        if $i \ge n$ then return ;
    end
    if $depth = 0$ then $score \leftarrow 0$ else $score \leftarrow C[depth - 1]$;
    $d \leftarrow depth - 1$;
    do
        $d \leftarrow d + 1$;
        $score \leftarrow score + M(d, S_{\mathsf{suf}[i]+d})$;
        $C[d] \leftarrow score$;
    while $(d < m - 1) \wedge (score \ge th_d)$;
    if $(d = m - 1) \wedge (score \ge th)$ then
        print "match at position $\mathsf{suf}[i]$ with score: $score$";
        while $i < n$ do
            $i \leftarrow i + 1$;
            if $\mathsf{lcp}[i] \ge m$ then print "match at position $\mathsf{suf}[i]$ with score: $score$" else
            break;
        end
    else
        $i \leftarrow skipchain(\mathsf{lcp}, \mathsf{skp}, n, i, d)$;
    end
    $depth \leftarrow \mathsf{lcp}[i]$;
end
```

Figure 3

Algorithm ESAsearch. The algorithm *ESAsearch* formulated in pseudocode. See text for detailed explanations of the used notions.

No other space (apart from the space for the enhanced suffix array) depending on input size is required for *ESAsearch*, leading to an $O(m)$ space complexity.

The worst case for *ESAsearch* occurs, if $th \le sc_{\min}(M)$ (M matches at each position in S), and no suffix of S shares a common prefix with any other suffix. In this case looka-

```
Function skipchain(lcp, skp, $n$, $i$, $d$)
input  : Tables lcp and skp of an enhanced suffix array, $|S|$ denoted with $n$, an index $i$
         of the $i$-th smallest suffix, and depth $d$ from where to start skipping.
output : An index $j$ of the $j$-th smallest suffix with $j > i$.
begin
    if $i < n$ then
        $j \leftarrow i + 1$;
        while $(j \leq n) \wedge (\mathsf{lcp}[j] > d)$ do
            $j \leftarrow \mathsf{skp}[j] + 1$;
        end
    else
        $j \leftarrow n$;
    end
    return $j$;
end
```

Figure 4
Function skipchain of the ESAsearch algorithm. Function skipchain computes a chain of entries in table skp to skip certain ranges of suffixes in table suf.

head scoring does not give any speedup and every suffix must be read up to depth to m, leading to an $O(nm)$ worst case time complexity. This is not worse but also not better than the complexity for *LAsearch*. Next we show that, *independent* of the chosen threshold *th*, the overall worst case running time boundary for *ESAsearch* drops to $O(n + m)$ under the assumption that

$$n \geq |\mathcal{A}|^m + m - 1 \qquad (1)$$

holds.

The shorter the common prefixes of the neighboring suffixes, the slower *ESAsearch* runs. Thus to analyze the worst case, we have to consider sequences containing as many different substrings of some length q as possible. Observe that a sequence can contain at most $|\mathcal{A}|^q$ different substrings of length $q > 0$, independent of its length. To analyze the behavior of *ESAsearch* on such a sequence, we introduce the concept of suffix-intervals on enhanced suffix arrays, similar to lcp-intervals as used in [18].

Definition 1 An interval $[i, j]$, $0 \leq i \leq j \leq n$, is a *suffix-interval* with offset $\ell \in \{0,..., n\}$, or *ℓ-suffix-interval*, denoted ℓ-$[i, j]$, if the following three conditions hold:

1. $\mathrm{lcp}[i] < \ell$

2. $\mathrm{lcp}[j + 1] < \ell$

3. $\mathrm{lcp}[k] \geq \ell$ for all $k \in \{x \mid i + 1 \leq x \leq j\}$

An *lcp-interval*, or ℓ-interval, with lcp-value $\ell \in \{0,..., n\}$ is a suffix-interval ℓ - $[i, j]$ with $i <j$ and $\mathrm{lcp}[k] = \ell$ for at least one $k \in \{i + 1,..., j\}$.

Every lcp-interval ℓ - $[i, j]$ of an enhanced suffix array for text S corresponds to an internal node v in a suffix tree for S, and the length of the string spelled out by the edge labels on the path from the root node to v is equal to ℓ. Leaves are represented as singleton intervals, ℓ - $[i, j]$ with $i = j$. We say that suffix-interval ℓ - $[i, j]$ embeds suffix-interval ℓ^+ - $[k, l]$, if and only if $\ell^+ > \ell$, $i \leq k <l \leq j$, and if there is no suffix-interval ℓ' - $[r, s]$ with $\ell < \ell' < \ell^+$ and $i \leq r \leq k <l \leq s \leq j$. As an example for ℓ-suffix-intervals, consider the enhanced suffix array given in Figure 2. [0, 5] is a 1-suffix-interval, because $\mathrm{lcp}[0] = 0 < 1$, $\mathrm{lcp}[5 + 1] = 0 < 1$, and $\mathrm{lcp}[k] \geq 1$, for all k, $1 \leq k \leq 5$. Suffix-interval 2-[3,5] is embedded in 1-[0,5], but 3-[0,1] is not. Consider an enhanced suffix array of a sequence which contains all possible substrings of length q. There are $|\mathcal{A}|$ 1-suffix-intervals, $|\mathcal{A}|^2$ 2-suffix-intervals, and so on. Consequently, up to depth q, there are a total of

$$E_q = \sum_{i=1}^{q} |\mathcal{A}|^i = \frac{|\mathcal{A}|^{q+1} - |\mathcal{A}|}{|\mathcal{A}| - 1} \qquad (2)$$

ℓ-suffix-intervals ($1 \leq \ell \leq q$). This corresponds to the number of internal nodes and leaves in a suffix tree, which is atomic up to at least depth q under our assumptions.

Since we are considering sequences that contain all possible substrings of length q, there are $|\mathcal{A}|^d d$-suffix-intervals at any depth d, $1 \leq d \leq q$. Let d-$[i, j]$ be a d-suffix-interval. We know that $pfxsc_d(v_i, M)$ is a partial sum of $pfxsc_q(v_i, M)$, and because $v_i[0..\, d - 1] = v_{i+1}[0..d - 1] = ... = v_j[0..\, d - 1]$, $pfxsc_d(v_i, M)$ is also a partial sum of $pfxsc_q(v_k, M)$ for $i \leq k \leq j$. That is, after *ESAsearch* has calculated $pfxsc_d(v_i, M)$ at depth d, at any suffix-interval $(d + 1)$ - $[r, s]$ embedded in d-$[i, j]$ it suffices to only calculate the "rest" of $pfxsc_q(v_k, M)$. At any depth d, the algorithm calculates $pfxsc_{d+1}(v_r, M) = pfxsc_d(v_i, M) + M(d, v_r[d])$, meaning that all prefix scores at depth $d + 1$ in a d-suffix-interval can be computed from the prefix scores at depth d by $|\mathcal{A}|$ matrix look-ups and additions as there are $|\mathcal{A}|$ embedded $(d + 1)$-suffix-intervals. There are $|\mathcal{A}|^d$ d-suffix-intervals at depth d. Hence, it takes *ESAsearch* a total of $|\mathcal{A}|^d \cdot |\mathcal{A}|$ matrix look-ups and additions to advance from depth d to $d + 1$, and thus we conclude that the algorithm requires a total of $O(E_q)$ operations to compute all scores for all substrings of length q.

Suppose that *ESAsearch* has read suffix v_i in some step up to depth q - 1 such that character $v_i[q - 1]$ is the last one read. If $\text{lcp}[i + 1] \geq q$ holds, then the algorithm has found a suffix-interval q-$[i, j]$ with a yet unknown right boundary j, otherwise $j = i$. *ESAsearch* reports all $\text{suf}[k]$ with $k \in [i, j]$ as matching positions by scanning over table lcp starting at position i until $\text{lcp}[k] < \text{lcp}[i]$ (such that it finds $j = k - 1$), and continues with suffix v_k at depth $\text{lcp}[k]$. Hence processing such a suffix-interval requires one matrix look-up and addition to compute the score, and $j - i + 1$ steps to report all matches and find suffix v_k. Since suffix-intervals do not overlap, the total length of all suffix-intervals at depth q can be at most n, so the total time spent on reporting matches is bounded by n.

There are three cases to consider when determining the time required for calculating the match scores for a PSSM of length m. Let $p : = m - q$.

1. If $p = 0$ ($\Rightarrow m = q$), then the time required to calculate all match scores is in $O(E_q)$ as discussed above.

2. If $p < 0$ ($\Rightarrow m<q$), then none of the m-suffix-intervals are singletons since we assumed that the sequence under consideration contains all possible substrings of length q, i.e., there must be suffixes sharing a common prefix of length m, and the time required to calculate all match scores is in $O(E_m)$.

3. If $p > 0$ ($\Rightarrow m > q$), then every m-suffix-interval can be a singleton, and all prefix scores for the PSSM prefix of length q are calculated in $O(E_q)$ time. However, the remaining scores for the pending substrings of length p must be computed for *every* suffix longer than q, taking $O(np)$ additional time, and leading to a total $O(E_q + np)$ worst case time complexity for computing all match scores.

Note that a text containing $|\mathcal{A}|^q$ different substrings must have a certain length, which must be at least $|\mathcal{A}|^q$. In fact, a minimum length text that contains all strings of length q has length $n = |\mathcal{A}|^q + q - 1$. It represents a *de Bruijn sequence*[28] without wrap-around, i.e., a *de Bruijn* sequence with its first q - 1 characters concatenated to its end. Since a *de Bruijn* sequence without wrap-around represents the minimum length worst case, we infer from Equation (2) that $E_q \in O(n)$. Hence, if $m = q$, then it takes $O(n)$ time to calculate all match scores. If $m <q$, then $E_m <E_q$ and thus it takes sublinear time. If $m > q$, it takes $O(n + np)$ time.

We summarize the worst case running time of *ESAsearch* for preprocessing a PSSM M of length m, searching with M, and reporting all matches with their match scores, as

$$O(n + n \cdot \max\{0, p\} + m).$$

Hence, the worst case running time is $O(n + m)$ for $p \leq 0$, implying that this time complexity holds for any PSSM of length m and threshold on any text of length $n \geq |\mathcal{A}|^m + m - 1$, as already stated in Inequality (1).

In practice, large numbers of suffixes can be skipped if the threshold is stringent enough, leading to a total running time *sublinear* in the size of the text, regardless of the relation between n and m. *ESAsearch* reads a suffix up to depth m unless an intermediate score falls short of an intermediate threshold, and skips intervals with the same or greater lcp if this happens. Right boundaries of skipped suffix-intervals are found quickly by following the chain of skip-values (see function skipchain in Figure 4). It are these jumps that make *ESAsearch* superior in terms of running time to *LAsearch* in practice. The best case is indeed $O(|\mathcal{A}|)$ which occurs whenever there is no score in the first row of the PSSM that is greater than th_0.

See Figure 5 for examples of enhanced suffix arrays, constructed from texts S and T that consist of all strings of a certain length m over some alphabet. In these enhanced suffix arrays no suffix shares a prefix of length m with any other suffix, forcing *ESAsearch* to compute scores for each suffix. But with the intermediate scores available while processing the suffixes, it takes exactly E_m steps to compute the scores, as can be figured out by manually applying *ESAsearch* to the depicted enhanced suffix arrays. For *S*, exactly $\frac{4^3 - 4}{4 - 1} = 20$, for *T*, exactly $\frac{2^4 - 2}{2 - 1} = 14$ operations are needed to compute *all* $|\mathcal{A}|^m \le n - m + 1$ possible scores (and to find all matches since *S* and *T* are both *de Bruijn* sequences without wrap-around). Only a single match is reported per matching substring, leading to $E_m \in O(n)$ operations to be performed during the search phase.

Performance improvements via alphabet transformations

Inequality (1) provides the necessary condition for $O(n + m)$ worst case running time. We now assume that m in Inequality (1) identifies not the length of a PSSM, but the threshold dependent *expected reading depth* for some PSSM. We denote this expected depth by $m^*(th) \le m$ and continue denoting the PSSM's length by m. As seen before, for PSSMs with length m, such that $p = m - m^*(th)$, the worst case running time is $O(n + n \cdot \max\{0, p\} + m)$, but the expected running time is $O(n + m)$, as on average we expect $p \le 0$. Inequality (1) with m substituted by $m^*(th)$ implies $\log_{|\mathcal{A}|}(n) \ge m^*(th)$. That is, to achieve linear worst case running time for the amino acid alphabet, $m^*(th)$ needs to be very small. For instance, if $n = 20^7$, then the search time is guaranteed to be linear in n only for PSSMs with a maximum length of 7, and expected to be linear for PSSMs with expected reading depth of 7. Observe that for $|\mathcal{A}| = 4$, $m^*(th)$ needs to be smaller or equal to 15 to achieve linear or sublinear running times. This provides the motivation to reduce the alphabet size by transforming $\mathcal{A}$ into a reduced size $\widehat{\mathcal{A}}$ such that $|\widehat{\mathcal{A}}| < |\mathcal{A}|$.

In practice, for reasonably chosen thresholds *th*, the performance of *ESAsearch* mainly depends on the fact that often large ranges of suffixes in the enhanced suffix array can be skipped. This is always the case if we drop below an intermediate threshold while calculating a prefix' score, and if that prefix is a common prefix of other suffixes. In terms of lcp-intervals, this means that we can skip all ℓ-intervals with $\ell \ge m^*(th)$ on average. In contrast to *suffix-intervals*, whose total count is in $O(n^2)$, size and number of *lcp-intervals* depend on $|\mathcal{A}|$, as illustrated in Figure 6. We

i	suf$[i]$	lcp$[i]$	$S_{\mathsf{suf}[i]}$
0	5	0	aaccgtcttggc$
1	6	1	accgtcttggc$
2	1	1	agataaccgtcttggc$
3	3	1	ataaccgtcttggc$
4	0	0	cagataaccgtcttggc$
5	7	1	ccgtcttggc$
6	8	1	cgtcttggc$
7	11	1	cttggc$
8	16	1	c$
9	2	0	gataaccgtcttggc$
10	15	1	gc$
11	14	1	ggc$
12	9	1	gtcttggc$
13	4	0	taaccgtcttggc$
14	10	1	tcttggc$
15	13	1	tggc$
16	12	1	ttggc$
17	17	0	$

i	suf$[i]$	lcp$[i]$	$T_{\mathsf{suf}[i]}$
0	2	0	aaacaccc$
1	3	2	aacaccc$
2	4	1	acaccc$
3	6	2	accc$
4	1	0	caaacaccc$
5	5	2	caccc$
6	0	1	ccaaacaccc$
7	7	2	ccc$
8	8	2	cc$
9	9	1	c$
10	10	0	$

Figure 5
Minimum size enhanced suffix arrays for worst case analysis. Enhanced suffix arrays for text *S* = cagataaccgtcttggc, consisting of all strings of length *m* = 2 over an alphabet of size 4, and *T* = ccaaacaccc, consisting of all strings of length *m* = 3 over an alphabet of size 2.

observe that smaller alphabet sizes imply (1) larger ℓ-intervals, and (2) an increasing number of ℓ-intervals for larger values of ℓ. Thus, by using reduced alphabets, we expect to skip larger and touch fewer lcp-intervals under the assumption that the average reading depth remains unchanged. Consequently, we expect to end up with an improved performance of *ESAsearch*. This raises the question for a proper reduction strategy for larger alphabets like the amino acid alphabet, and how this strategy can be incorporated into *ESAsearch*.

We now describe how to take advantage of reduced alphabets as fast filters in the *ESAsearch* algorithm. Let $\mathcal{A} = \{a_0, a_1,..., a_k\}$ and $\widehat{\mathcal{A}} = \{b_0, b_1,..., b_l\}$ be two alphabets, and $\Phi : \mathcal{A} \to \widehat{\mathcal{A}}$ a surjective function that maps a character $a \in \mathcal{A}$ to a character $b \in \widehat{\mathcal{A}}$. We call $\Phi^{-1}(b)$ the character class corresponding to b. For a sequence $S = s_1 s_2 \ldots s_n \in \mathcal{A}^n$ we denote the transformed sequence with $\hat{S} = \Phi(s_1)\,\Phi(s_2) \ldots \Phi(s_n) \in \widehat{\mathcal{A}}^{\,n}$. Along with the transformation of the sequence, we transform a PSSM such that we have a one to one relationship between the columns in the PSSM and the characters in $\widehat{\mathcal{A}}$. We define the transformed PSSM $\widehat{M}$ of M as follows:

Definition 2 Let M be a PSSM of length m over alphabet $\mathcal{A}$, and $\Phi : \mathcal{A} \to \widehat{\mathcal{A}}$ a surjective function. The transformed PSSM $\widehat{M}$ is defined as a function $\widehat{M}$: $[0, m - 1] \times \widehat{\mathcal{A}} \to \mathbb{R}$ with

$$\widehat{M}\ (i, b){:} = \max\ \{M(i, a) \mid a \in \Phi^{-1}(b)\}. \quad (3)$$

Figure 7 gives an example of the relationship between M and $\widehat{M}$. $\hat{S}$ can be easily determined from S in $O(n)$ time, $\widehat{M}$ in $O(|\mathcal{A}|m)$ time, given M. We define the set of matches to M on S and $\widehat{M}$ on $\hat{S}$, respectively, as

$$MS := \{j \in [0, n - m] \mid sc\ (S[j..j + m - 1], M) \geq th\}$$

$$\widehat{MS} := \{j \in [0, n - m] \mid sc\ (\hat{S}\,[j..j + m - 1], \widehat{M}) \geq th\}.$$

Now observe that we can use matches of $\widehat{M}$ on $\hat{S}$, for the computation of matches of M on S, since $MS \subseteq \widehat{MS}$. We prove that $MS \subseteq \widehat{MS}$ holds for all $th \in [sc_{\min}(M), sc_{\max}(M)]$ by proving the more general statement given in the following Lemma.

Lemma 2 $sc\ (w, M) \leq sc\ (\hat{w}, \widehat{M})$ holds for all $w \in \mathcal{A}^m$.

Proof:

$$sc(w,M) = \sum_{i=0}^{m-1} M(i,w[i]) \leq \sum_{i=0}^{m-1} \max\left\{M(i,a) \mid a \in \Phi^{-1}\left(\Phi(w[i])\right)\right\}$$
$$= \sum_{i=0}^{m-1} \widehat{M}\left(i,\Phi(w[i])\right) = sc\left(\hat{w},\widehat{M}\right).$$

Thus the following implications follow directly

- $sc\ (w, M) \geq th \Rightarrow sc\ (\hat{w}, \widehat{M}) \geq th$

- $i \in MS \Rightarrow i \in \widehat{MS}$

and we conclude: $MS \subseteq \widehat{MS}$ holds for $th \in [sc_{\min}(M), sc_{\max}(M)]$.

Hence we can search with $\widehat{M}$ in $\hat{S}$ prefiltering of matches to M in S, profiting of longer and larger ℓ-intervals in $\hat{S}$ by extending algorithm *ESAsearch* as follows:

(1) Transform S into $\hat{S}$ and build the enhanced suffix array for $\hat{S}$;

(2) Construct $\widehat{M}$ from M;

(3) Compute $\widehat{MS}$ by searching with $\widehat{M}$ on the enhanced suffix array of $\hat{S}$ using algorithm *ESAsearch*;

(4) For each $i \in \widehat{MS}$ re-score match with $\sigma = sc\ (S[i..i + m - 1], M)$, and report i and σ if and only if $\sigma \geq th$.

As a further consequence of Definition 2 the maximum score values in each row of M and $\widehat{M}$ and thus the intermediate thresholds remain unchanged in the transformation process. Unfortunately the necessary PSSM transformation accompanying alphabet size reduction affects the expected reading depth $m^*(th)$ in such a way that it increases with more degraded alphabets, and therefore reduces the expected performance improvement. Due to maximization according to Equation (3) the matrix values in $\widehat{M}$ increase and we expect a decreased probability

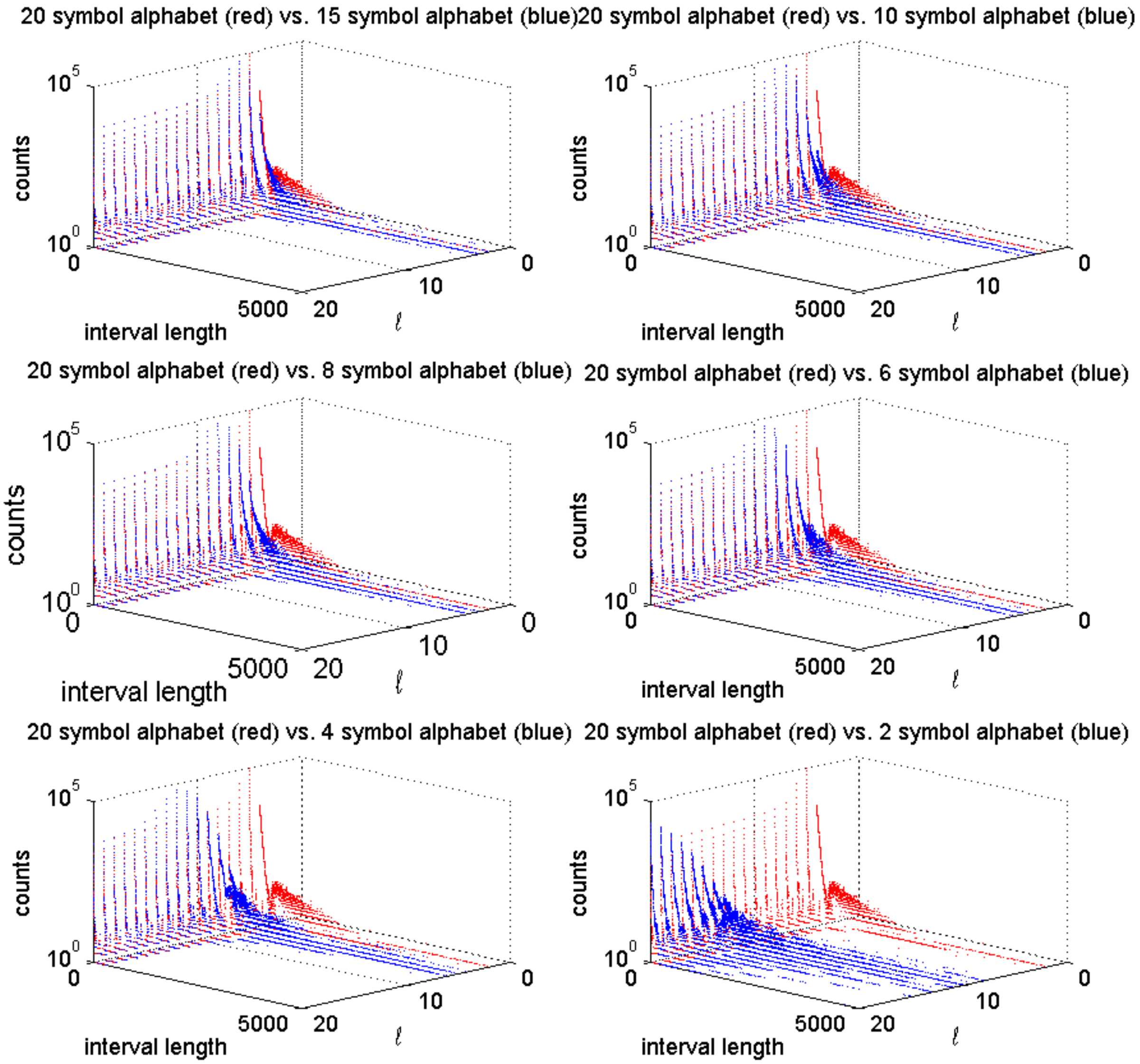

Figure 6

Number of ℓ-intervals for various reduced alphabets. Numbers of ℓ-intervals for ℓ ∈ [1, 20] of different length for various reduced alphabets. We built the enhanced suffix array with sequences from the RCSB protein data bank (PDB) (total sequence length 4,264,239 bytes). The used reduced amino acid alphabets are given in Figure 8. Note that we limited the interval lengths in the figures to 5,000 to prevent distortion.

of falling short of an intermediate threshold early. Observe that there is a trade-off between increased expected reading depth and increased lcp-interval sizes at low reading depths. Therefore it is desirable to minimize the effect of maximization by grouping PSSM columns with similar score values, i.e., highly correlated columns. Since PSSMs reflect the properties of the underlying multiple alignment, we expect correlations of PSSM columns

(A)denin	(C)ytosin	(G)uanin	(T)hymin
28.50	256.54	85.51	28.50
28.62	47.70	47.70	9.54
45.54	45.54	45.54	500.92
320.83	0.00	71.29	106.94
47.29	15.76	15.76	31.53
41.34	13.78	41.34	96.46
32.95	8.24	32.95	41.19
21.28	21.27	148.95	106.40
9.54	28.62	47.70	47.70

(P)urine	P(Y)rimidine
85.51	256.54
47.70	47.70
45.54	500.92
320.83	106.94
47.29	31.53
41.34	96.46
32.95	41.19
148.95	106.40
47.70	47.70

Figure 7

PSSM alphabet transformation. In the left PSSM *M* we used the normal four letter nucleotide alphabet $\mathcal{A}$ = {*A*, *C*, *G*, *T*} to describe a transcription factor binding site found in Hox A3 gene promoters. In the right PSSM $\widehat{M}$ we used a reduced two letter alphabet $\widehat{\mathcal{A}}$ = {*P*, *Y*} that differs only between purine (adenine or guanine) and pyrimidine (cytosine or thymine) nucleotides. Hence we have two character classes: $\Phi^{-1}(P) = \{A, G\}$ and $\Phi^{-1}(Y) = \{C, T\}$. Consequently $\widehat{M}(i, P) = \max\{M(i, a) \mid a \in \{A, G\}\}$ and $\widehat{M}(i, Y) = \max\{M(i, a) \mid a \in \{C, T\}\}\ \forall i \in [0, 8]$

according to biologically motivated symbol similarities. Hence character correlation is the motivation for our alphabet reduction strategy.

Reduced amino acid alphabets

It is well known that various of the naturally occurring amino acids share certain similarities, like similar physiochemical properties. Accordingly, the complexity of protein sequences can be reduced by sorting these amino acids with similarities into groups and deriving a transformed, reduced alphabet [29]. These reduced alphabets contain symbols that represent a specific character class of the original alphabet. Since PSSMs and the sequences to be searched have to be encoded over the same alphabet, we are more interested in a single reduced alphabet suitable for all PSSMs under consideration, than in PSSM-specific reduced alphabets. The latter implies an unacceptable overhead of index generation for sequences over PSSM-specific alphabets, even though it may result in a lower expected reading depth. The basis for our reduction of the 20-letter amino acid alphabet to smaller alphabets are correlations indicated by the BLOSUM similarity matrix as described in [30]. That is, amino acid pairs with high similarity scores are grouped together (see Figure 8 for an example). Let a and b be two amino acids and Y a 20 × 20 score matrix, then a measure of amino acid correlation $c_{a,b}$ between a and b can be defined as

$$c_{a,b} := \frac{\sum_{i=1}^{20} Y_{a,i} Y_{b,i}}{\left(\sum_{i=1}^{20} Y_{a,i}^2\right)\left(\sum_{i=1}^{20} Y_{b,i}^2\right)}$$

and amino acid pairs can be iteratively grouped together according to their correlations, starting with the most correlated pairs, until all the amino acids are divided into the desired number of groups.

Finding an appropriate threshold for PSSM searching: LazyDistrib

Probabilities and expectation values

The results of PSSM searches strongly depend on the choice of an appropriate threshold value *th*. A small threshold may produce a large number of false positive matches without any biological meaning, whereas meaningful matches may not be found if the threshold is too stringent. PSSM-scores are not equally distributed and thus scores of two different PSSMs are not comparable. It is therefore desirable to let the user define a significance threshold instead. The expected number of matches in a given random sequence database (E-value) is a widely accepted measure of the significance. We can compute the E-value for a known background distribution and length of the database by exhaustive enumeration of all substrings. However, the time complexity of such a computa-

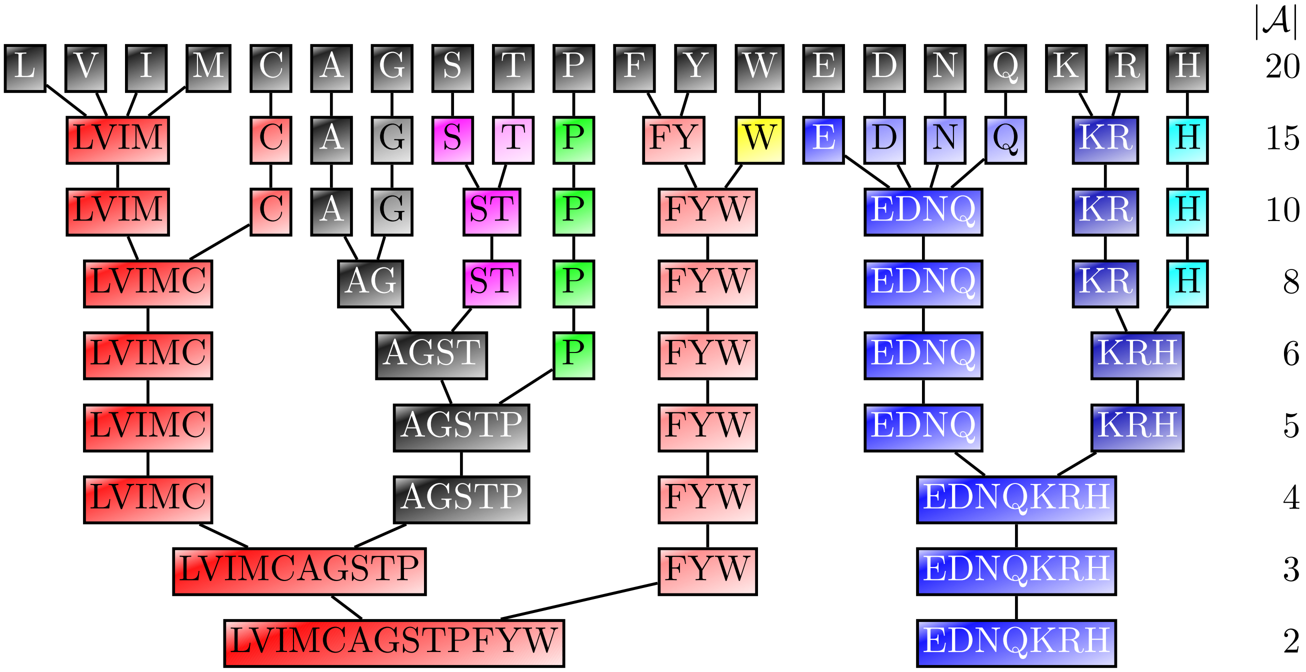
|𝒜|
L V I M C A G S T P F Y W E D N Q K R H 20
LVIM C A G S T P FY W E D N Q KR H 15
LVIM C A G ST P FYW EDNQ KR H 10
LVIMC AG ST P FYW EDNQ KR H 8
LVIMC AGST P FYW EDNQ KRH 6
LVIMC AGSTP FYW EDNQ KRH 5
LVIMC AGSTP FYW EDNQKRH 4
LVIMCAGSTP FYW EDNQKRH 3
LVIMCAGSTPFYW EDNQKRH 2

Figure 8

Schemes for amino acid alphabet reduction. Reduction of the amino acid alphabet into smaller groups. Amino acid pairs are iteratively grouped together based on ther correlations $c_{a,b}$ (see text for the definition of $c_{a,b}$), starting with the most correlated pairs, until al amino acids are divided into the desired number of groups. Here we used BLOSUM50 similarities for the determination of $c_{a,b}$. Observe that, hydrophobic amino acids, especially (LVIM) and (FYW) are conserved in many reduced alphabets. The same is true for the polar (ST), (EDNQ), and (KR) groups. The smallest alphabet contains two groups that can be categorized broadly as hydrophobic/small (LVIMCAGSTPFYW) and hydrophilic (EDNQKRH).

tion is $O(|\mathcal{A}|^m m)$ for a PSSM of length m. If the values in M are integers within a certain range $[r_{min}, r_{max}]$ of size $R = r_{max} - r_{min} + 1$, then dynamic programming (DP) methods (cf. [12,21,22]) allow to compute the probability distribution (and hence the E-value) in $O(m^2 R|\mathcal{A}|)$ time.

In practice the probability distribution is often not exactly, or completely calculated due to concerns of speed. E.g., in the *EMATRIX* system [12] score thresholds are calculated and stored for probability values in the interval $\pi = 10^{-1}, 10^{-2},..., 10^{-40}$ only. Consequently, the user can only specify one of these p-value cutoffs. For the calculation of the p-value from a determined match score, *EMATRIX* uses log-linear interpolation on the stored thresholds. A different, commonly used strategy to derive a continuous distribution function uses the extreme value distribution as an approximation [31-33] of high scoring matches.

Even though it is widely accepted that high-scoring local alignment score distributions of the popular position independent scoring systems PAM and BLOSUM can be well approximated by an extreme value distribution, this cannot be generalized for arbitrary PSSMs.

To check whether an extreme value distribution is a suitable approximation for the distribution of PSSM match scores, we sampled the match scores of PSSMs arbitrarily chosen from the TRANSFAC and BLOCKS database. We randomly shuffled 1000 human promotor sequences of length 1200, taken from the database of transcriptional start sites (DBTSS) and 1000 protein sequences of length 365 (= average sequence length in Uniprot-Swissprot), respectively, preserving their mono-symbol composition. From the derived random PSSM match scores we took the best score for each sequence and calculated the empirical cumulative distribution function. If the match scores S are extreme value distributed, a X-Y plot with $X = S$ and $Y = \log(-\log(S))$ should appear linear, since $\log\left(-\log\left(e^{-e^{-\lambda(x-u)}}\right)\right) = -\lambda(x-u)$ holds. For the TRANSFAC PSSM shown in Figure 9, the X-Y plot clearly indicates that an extreme value distribution is not an appropriate approximation. For PSSM IPB003211A (see Figure 10) from the BLOCKS database, it seems as if the score distribution can be approximated quite well with an extreme value distribution. However, we then still have

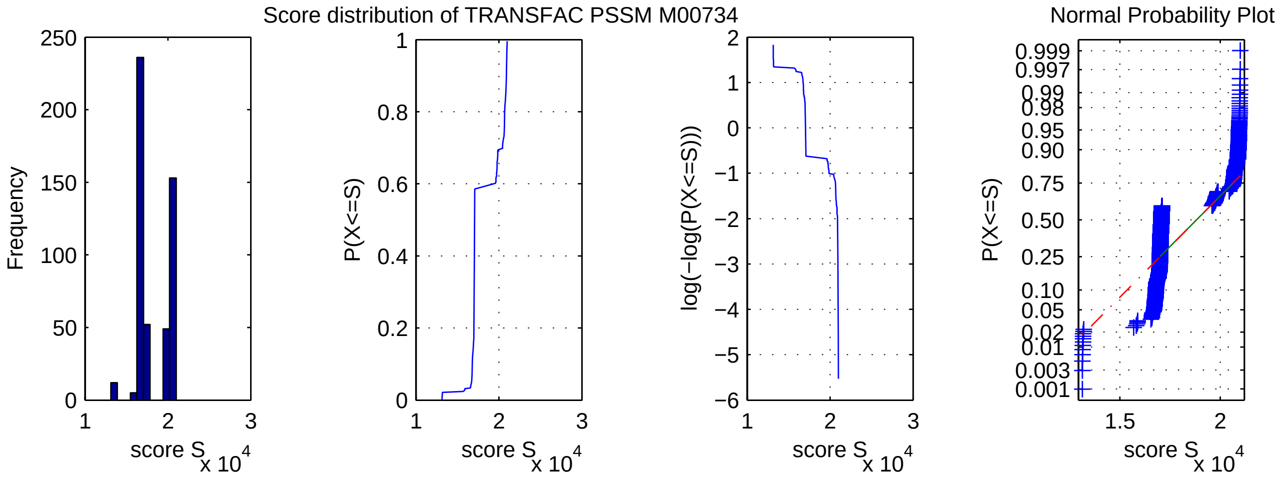

Figure 9
Score distribution of TRANSFAC PSSM M00734. Histogram, cumulative score distribution function, X-Y plot, and normal probability plot of TRANSFAC PSSM M00734 (PSSM length m = 9).

the problem of adequate parameter estimation for the distribution function. Since we do not make any assumptions about the used PSSMs in our algorithm, neither about the type of scores, nor the score range, a proper approximation of the score distribution of arbitrary PSSMs is not possible, without time consuming simulations. That is why we are more interested in an exact solution and thus we focus on the efficient computation of an exact discrete score distribution.

Calculation of exact PSSM score distributions

While recent publications focus on the computation of the complete probability distribution, what is required specifically for PSSM matching, is computing a partial cumulative distribution corresponding to an E-value resp. p-value specified by the user. Therefore, we have developed a new "lazy" method to efficiently compute only a small fraction of the complete distribution.

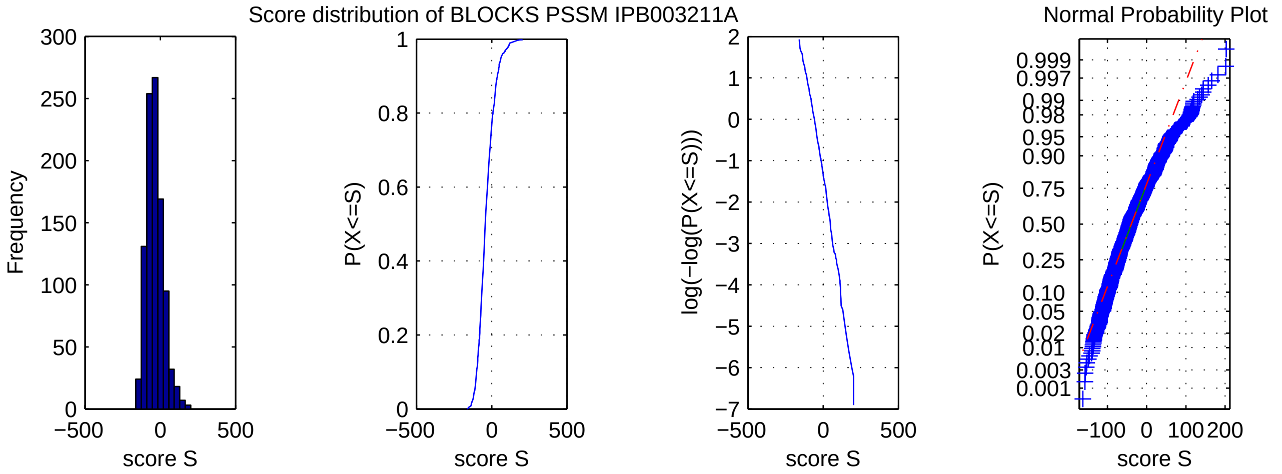

Figure 10
Score distribution of BLOCKS PSSM IPB003211A. Histogram, cumulative score distribution, X-Y plot, and normal probability plot of a PSSM taken from the BLOCKS database (Accession: IPB003211A; PSSM length m = 40), describing the UreI protein of *Helicobacter pylori*, a proton gated urea channel [36].

We formulate the problem we solve w.r.t. E-values and p-values: Given a user specified E-value η, find the minimum threshold $Tmin_E(\eta, M)$, such that the expected number of matches of M in a random sequence of given length is at most η. Given a user specified p-value π, find the minimum threshold $Tmin_{\mathcal{P}}(\pi, M)$, such that the probability that M matches a random string of length m is at most π. The threshold $Tmin_E(\eta, M)$ can be computed from $Tmin_{\mathcal{P}}(\pi, M)$ according to the equation $Tmin_E(\pi \cdot (n - m + 1), M) = Tmin_{\mathcal{P}}(\pi, M)$. Hence we restrict on computing $Tmin_{\mathcal{P}}(\pi, M)$.

Since all strings of length m have a score between $sc_{\min}(M)$ and $sc_{\max}(M)$, we conclude $Tmin_{\mathcal{P}}(1, M) = sc_{\min}(M)$ and $Tmin_{\mathcal{P}}(0, M) > sc_{\max}(M)$. To explain our lazy evaluation method, we first consider existing methods based on DP.

Evaluation with dynamic programming

We assume that at each position in sequence S, the symbols occur independently, with probability $f(a) = (1/n) \cdot |\{i \in [0, n - 1] \mid S[i] = a\}|$. Thus a substring w of length m in S occurs with probability $\prod_{i=0}^{m-1} f(w[i])$ and the probability of observing the event $sc(w, M) = t$ is $\mathbb{P}[sc(w, M) = t] = \sum_{w \in \mathcal{A}^m : sc(w,M)=t} \prod_{i=0}^{m-1} f(w[i])$. We obtain $Tmin_{\mathcal{P}}(\pi, M)$ by a look-up in the distribution:

$Tmin_{\mathcal{P}}(\pi, M) = \min\{t \mid sc_{\min}(M) \leq t \leq sc_{\max}(M), \mathbb{P}[sc(w, M) \geq t] \leq \pi\}$.

If the values in the PSSM M are integers in a range of width R, dynamic programming allows to efficiently compute the probability distribution. The dynamic programming aspect becomes more obvious by introducing for each $k \in [0, m - 1]$ the *prefix* PSSM $M_k : [0, k] \times \mathcal{A} \rightarrow \mathbb{N}$ defined by $M_k(j, a) = M(j, a)$ for $j \in [0, k]$ and $a \in \mathcal{A}$.

Corresponding distributions $Q_k(t)$ for $k \in [0, m - 1]$ and $t \in [sc_{\min}(M_k), sc_{\max}(M_k)]$, and $Q_{-1}(t)$, are defined by

$$Q_{-1}(t) := \begin{cases} 1 & \text{if } t = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$Q_k(t) := \sum_{a \in \mathcal{A}} Q_{k-1}(t - M(k,a)) f(a)$$

We have $\mathbb{P}[sc(w, M) = t] = Q_{m-1}(t)$. The algorithm computing Q_k determines a set of probability distributions for M_0, ..., M_k. Q_k is evaluated in $O(sc_{\max}(M)|\mathcal{A}|)$ time from Q_{k-1}, summing up to $O(sc_{\max}(M)\ \mathcal{A}|\ m)$ total time. See Figure 11 for an example.

If we allow for floating point scores that are rounded to ε decimal places, the time and space requirement increases by a factor of 10^{ε}. Conversely, if all integer scores share a greatest common divisor z, the matrix should be canceled down by z.

Restricted probability computation

In order to find $Tmin_{\mathcal{P}}(\pi, M)$ it is not necessary to compute the whole codomain of the distribution function $Q = Q_{m-1}$. We propose a new method only computing a partial distribution by summing over the probabilities for decreasing threshold values $sc_{\max}(M)$, $sc_{\max}(M) - 1$,..., until the given p-value π is exceeded (see Figures 11, 12).

In step d we compute $Q(sc_{\max}(M) - d)$ where all intermediate scores contributing to $sc_{\max}(M) - d$ have to be considered. In analogy to lookahead scoring, in each row j of M we avoid all intermediate scores below the intermediate threshold th_j because they do not contribute to $Q(sc_{\max}(M) - d)$. The algorithm stops if the cumulated probability for threshold $sc_{\max}(M) - d$ exceeds the given p-value π and we obtain $Tmin_{\mathcal{P}}(\pi, M) = sc_{\max}(M) - d + 1$.

Lazy evaluation of the permuted matrix

The restricted computation strategy performs best if there are only few iterations (i.e., $Tmin_{\mathcal{P}}(\pi, M)$ is close to $sc_{\max}(M)$) and in each iteration step the computation of $Q_k(t)$ can be skipped in an early stage, i.e., for small values of k. The latter occurs to be more likely if the first rows of M contain strongly discriminative values leading to the exclusion of the small values by comparison with the intermediate thresholds. An example of this situation is given in Figure 1. Since $Q_k(t)$ is invariant to the permutation of the rows of M, we can sort the rows of M such that the most discriminative rows come first. We found that the difference between the largest two values of a row is a suitable measure for the level of discrimination since a larger difference increases the probability to remain below the intermediate threshold. Since the rows of M are scanned several times, we save time by initially sorting each row in order of descending score. We divide the computation steps where the step d computes $Q(sc_{\max}(M) - d)$:

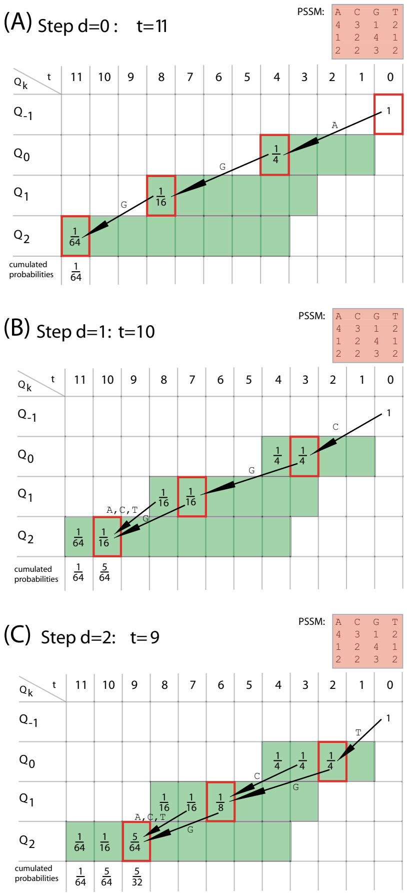

Figure 11

Evaluation with dynamic programming. The simple DP scheme computes all probability vectors Q_0, Q_1, Q_2 completely within the green marked area, corresponding to score ranges of prefix PSSMs M_k. In contrast to the simple scheme, the restricted probability computation method computes only the upper end of the probability distribution until the given p-value threshold is exceeded, omitting parts of the green area. In this example we show how to compute the score threshold $Tmin_{\mathcal{P}}$ (π, M) for PSSM M of length $m = 3$ and a score range of [4,11] corresponding to a given p-value threshold of $\pi = \frac{1}{8}$. For simplicity we assume a uniform character distribution of $f(A) = f(C) = f(G) = f(T) = \frac{1}{4}$. Cells of the matrix that are computed in the step actually under consideration are marked red. In step $d = 0$, see (A), the algorithm computes $Q_2(11)$ recursively for all paths through M that achieve a score of 11, i.e. $Q_2(11) = Q_1(8) \cdot f(G)$, $Q_1(8) = Q_0(4) \cdot f(G)$, $Q_0(4) = Q_{-1}(0) \cdot f(A) = 1 \cdot \frac{1}{4}$, since AGG is the only path achieving score 11. It follows $Q_2(11) = \frac{1}{64}$. In step $d = 1$ all paths achieving a score of $11 - d = 10$ to determine $Q_2(10)$ are computed, see (B). We conclude $Q_2(10) = \frac{1}{16}$. In this step, DP allows to reuse value $Q_1(8)$ without recomputation. In step $d = 2$, see (C) values $Q_1(7)$ and $Q_0(3)$ can be reused to compute $Q_2(9) = \frac{5}{64}$. In step $d = 2$ the cumulated probability $Q_2(11) + Q_2(10) + Q_2(9) = \frac{5}{32}$ exceeds the given p-value threshold of $\pi = \frac{1}{8}$, and the restricted probability computation method skips the rest of the computation. We obtain a score threshold of $th = 10$ correponding to π.

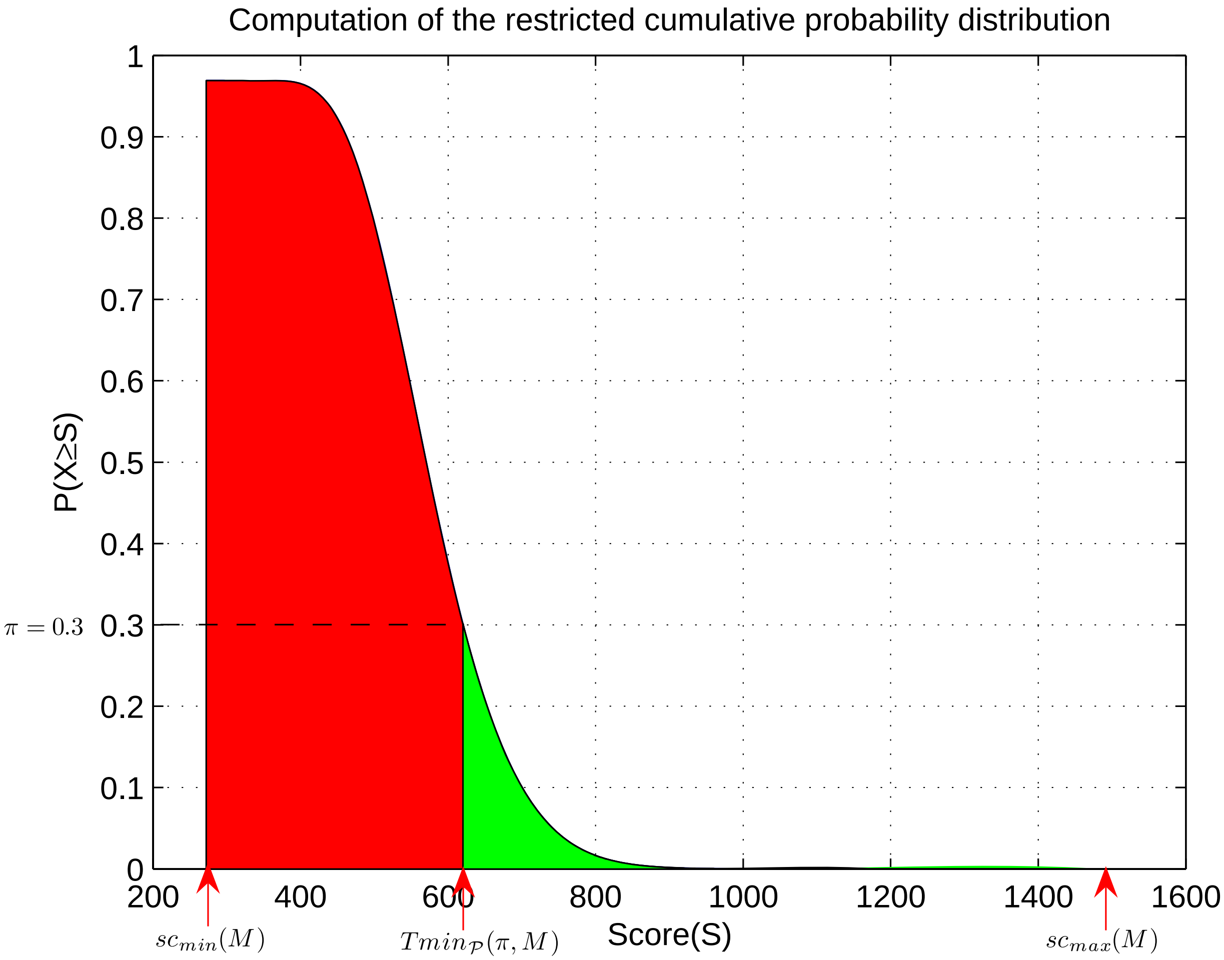

Figure 12

Restricted probability computation. Computation of the partial cumulative distribution function. Observe that in order to determine $Tmin_{\mathcal{P}}$ (π, *M*) for $\pi = 0.3$ we do not have to calculate the complete distribution in the score range [$sc_{\min}(M)$, $sc_{\max}(M)$]. It is sufficient to calculate only the upper end (green area) starting with $sc_{\max}(M)$ until $\mathbb{P}[X \geq S] \geq \pi$.

In step $d = 0$ only the maximal scores $\max_i$, $i \in [0, m - 1]$ in each row have to be evaluated.

In step $d > 0$ all scores $M(i, a) \geq \max_i - d$ may contribute to $Q(sc_{\max}(M) - d)$. Since in general a score value $M(i, a) \geq \max_i - d$ also gives contribution to $Q(sc_{\max}(M) - l)$ for $l > d$, we can save time by storing $Q_i(\max_i - l)$ for $l > d$, in step d in a buffer and reusing the buffer in steps $d + 1$, $d + 2$,.... This allows for the computation of $Q_k(sc_{\max}(M) - d)$ only based on the buffer and scores $M(i, a) = \max_i - d$ while scores $M(i, a) > \max_i - d$, $i \in [0, m - 1]$, can be omitted. We therefore have developed an algorithm *LazyDistrib* employing lazy evaluation of the distribution. That is, given a threshold *th*, the algorithm only evaluates parts of the DP vectors necessary to determine $Q_k(th)$ and simultaneously saves sub-results concerned with score *th* in an additional buffer matrix *Pbuf* (instead of recomputing them later, see Figure 13 for an example). This is described by the following recurrence:

$$\begin{aligned} Q_k(th-d) &= Pbuf_k(th-d) + \\ &\quad \sum_{a\in\mathcal{A}:M(k,a)\geq\max_k-d} Q_{k-1}(th-d-M(k,a))f(a) \\ Pbuf_k(th-d) &:= \sum_{a\in\mathcal{A}:M(k,a)<\max_k-d} Q_{k-1}(th-d-M(k,a))f(a) \end{aligned}$$

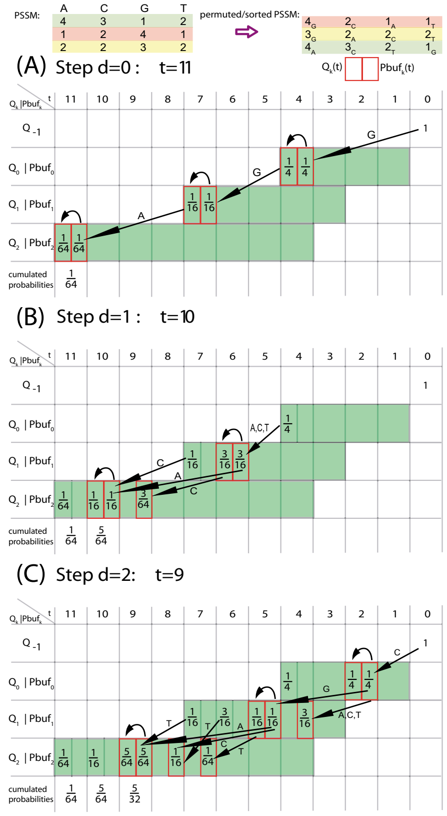

Figure 13

Probability computation using lazy evaluation of the DP matrix. In this example we use the same PSSM *M*, character distribution, and p-value threshold $\pi = \frac{1}{8}$ as in Figure 11. However, in each row of the PSSM the scores are sorted in descending order, and the rows are sorted with the most discriminant row coming first (see coloured PSSMs for this relationship). Observe that the *LazyDistrib* algorithm evaluates the DP vectors non-recursively top-down. Cells computed in the actual step are marked red. In step $d = 0$ the algorithm computes $Q_2(11)$ by evaluating paths through the PSSM contributing to Q_2(II), which is in this example only the high scoring path GGA. Intermediate results of $Q_0(4)$, $Q_1(7)$, and $Q_2(11)$ are collected in buffers $Pbuf_0(4)$, $Pbuf_1(7)$, and $Pbuf_2(11)$ first, and finally copied to the correponding cells in *Q*. See (A) for the situation after step $d = 0$ has been completed. In step $d = 1$, see (B), the algorithm computes $Q_2(10)$, starting in row $k = 1$ with the determination of $Pbuf_1(6)$ and $Q_1(6)$. That is, $Q_1(6) = Pbuf_1(6) = Q_0(4)\cdot f(A) + Q_0(4)\cdot f(C) + Q_0(4)\cdot f(T) = \frac{3}{16}$. Analogously $Q_2(10)$ and $Pbuf_2(10)$ are computed based on $Q_1(7)$ and $Q_1(6)$. Additionally $Pbuf_2(9)$ is filled for further reuse in subsequent steps $d + 1$, $d + 2$,.... We compute $Pbuf_2(9) = Q_1(6)\cdot f(C) = \frac{3}{64}$. The algorithm can directly start in row $k = 1$ with the computation of $Q_1(6)$ instead of $Q_0(3)$ since a score of 3 cannot be achieved by the first prefix PSSM M_0. Only score 4 of M_0 contributes to $Q_2(10)$, scores 2 and 1 do not. In step $d = 2$, see (C), the algorithm computes $Q_2(9)$, starting in row $k = 0$. $Pbuf_2(9)$ is computed reusing the partial sum calculated in previous steps, such that $Pbuf_2(9) = \frac{3}{64} + Q_1(7)\cdot f(T) + Pbuf_1(5)\cdot f(A) = \frac{5}{64}$, and then copied to $Q_2(9)$. $Pbuf_1(4)$, $Pbuf_2(8)$, and $Pbuf_2(7)$ are filled based on $Pbuf_0(2)$, $Q_1(6)$, $Pbuf_1(5)$, and $Q_1(5)$ for further reuse. After step $d = 2$ the rest of the computation can be skipped since the cumulated probability $Q_2(11) + Q_2(10) + Q_2(9) = \frac{5}{32}$ exceeds the given p-value $\pi = \frac{1}{8}$ and we obtain a score threshold of $th = 10$ corresponding to π.

Table 1: Performed experiments and experimental input.

	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5	Exp. 6
# searched sequences	59,021	30,964	19,111	1 (*H.s.* Chr. 6)	19,111	19,111
total length	20.2 MB	37.2 MB	4.3 MB	162.9 MB	4.3 MB	4.3 MB
sequence source	see [13]	DBTSS 5.1	RCSB PDB	Sanger V1. 4	RCSB PDB	RCSB PDB
sequence type/PSSM type	protein	DNA	protein	DNA	protein	protein
# PSSMs	4,034	220	11,411	576	28,337	10,931
PSSM source	see [13]	MatInspector	PRINTS 38	TRANSFAC Prof. 6.2	BLOCKS 14.1	PRINTS 38
avg. length of PSSMs	29.74	14.21	17.32	13.33	26.3	17.37
index construction (sec)	41	146	10.2	586	10.2	10.2
mdc (sec)	1960	-	1486	-	11871	1486
MatInspector		x				
FingerPrintScan			x			
Blimps					x	
DN00	x					
LAsearch	x	x	x	x	x	
ESAsearch	x	x	x	x	x	x
ESAsearch (reduced $\mathcal{A}$)						x

Overview of the sequences and PSSMs used in the performed experiments. For the experiments that use p-value or E-value cutoffs, we precomputed the cumulative score distributions and stored them on file. *mdc* is the time needed for this task. In Experiment 1 we measured the running time of the Java-program from [13], referred to by *DN00*. We ran *DN00* with a maximum of 2 GB memory assigned to the Java virtual machine. *DN00* constructs the suffix tree in main memory and then performs the searches. For a fair comparison, we therefore measured the total running time, and the time for matching the PSSMs (without suffix tree construction). For Experiment 2, we implemented the matrix similarity scoring scheme (MSS) of *MatInspector* and matched the PSSMs against both strands of the DNA sequences with different MSS cutoff values. The MSS of PSSM M of length m and a sequence $w \in \mathcal{A}^m$ is defined as $\mathrm{MSS} = \frac{sc(w,M) - sc_{\min}(M)}{sc_{\max}(M) - sc_{\min}(M)}$ and hence given an MSS cutoff value, the threshold th is determined as $th = \mathrm{MSS}\cdot(sc_{\max}(M) - sc_{\min}(M)) + sc_{\min}(M)$. Instead of using the reverse strand we use the reverse complement $\overline{M}$ of the PSSM M, defined by $\overline{M}(i, a) = M(m - 1 - i, \overline{a})$ for all $i \in [0, m - 1]$ and $a \in \mathcal{A}$, where $\overline{a}$ is the Watson Crick complement of nucleotide a. This allows to use the same enhanced suffix array for both strands. In Experiment 5 we used a PERL-based wrapper for the *Blimps* program shipped with the BLIMPS distribution to do bulk sequence searches. The overhead for the PERL interpreter call was found to be negligible. For Experiment 6 we used the reduced alphabets given in Figure 8. The last seven rows show which programs were used in which experiment.

In the present implementation, the algorithm assumes independently distributed symbols. The algorithm can be extended to an order d-Markov model (w.r.t. the background alphabet distribution). This increases the computation time by a factor of $|\mathcal{A}|^d$.

Implementation and computational results

We implemented *LAsearch*, *ESAsearch*, both capable to handle reduced alphabets, and *LazyDistrib* in C. The program was compiled with the GNU C compiler (version 3.1, optimization option -03). All measurements were performed on a 8 CPU Sun UltraSparc III computer running at 900 MHz, with 64 GB main memory (using only one CPU and a small fraction of the memory). Enhanced suffix arrays were constructed with the program mkvtree, see [34].

We performed seven experiments comparing different programs for searching PSSMs. Table 1 gives more details on the experimental input for Experiments 1–6. Results are given in Table 2 (Exp. 1–5) and Figure 14 (Exp. 6). For Experiment 7, see Figures 15 and 16. In these experiments *ESAsearch* performed very well, especially on nucleotide PSSMs, see Experiments 2 and 4. It is faster than *MatInspector* by a factor between 63 and 1,037, depending on the stringency of the given thresholds. The commercial advancement of *MatInspector*, called *MATCH*, was not available for our comparisons, but based on [7] we presume a running time comparable to *MatInspector*. Compared to *LAsearch*, *ESAsearch* is faster by a factor between 17 (MSS = 0.80) and 196 (MSS = 0.95) (see Experiment 2). On larger nucleotide sequences (see Experiment 4) the speedup factors increase, ranging from 58 (MSS = 0.85) to 275 (MSS = 0.95). See Table 1 for the definition of MSS. In the experiments using protein PSSMs, *ESAsearch* is faster than the method of [13] by a factor between 1.5 and 1.8 (see Experiment 1). This is due to the better locality behavior of the enhanced suffix array compared to a suffix tree. For larger p-values *LAsearch* performs slightly better than *ESAsearch*. Increasing the stringency, the performance of *ESAsearch* increases, resulting in a speedup of factor 1.5 for a p-value of 10^{-40}. We explain this behavior by

Table 2: Results of Experiments 1–5.

Experiment 1: 4,034 PSSMs in 20.2 MB protein sequences

p-value	*DN00* (total time)	*DN00* (search)	*LAsearch*	*ESAsearch* +41 sec.
10^{-10}	65,808	64,939	39,839	41,813
10^{-20}	38,773	37,706	23,786	24,378
10^{-30}	21,449	20,362	14,111	13,084
10^{-40}	9,606	8,533	8,067	5,374

Experiment 2: 220 PSSMs in 37.2 MB DNA

MSS		*MatInspector*	*LAsearch*	*ESAsearch* +32 sec.
0.80		12,773	3,605	202
0.85		12,567	3,189	108
0.90		12,487	2,818	53
0.95		12,445	2,356	12
1.00		12,429	885	1

Experiment 3: 11,411 PSSMs in 4.3 MB protein sequences

E-value		*FingerPrintScan*	*LAsearch*	*ESAsearch* +10.2 sec.
10^{-10}		4,733	3,423	1.244
10^{-20}		4,710	486	52
10^{-30}		4,706	27	10

Experiment 4: 576 PSSMs in 162.9 MB DNA

MSS			*LAsearch*	*ESAsearch* +586 sec.
0.85			18,446	318
0.90			16,376	150
0.95			13,764	50
1.00			5,294	1

Experiment 5: 28,337 PSSMs in 4.3 MB protein sequences

raw-*th*		*Blimps*	*LAsearch*	*ESAsearch* +10.2 sec.
945		271:30:16	16:03:12	11:35:58

Experiment 1: Running times in seconds of the different PSSM searching methods at different levels of stringency, when searching for 4,034 amino acid PSSMs in 59,021 sequences (21.2 MB) from SwissProt. These are the same PSSMs and sequences used in the experiments of [13]. Experiment 2: Running times in seconds of *MatInspector*, *LAsearch*, and *ESAsearch*, when searching 220 PSSMs on both strands of 37.2 MB DNA sequence data at different matrix similarity score (MSS) cutoffs. Experiment 3: Running times in seconds of *FingerPrintScan*, *LAsearch*, and *ESAsearch* when searching all 11,411 PSSMs from the PRINTS database in the RCSB protein data bank (PDB) for different E-values. Experiment 4: Running times in seconds of *LAsearch* and *ESAsearch* when searching 576 PSSMs in H. sapiens chr. 6 at different matrix similarity score (MSS) cutoffs. Experiment 5: Running times in hh:mm:ss of *Blimps*, *LAsearch*, and *ESAsearch* when searching all 28,337 PSSMs from the BLOCKS database in PDB. We used a raw score threshold of 945 as suggested in the *Blimps* documentation for searching large databases. For each experiment, the additional time needed for the construction of the enhanced suffix array is shown in the head of the *ESAsearch* column.

the larger alphabet size, resulting in shorter common prefixes and therefore smaller skipped areas of the enhanced suffix array. With increasing stringency of the threshold, the expected reading depth decreases, resulting in larger skipped areas of the enhanced suffix array. Compared to the *FingerPrintScan* program, *ESAsearch* achieves a speedup factor between 3.8 and 470, see Experiment 3. In comparison to *Blimps*, the PSSM-searching program of the BLOCKS database, *ESAsearch* is faster by a factor of 23 (see Experiment 5) for the chosen threshold. In Experiment 6 (see Figure 14), we measured the influence of alphabet reductions on the running time of *ESAsearch* when using

Table 3: Running times of the LazyDistrib algorithm.

p-value	simple DP	*LazyDistrib*	speedup factor
10^{-10}	1,486	485.8	3
10^{-20}	1,486	92.5	95
10^{-30}	1,486	8.9	166
10^{-40}	1,486	4.5	330

Running times in seconds when computing score thresholds for all 11,411 PSSMs from the PRINTS database (Rel. 38), given different p-values. Running times given in this table are measurements performed with improved versions of the simple DP and *LazyDistrib* algorithms and thus are much lower than the times given in [25].

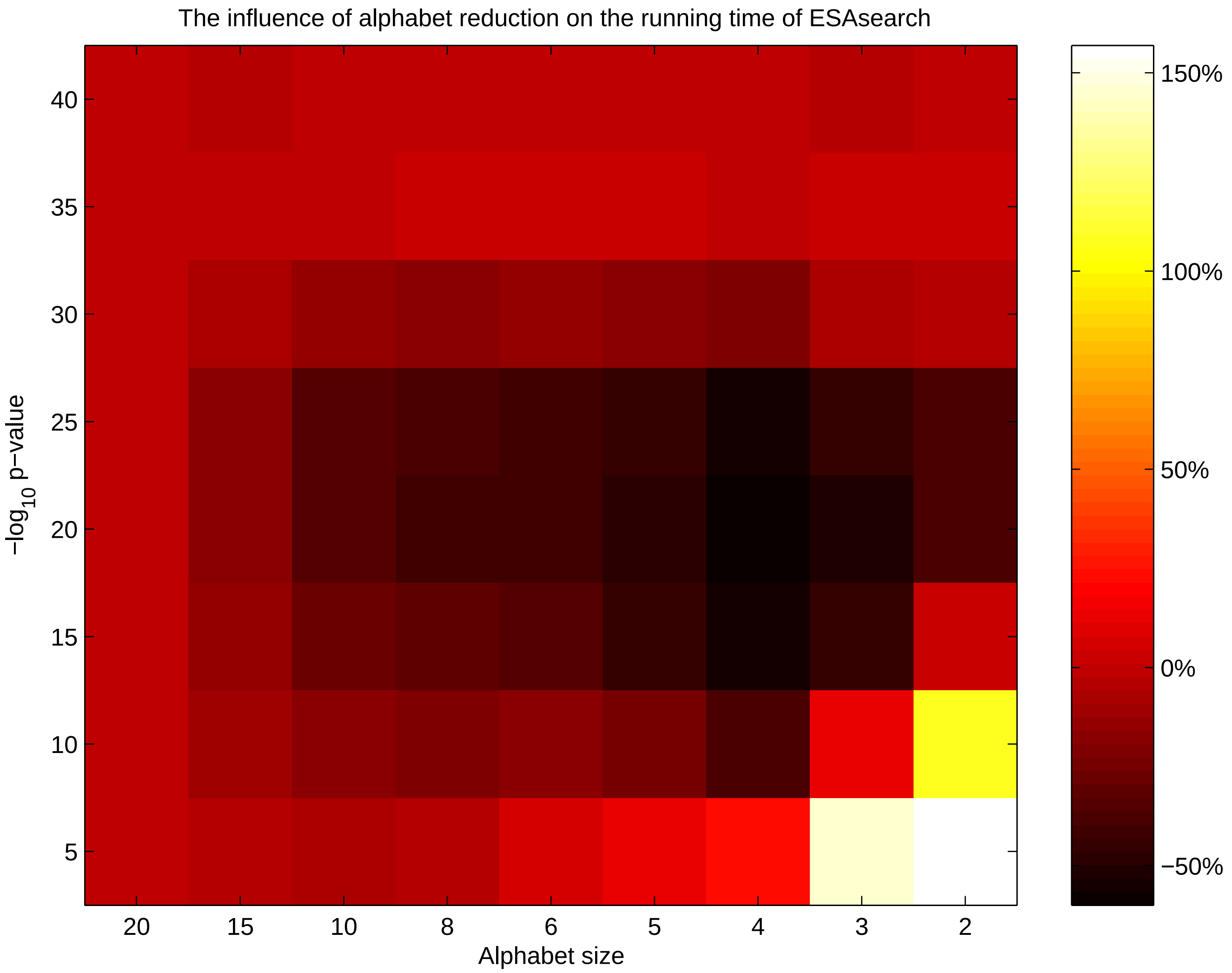

Figure 14
Effect of alphabet reduction on the running time of ESAsearch. Experiment 6: Relative deviations of running time of *ESAsearch* when using reduced alphabets at different levels of stringency. We measured the relative percentage deviation with respect to the running time when using the standard 20 letter amino acid alphabet (= 0%). We searched with 11,411 PSSMs from the PRINTS database (Rel. 38) in the RCSB Protein Data Bank (PDB) with a total sequence length of 4.3 MB. In this example, the maximum performance improvement is achieved for an alphabet of size 4 and a p-value cutoff of $\pi = 10^{-20}$.

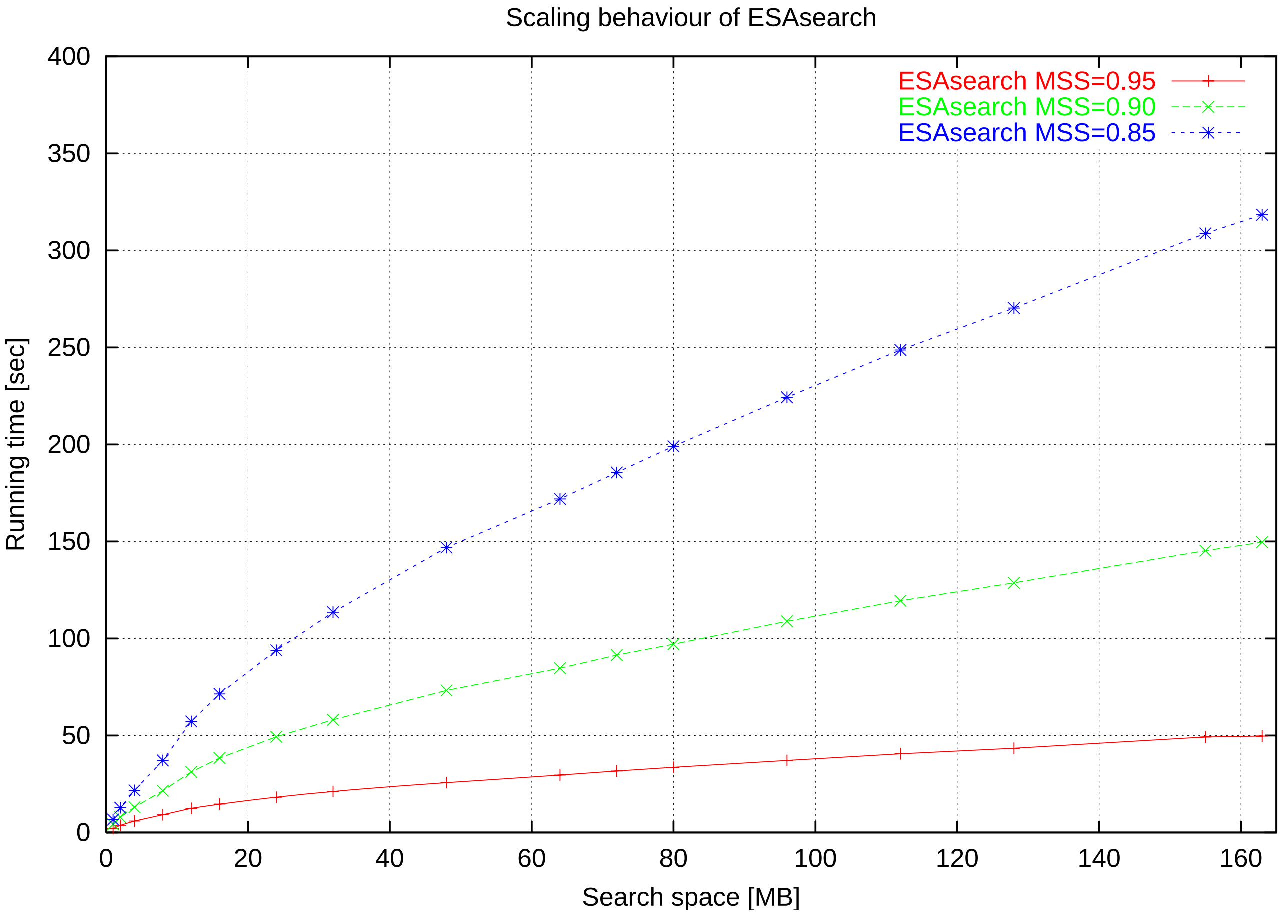

Figure 15
Scaling behaviour of ESAsearch. Experiment 7: Scaling behavior of *ESAsearch* when searching with 576 TRANSFAC PSSMs on subsets of human chromosome 6 of different sizes and with different matrix similarity cutoff values (MSS). The subsets are prefixes of human chromosome 6 of length 2^k for k = 0, 1, 2,..., 7.

protein PSSMs. Compared to the performance of *ESAsearch* operating on the normal 20 letter amino acid alphabet a speedup up to factor 2 can be achieved when using a 4 letter alphabet and a p-value cutoff of 10^{-20}. Experiment 7 (see Figures 15 and 16) shows that the expected running time of *ESAsearch* is sublinear, whereas *LAsearch* runs in linear time. In a final experiment, we compared algorithm *LazyDistrib* with the DP-algorithm computing the complete distribution. *LazyDistrib* shows a speedup factor between 3 and 330 on our test set, depending on the stringency of the threshold (see Table 3).

PoSSuM software distribution

Our software tool *PoSSuMsearch* implements all algorithms and ideas presented in this work, namely *Simplesearch*, *LAsearch*, *ESAsearch* and *LazyDistrib*. A user can search for PSSMs in enhanced suffix arrays built by mkvtree from the *Vmatch* package, as well as on plain sequence data in FASTA, GENBANK, EMBL, or SWISSPROT format. The search algorithm can be chosen from the command line.

PSSMs are specified in a simple plain text format, where one file may contain multiple PSSMs. The alphabet a PSSM refers to, and alphabet character to PSSM column assignments can be specified on a per-PSSM basis for most flexible alphabet support. All implemented algorithms support alphabet transformations. PSSMs can contain integer as well as floating point scores. To prevent rounding errors for integer based PSSMs, *PoSSuMsearch* uses integer arithmetics for these, resulting in an additional speedup on most CPU architectures. Searching on the reverse strand of nucleotide sequences is implemented by PSSM transformation according to Watson-Crick base

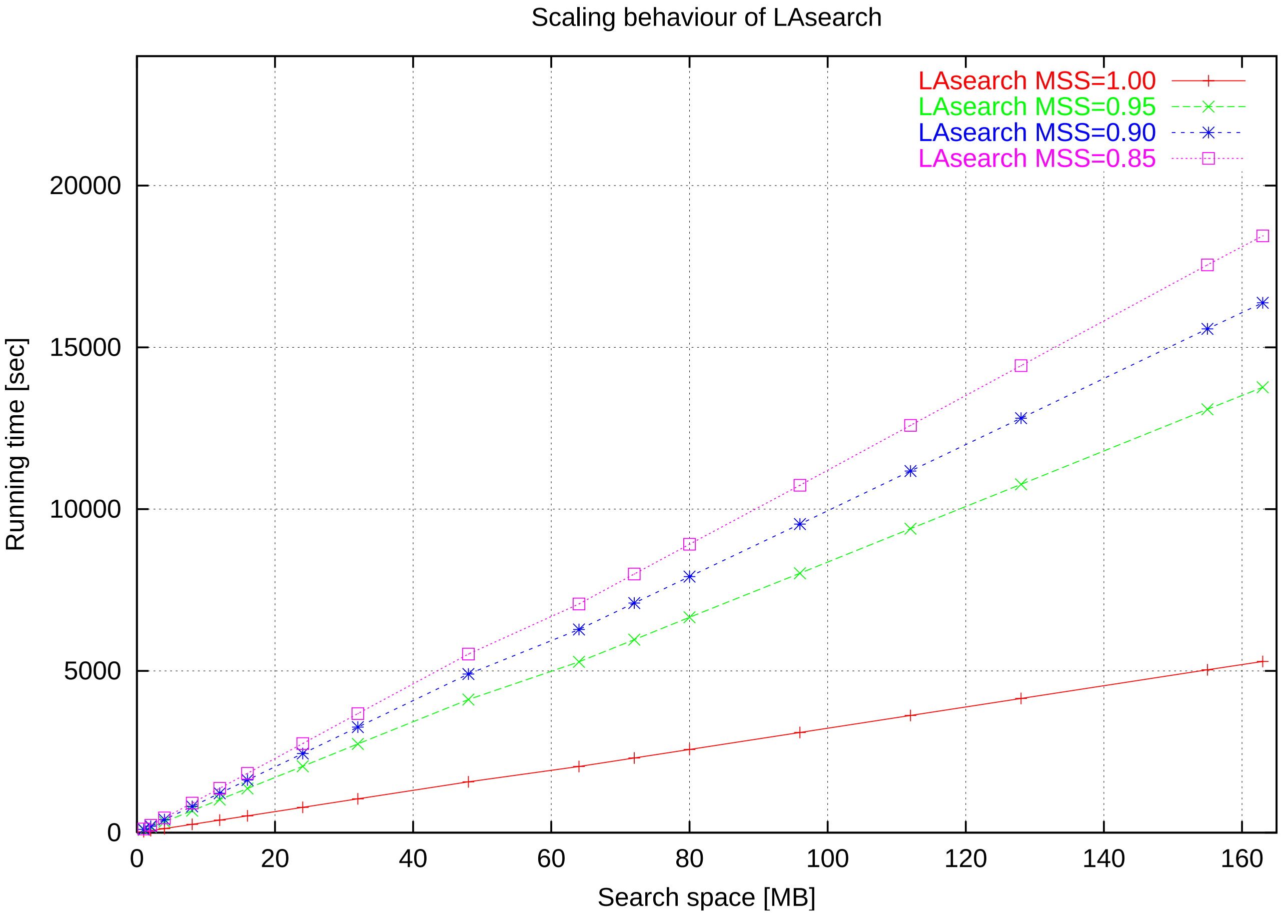

Figure 16
Scaling behaviour of LAsearch. Experiment 7: Scaling behavior of *LAsearch* when searching with 576 TRANSFAC PSSMs on subsets of human chromosome 6 of different sizes and with different matrix similarity cutoff values (MSS). The subsets are prefixes of human chromosome 6 of length 2^k for k = 0, 1, 2,..., 7.

pairing. Hence it is sufficient to build the enhanced suffix array for one strand only. This can then be used to search both strands.

The cutoff can be specified as p-value, E-value, MSS (matrix similarity score), or raw score threshold. If only the best matches with the highest scores need to be known, then *PoSSuMsearch* can be asked to report only the *k* highest scoring matches without even specifying an explicit cutoff. To do so, the search algorithms dynamically adapt the threshold during the search. When using p- or E-values, the score threshold is determined by either the lazy dynamic programming algorithm introduced in this contribution, or read from file that stores the complete precalculated probability distribution. Background distributions can be specified arbitrarily by the user, or determined from a given sequence database. We provide a tool, *PoSSuMdist*, to generate a compressed file containing the complete precalculated probability distribution for a set of PSSMs.

PSSM matches can be sorted by specifying a list of sort keys, like p-value, match score, sequence number, and so on. The output formats of *PoSSuMsearch* print out all available information about a match, either in a human readable format, tab delimited, or in machine readable, XML-based CisML [35]. *PoSSuMsearch* as well as *PoSSuMdist* support multi-threading for a further reduction of running time on multi CPU machines.

The *PoSSuM* software distribution includes the searching tool *PoSSuMsearch* itself, and additional tools to determine character frequencies from sequence data, for probability distribution calculation, and PSSM format converters for *TRANSFAC*, *BLOCKS*, *PRINTS*, and *EMATRIX* style PSSMs.

Discussion and conclusion

We have presented a new non-heuristic algorithm for searching with PSSMs, achieving expected sublinear running time. Our analysis of *ESAsearch* shows that for sequences not shorter than $|\mathcal{A}|^m + m - 1$ a linear runtime in the worst case is achieved. It shows superior performance over the most widely used programs, especially for DNA sequences. The enhanced suffix array, on which the method is based, requires only $9n$ bytes. This is a space reduction of more than 45 percent compared to the $17n$ bytes implementation of [13]. Further on, we developed a systematic concept for alphabet reduction, especially useful on amino acid sequences and PSSMs for gaining additional speedup. Our third main contribution is a new algorithm for the efficient calculation of score thresholds from user defined E-values and p-values. The algorithm allows for accurate on-the-fly calculations of thresholds, and has the potential to replace formerly used approximation approaches. Beyond the algorithmic contributions, we provide a robust, well documented, and easy to use software package, implementing the ideas and algorithms presented in this manuscript.

Availability

The *PoSSuM* software distribution and its documentation is available precompiled for different operating systems and architectures on [24]. A version of mkvtree is included. A web based version of *PoSSuMsearch* is available under the same URL.

Authors' contributions

M.B. developed the algorithms presented in this manuscript, and wrote significant parts of the manuscript. R.H. implemented the algorithms, created the software distribution, and contributed to the manuscript. M.B. and R.H. wrote the documentation for the software package. R.G. provided supervision and guidance on the project and provided essential infrastructure. S.K. provided supervision, and contributed to the manuscript. All authors read and approved the final manuscript.

Acknowledgements

The authors thank Sven Rahmann and three anonymous reviewers for comments on the manuscript, Alexander Kel from Biobase GmbH Germany for providing the TRANSFAC PSSMs used in the benchmark experiments, and Jan Krüger for setting up the web interface and integrating *PoSSuMsearch* in our local web-service environment. M.B. and R.H. were supported by the International NRW Graduate School in Bioinformatics and Genome Research.

References

1. Gribskov M, McLachlan M, Eisenberg D: **Profile Analysis: Detection of Distantly Related Proteins.** *Proc Nat Acad Sci USA* 1987, **84:**4355-4358.
2. Hulo N, Sigrist C, Le Saux V, Langendijk-Genevaux PS, Bordoli L, Gattiker A, De Castro E, Bucher P, Bairoch A: **Recent improvements to the PROSITE database.** *Nud Acids Res* 2004, **32:**134-137.
3. Attwood TK, Bradley P, Flower DR, Gaulton A, Maudling N, Mitchell AL, Moulton G, Nordle A, Paine K, Taylor P, Uddin A, Zygouri C: **PRINTS and its automatic supplement, prePRINTS.** *Nucl Acids Res* 2003, **31:**400-402.
4. Henikoff J, Greene E, Pietrokovski S, Henikoff S: **Increased Coverage of Protein Families with the Blocks Database Servers.** *Nucl Acids Res* 2000, **28:**228-230.
5. Wu T, Nevill-Manning C, Brutlag D: **Minimal-risk scoring matrices for sequence analysis.** *J Comp Biol* 1999, **6(2):**219-235.
6. Sandelin A, Alkema W, Engstrom P, Wasserman W, Lenhard B: **JASPAR: an open-access database for eukaryotic transcription factor binding profiles.** *Nucl Acids Res* 2004, **32:**D91-D94.
7. Matys V, Fricke E, Geffers R, Gößiling E, Haubrock M, Hehl R, Hornischer K, Karas D, Kel AE, Kel-Margoulis OV, Kloos DU, Land S, Lewicki-Potapov B, Michael H, Munch R, Reuter I, Rotert S, Saxel H, Scheer M, Thiele S, Wingender E: **TRANSFAC(R): transcriptional regulation, from patterns to profiles.** *Nucl Acids Res* 2003, **31:**374-378.
8. Scordis P, Flower D, Attwood T: **FingerPRINTScan: intelligent searching of the PRINTS motif database.** *Bioinformatics* 1999, **15(10):**799-806.
9. Quandt K, Frech K, Wingender E, Werner T: **MatInd and MatInspector: new fast and versatile tools for detection of consensus matches in nucleotide data.** *Nucl Acids Res* 1995, **23:**4878-4884.
10. Rajasekaran S, Jin X, Spouge J: **The Efficient computation of Position Specific Match Scores with the Fast Fourier Transformation.** *J Comp Biol* 2002, **9:**23-33.
11. Freschi V, Bogliolo A: **Using sequence compression to speedup probabilistic profile matching.** *Bioinformatics* 2005, **21(10):**2225-2229.
12. Wu T, Nevill-Manning C, Brutlag D: **Fast Probabilistic Analysis of Sequence Function using Scoring Matrices.** *Bioinformatics* 2000, **16(3):**233-244.
13. Dorohonceanu B, Nevill-Manning C: **Accelerating Protein Classification Using Suffix Trees.** In *Proc. of the International Conference on Intelligent Systems for Molecular Biology* Menlo Park, CA: AAAI Press; 2000:128-133.
14. Gonnet H: **Some string matching problems from Bioinformatics which still need better solutions.** *Journal of Discrete Algorithms* 2004, **2(1):**3-15.
15. Tatusov R, Altschul S, Koonin E: **Detection of conserved segments in proteins: Iterative scanning of sequence databases with alignment blocks.** *Proc Nat Acad Sci USA* 1994, **91(25):**12091-12095.
16. Henikoff J, Henikoff S: **Using substitution probabilities to improve position-specific scoring matrices.** *Bioinformatics* 1996, **12(2):**135-143.
17. Kel A, Gößling E, Reuter I, Cheremushkin E, Kel-Margoulis O, Wingender E: **MATCH: a tool for searching transcription factor binding sites in DNA sequences.** *Nucl Acids Res* 2003, **31(13):**3576-3579.
18. Abouelhoda M, Kurtz S, Ohlebusch E: **Replacing Suffix Trees with Enhanced Suffix Arrays.** *Journal of Discrete Algorithms* 2004, **2:**53-86.
19. Kurtz S: **Reducing the Space Requirement of Suffix Trees.** *Software – Practice and Experience* 1999, **29(13):**1149-1171.
20. Giegerich R, Kurtz S: **A Comparison of Imperative and Purely Functional Suffix Tree Constructions.** *Science of Computer Programming* 1995, **25(2–3):**187-218.
21. Staden R: **Methods for calculating the probabilities for finding patterns in sequences.** *Comp Appl Biosci* 1989, **5:**89-96.
22. Rahmann S: **Dynamic programming algorithms for two statistical problems in computational biology.** In *Proc. of the 3rd Workshop of Algorithms in Bioinformatics (WABI)* LNCS 2812, Springer Verlag; 2003:151-164.
23. Rahmann S, Müller T, Vingron M: **On the Power of Profiles for Transcription Factor Binding Site Detection.** *Statistical Applications in Genetics and Molecular Biology* 2003, **2(1):**.
24. Beckstette M, Homann R, Giegerich R, Kurtz S: **PoSSuM software distribution.** 2006 [http://bibiserv.techfak.uni-bielefeld.de/possumsearch/].
25. Beckstette M, Strothmann D, Homann R, Giegerich R, Kurtz S: **PoSSuMsearch: Fast and Sensitive Matching of Position Specific Scoring Matrices using Enhanced Suffix Arrays.** In *Proc. of the*

German Conference on Bioinformatics Volume P-53. GI Lecture Notes in Informatics; 2004:53-64.

26. Kärkkäinen J, Sanders P: **Simple Linear Work Suffix Array Construction.** In *Proceedings of the 13th International Conference on Automata, Languages and Programming* Springer; 2003.
27. Kasai T, Lee G, Arimura H, Arikawa S, Park K: **Linear-time Longest-Common-Prefix Computation in Suffix Arrays and its Applications.** In *12th Annual Symposium on Combinatorial Pattern Matching (CPM2001) Volume 2089*. Springer-Verlag, New York: Lecture Notes in Computer Science; 2001:181-192.
28. de Bruijn N: **A Combinatorial Problem.** *Koninklijke Nederlands Akademie van Wetenschappen Proceedings* 1946, **49:**758-764.
29. Li T, Fan K, Wang J, Wang W: **Reduction of protein sequence complexity by residue grouping.** *Protein Engineering* 2003, **16(5):**323-330.
30. Murphy LR, Wallqvist A, Levy R: **Simplified amino acid alphabets for protein fold recognition and implications for folding.** *Protein Engineering* 2000, **13(3):**149-152.
31. Castillo G: *Extreme Value Theory in Engineering* Academic Press; 1988.
32. Embrechts P, Klüppelberg C, Mikosch T: *Modelling Extremal Events* Springer; 1997.
33. Goldstein L, Waterman M: **Approximations to profile score distributions.** *J Comp Biol* 1994, **1:**93-104.
34. Kurtz S: **The Vmatch large scale sequence analysis software.** 2005 [http://www.vmatch.de/].
35. Haverty P, Weng Z: **CisML: an XML-based format for sequence motif detection software.** *Bioinformatics* 2004, **20(11):**1815-1817.
36. Weeks D, Eskandari S, Scott D, Sachs G: **A H+-gated urea channel: the link between Helicobacter pylori urease and gastric colonization.** *Science* 2000, **287:**482-485.