Efficient Search of Position Specific Scoring Matrices with a Truncated Suffix Tree

Alessio Favaretto and Cinzia Pizzi*

Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Padova, Italy
favaret1@dei.unipd.it, cinzia.pizzi@dei.unipd.it

Abstract. Position Specific Scoring Matrix (PSSM) are a widely used model of choice for several kind of biological signals in both DNA and protein contexts. Recently, several algorithms have been proposed to speed up the time required by the basic operation of matching a matrix with a sequence. In this paper we present TruSTsearch, an offline algorithm based on the truncated suffix trees data structure. We provide experimental comparison between our algorithm and other known offline algorithms based on suffix trees and suffix arrays, showing that thanks to the specifically designed TruSTCoding, our solution requires less space, and allows better performances.

1 Introduction

The problem of "matching" matrices and sequences has drawn lot of attention in recent years. The exponential grow of databases of both biological signals and sequences requires faster algorithms to perform the basic task of finding matching between matrices and segments of sequences. This problem emerges ubiquitously in DNA and protein contexts. Matrices can be used to model transcription factor binding sites and then search newly sequenced genes to check if known factors bind to their upstream region. Matrices can also be used to model protein families on the basis sequence similarity, and then they can be used to classify protein with unknown function. Despite lot of work has been devoted in past year in the definition of the criteria used to build the matrices, and to define appropriately the threshold for a match, less attention was initially posed to the actual basic problem of accelerating the searching phase. Many available softwares, in fact, were actually running the brute force sliding window solution to the problem (e.g. [14, 18, 21]). In recent years a bunch of advanced algorithms based on score properties [22], indexing data structures [3, 4], Fast Fourier Transform [15], data compression[5], matrix partitioning [8, 11, 12], filtering algorithms [3, 8, 11, 12], pattern matching [9, 11, 12, 16] have been proposed to reduce the expected time of computation. An in-depth description of all these algorithms is beyond the scope of this paper. A survey can be found in [13]. In this paper we concentrate our attention to offline methods, where it is possible to preprocess the sequence in order to obtain an index to speed up the search. In literature both suffix tree

* Work by this author was partially supported by 'Premio di Ricerca Avere Trent'Anni'

and suffix array based solution have been proposed in the past. In this paper we present a new algorithm based on the truncated suffix tree data structure [10]. We run experiments and measure both running time and space needed to search random and genomic sequences.

The reminder of the paper is as follows. In section 2 we formalize the problem. In section 3 we describe the algorithms based on suffix tree and suffix array, and present TrusSTSearch, our solution based on truncated suffix tree. In section 4 we recall the basics of the efficient coding for suffix tree in its full and truncated version already available in literature. In section 5 we present TrustCoding, a specific coding for the profile matching problem. Section 6 presents experimental comparison, on both time and space requirements, between our truncated suffix tree solution and the suffix tree and array solutions already presented in literature.

2 The Problem of Profile Matching

A position specific scoring matrix, or profile, is a representation for a weighted pattern defined over an alphabet Σ. The matrix has $|\Sigma|$ columns and m rows, where m is the length of the pattern that is represented. An example of PSSM is shown in Table 1.

	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

Table 1. A positionally weighted pattern on DNA alphabet. The pattern was obtained from the count matrix GATA-3 (MA0037) of JASPAR [17] 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 pseudocount q_a was first added to the counts for each alphabet symbol a.

Given a matrix M, and a segment $s = s_1 s_2 \dots s_m$ defined over the same alphabet, the associated score is:

$$S_c(s) = \sum_{i=1}^{m} M_{i,s_i}$$

The score gives indication of the likelihood that the segment is generated according to the model described by the matrix rather than being generated by a background distribution. Given a threshold T, if the score $S_c(s) \geq T$ then we say we have a match between the motif described by the PSSM and the segment s. The threshold T can be given directly, or it can be computed given a p-value [19]. This second option is the best choice when we search for several matrices. In fact, a fixed threshold might or might not be effective in the discrimination

between real matches and false positives or negatives, depending on the matrix itself. On the other end, the p-value allows to compute thresholds that are specific for each matrix, with the guarantee that the probability of having a random match is equal to the p-value for all of them.

Problem definition. Given a sequence $x = x_1 \dots x_n$, a matrix M of size $m \times |\Sigma|$ and a threshold T, the problem of *profile matching* is to find all position i in x where the score associated to $x_i \dots x_{i+m-1}$ according to M is greater or equal than the threshold T.

2.1 The brute-force algorithm

The basic algorithm for solving the problem of profile matching consists in computing the score for every segment of length m in the sequence. These are $n - m + 1$, and computing the score requires m steps for each. The time complexity is then $O(mn)$.

2.2 The lookahead technique

The lookahead technique (LA) is at the basis of several advanced algorithms, including those presented in the next sections. LA pre-computes 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 $T_h = T - R_h$ are computed using the *maximal remainder scores*, defined for $0 \leq h \leq m$ as

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

Let S_h be the score of the prefix of length h computed so far. The simple test: $S_h < T_h$ allows us to stop the computation of the score. In fact, in this case, there is no chance that the segment under analysis would lead to a score greater than or equal to the threshold. If the inequality does not hold, we proceed further computing the score for the following prefix $h + 1$ and we repeat the test.

Although in the worst case we still need to perform $O(m)$ operations, in practice this method allows us to drop many segments at early stages in the computation of their score, leading to substantial time savings.

3 Indexing and Profile Matching

3.1 Suffix Tree

The suffix tree is a compact trie of the suffixes of an input string x, i.e. each leaf in the tree corresponds to a suffix of x. Compactness means that runs of nodes with one child are joined together in a single arc. This implies that every internal node must be a branching node. Since there are n leaves there must be

at most $n-1$ internal nodes, hence $O(n)$ nodes in total. To know whether a word w occurs in the input string x, one has to follow the path from the root labeled with the word characters. If the path exists, then w occurs in x. The set of positions at which w occurs is given by the leaves of the subtree rooted at the node in which the path ends, or at the next following node if the path ends in the middle of an arc. In fact, words ending in the middle of an arc share the same starting positions. Word occurrence queries can be answered on a suffix tree in time linear in the length of the word. Suffix trees can be constructed in time linear in the size of the input string [20].

In [4] LA is implemented over a suffix tree. The scoring of the segments is done by a depth first traversal of the tree, calculating the score for the edge labels. When the partial score is below the intermediate threshold all the segments corresponding to positions in the subtree of the current node (or the following node if we are in the middle of an arc) can be dropped, and the search continue from the next sibling. With this approach many segments can be ignored at once, and speed up is easily achieved. The implementation of the algorithm in [4] took $17n$ bytes.

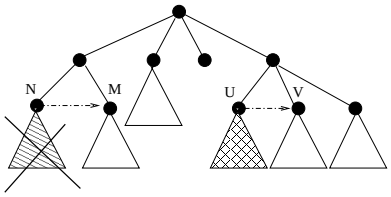

Fig. 1. An example of LA over a suffix tree. If the score at node N is below the partial threshold, then the subtree rooted at N can be skipped completely, and the search move to the next sibling M. If the score of node U is above the threshold T all the leaves that belong to the subtree rooted at U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

3.2 Suffix Array

A much more space-efficient solution to speedup profile matching comes from enhanced suffix arrays and is implemented in the PoSSuMsearch suite [3].

Enhanced suffix arrays (ESA) are as powerful as suffix trees [1] but for profile searching require only $9n$ bytes. The enhanced suffix array for profile matching consists of three tables: the classical suffix array tables *suf* (sorted suffixes) and *lcp* (longest common prefix), and a further table *skp* (skip). The suffix array can be computed in linear time [6], and once the *suf* and *lcp* tables are known, the *skp* table can also be computed in linear time. To perform profile matching as it was done with the suffix tree, one can simulate a depth first traversal of the tree by processing *suf* and *lcp* from left to right [1]. The *skp* table is used to skip uninteresting ranges corresponding to segments with prefixes that are below the partial threshold. It is worth noting in Fig.2 that skipping uninteresting ranges in the suffix array corresponds to skipping sibling nodes in the suffix tree.

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

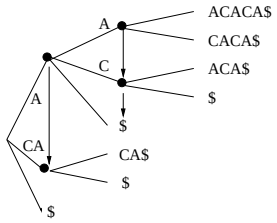

Fig. 2. An example of enhanced suffix array and the corresponding suffix tree for the same sequence S=AAACACA$.

3.3 TruSTSearch Algorithm

Experimental comparison of the suffix tree and suffix array based algorithms proved the suffix array solution to perform better. Indeed, the suffix tree carries information related to suffixes of any length up to the length of the sequences. However, at any time, only nodes that can be reached following a path of length at most equal to that of the matrix to search will ever be traversed. This suggested us the use of a *truncated suffix tree* (TST) instead. In fact, a truncated suffix tree [10], or k-factor tree [2], is a suffix tree of all the suffixes of length at most k. As a consequence, each leaf usually holds more than one positions.

When searching for hits in an indexed sequence of an entire database of matrices, we can devise two variants of the search algorithm. Let m_i be the length of the matrix M_i in the database. We can have a *full* version of the TruSTSearch algorithm where $k = \max_i m_i$, and a *filter* version where k is some fixed value, computed in such a way to fit the TST in memory. Similarly to the suffix tree based algorithm, we traverse the tree with a depth first guided by lookahead.

In the *full* variant when we reach the length m_i of the current matrix at a node N (or N is the next node if we are in the middle of an arc), if $m_i < k$ we report the set of positions associated to the leaves of the subtree rooted at N. On the other hand, if $m_i = k$ we report the positions associated with N.

In the *filter* variant we can also have the case in which $m_i > k$. In this case, when we reach a leaf, we still need to compute the score for each of the positions p associated with it. This is done by applying the lookahead technique to the $m_i - k$ symbols following each position p in the indexed sequence.

4 Efficient Suffix Tree and k-factor Tree Codings

The coding we use for our data structure is based on Kurtz coding for suffix tree [7] as adapted in [2] for k-factor trees. We report here the basic characteristics of Kurtz coding and the original coding for k-factor tree, and then explain our TruSTCoding in the next section.

4.1 Kurtz's Coding

In Kurtz's coding, internal nodes and leaves are coded using two different tables. The coding of an internal node N contains the following information:

- Right sibling of N, if present
- First child of N
- *path(N)* = length of the path from the root to N
- *head(N)* = first occurrence of *path(N)* that required the creation of N
- Suffix link of N.

Kurtz's coding distinguishes between two types of nodes: *Large nodes* and *Small nodes*. During the construction, in fact, internal nodes can be created in series. In particular, if during a phase of the construction algorithm z nodes $N_1, N_2, \ldots N_z$ are created, the first $z-1$ nodes are small nodes, while the last one, N_z, is the large node to which the small nodes refer to. For a generic small node N_i, the following properties hold: i) its suffix link points to N_{i+1}; ii) $|path(N_i)| = |path(N_z)| + (z-i)$; iii) $head(N_i) = head(N_z) - (z-i)$. Hence, they do not need to be store explicitly, allowing a strong gain in terms of space occupation. The coding of a large node uses 16 bytes, the coding of a small node uses 8 bytes. The coding of a leaf F only contains the right sibling of F. This information is enough since *head(F)* $= F$ and $|path(F)| = |S| - F$. Every leaf requires 4 bytes. Kurtz's coding requires about 12 bytes per symbol.

4.2 The Original Coding for k-Factor Trees (OC).

The k-factor tree coding proposed in [2] is a modification of Kurts'z coding. The coding of a node is modified in order to free bits in the coding scheme of a leaf. In particular, leaves have two free bits that are used by nodes to store their suffix link. Let us consider a generic node N. If N is a small node its suffix link is not stored but it is deduced directly from the node. If N is a large node the suffix link has to be stored. If the id of N is less or equal than 2^{26}, the suffix link points to a node that requires 26 bits for coding and the coding scheme is the same. Otherwise the suffix link could point to a node whose number is at least 2^{26} an alternative scheme is used instead.

Leaves coding is strongly modified with respect to Kurtz's coding, because they code in a linked list the leaves of the subtree rooted at that node in the full suffix tree. The coding, then, distinguishes between two types of leaf: the rightmost leaf of a linked list, called master leaf, and the ones preceding it. A bit is used to discriminate between the two. For the master leaf two alternative codings are possible depending on whether it has a sibling or not. To guarantee that the master leaf is reached in constant time (and so the right sibling, if present, since it is coded in the master leaf), 4 bits are used in the other leaves to code part of the master leaf id. In this way, after traversing the first 8 leaves, one can reach the master leaf.

5 The TrustSearch Coding (TC)

TruSTsearch is based on a depth-first traversal of the tree, guided by lookahead technique. The information it needs, in particular, is the following.

- For an internal node N: *head(N)*, $|path(N)|$, first child of N, and sibling of N (if present);
- For a leaf F: *head(F)*, $|path(F)|$, sibling of F (if it is a leaf at the end of a linked list and if it has a sibling).

The algorithm never uses suffix links so we can free the bits needed to store this information from the original coding, and use the recovered space to make further optimizations. Some values, in particular, are little enough to be stored using a smaller number of bits. Suppose, for example, that the successor of a leaf in a linked list is less than 2^{19}. It can be stored using 19 bits thus saving 1 byte. This approach, however, does not guarantee a strong gain in terms of space occupation. For example, if the number of leaves $l = 2^{23}$ (that is, an 8MB sequence), only $2^{19}/2^{23} = 6.25\%$ of leaves uses the reduced space. We can obtain further space reduction by storing the difference between the leaf and the successor rather than using the successor itself. This approach can be extended to *sibling*, *first_child* and *head*. We need, however, additional control bits. In the case of internal nodes, we can use the space recovered from suffix links. In the case of a leaf there is no free space, and the coding need to be modified.

5.1 Leaves Coding.

TruSTCoding distinguishes between three types of leaves:

1. Master leaf: the rightmost leaf in a linked list
2. Small leaf: intermediate leaves in a linked list (it could not be present)
3. Large leaf: first leaf in a linked list (it could not be present)

Consider a linked list with at least three leaves. The rightmost leaf is a master leaf and, as in the original coding, it points the right sibling (if present). The other leaves point to their successor. Storing the difference between a leaf and its successor allow us to save 1 byte if its value is small enough. Otherwise we use 4 bytes and store directly the successor of the leaf. We need a control bit to discriminate between the two cases. The coding of a leaf has no free bits, however we can use the 4 bits coding part of the index of the master leaf. In this way we loose the possibility to reach the master leaf in constant time so, in the first leaf of the linked list, we add a pointer to the master leaf (that is why it is called large leaf). In a linked list with just one leaf this is a master leaf. In a linked list with two leaves the rightmost one is a master leaf but the first, rather of being a large leaf, is a small leaf. The reason is that its successor is the master leaf, so another pointer to the master leaf would be useless.

Summarizing, every linked list has a master leaf. Small leaves are present if and only if the linked list has at least two leaves. Large leaves are present if and

only if the linked list has at least three leaves. The first bit of a leaf is flagged if it is a master leaf, it is not otherwise. If it is not a master leaf, the second bit is flagged if it is a large leaf, it is not if it is a small leaf.

Master leaves coding. If a master leaf has no sibling, its coding requires 1 byte. Otherwise the second bit of the first byte is flagged, and 3 or 4 bytes are needed. Let *diff* be the difference between the id of the leaf and the id of its sibling. The third bit of a master leaf can be used to discriminate whether $|diff| < 2^{19}$ or not. If so we use 19 bits. We need one additional bit to know the sign of *diff*, and another one to know if the sibling is a node or a leaf. These information are enough to recover the right sibling and save 1 byte. If $|diff| \geq 2^{19}$ we still need 4 bytes, so the control bit is flagged and we store the right sibling.

Small leaf coding. The first two control bits are not flagged in the case of a small leaf. Let *diff* be the difference between the id of the leaf and the id of its successor. We save 1 byte storing *diff* rather than the id of the successor if $|diff| < 2^{21}$. We also need 1 additional bit to discriminate whether $diff < 2^{21}$ or not. Since all leaves in a linked list are ordered in decreasing order $diff > 0$, we do not need a sign bit. If $diff \geq 2^{21}$ we still need 4 bytes, so the control bit is flagged and the full id of the successor is stored.

Large leaves coding. Since we do not use anymore the 4 bits coding part of the master leaf, the only way to reach it is to traverse all the leaves of the linked list. Consider the following cases:

1. A match is found at a branching node N (or in the middle of an arc whose following node is N) and the master leaf is in the subtree rooted at N
2. The master leaf has been reached, but its depth is less than the pattern length
3. Lookahead stops in the middle of the arc connecting to the linked list.

In the first case, when we reach the beginning of the linked list (that is, the large leaf), we traverse all the leaves of the linked list, checking if they are matching positions. When we reach the end of the linked list (that is, the master leaf) we can jump to the right sibling, if present. In the second case, we have to calculate the score of all leaves of the linked list, guided by lookahead technique, and when we reach the end of the linked list (that is, the master leaf) we can jump to the right sibling, if present. The third case is more tricky. When lookahead technique stops, in fact, we have to jump directly to the right sibling, if present. This information, however, is stored in the last cell of the linked list (the master leaf). Since the arc links to the first cell (the large leaf), the only way to know the right sibling is to traverse the whole linked list, so we cannot jump to the right sibling in constant time. To prevent case 3 to occur, and so to guarantee access to the right sibling in constant time also in this case, we add at the large leaf an additional pointer to the master leaf (this is why it is called large). We use additional control bits to discriminate the various cases. This coding requires between 6 and 8 bytes.

5.2 Nodes Coding

Nodes coding is strongly modified with respect to the original coding. The main differences are the following:

- It does not store the suffix link because TruSTsearch does not need it;
- It frees useless bits when the node has no right sibling;
- It stores, whenever possible, differences from values rather than the values themselves.

Small nodes coding. We recall that a small node N needs to store the right sibling (if present), the first child and the distance to the large node with which it is associated. $head(N)$ and $path(N)$ can then be retrieved from the large node. We only use 3 bits for coding the distance to the large node rather than 5, so the maximal value that can be stored in this field is 7. This is not a problem even when the real distance is above 7. In this case, in fact, it is possible to follow a chain of small nodes at distance 7 one from the other, until the large node is reached. This means that the large node can be reached after at most $\lceil 31/7 \rceil = 5$ jumps, that is, $head(N)$ and $|path(N)|$ for a small node N can still be retrieved in constant time. Additional bits are used to discriminate the various cases. The number of bytes required for coding a small node varies between 4 and 8 bytes.

Large nodes coding. The coding for large nodes uses 23 bits to store each difference between the ids of the current node and its sibling, its first child, and head. The bits used for coding the path length are at least 7, which is a reasonable bound for a profile matching problem where the length of the matrix is at most around 30. This coding requires between 8 and 15 bytes, depending on the various cases.

6 Experiments

Experiments were run on the Jaspar CORE database[17] for transcription factor binding sites. The database consists of 138 matrices with average length 10.67

The matrices were searched both on randomly generated and real DNA sequences of increasing length to test scalability. TruSTsearch was developed in C. We used this software to collect data for both the full suffix tree and its truncated version. For the suffix array based algorithm we run the software PoSSuMsearch. Experiments were run on a 3GHz processor, with 4GB of memory, under Linux.

6.1 Comparison of Space Efficiency

In Fig.5 we compare the space required by the original coding for k-factor tree (OC) and our proposed TruSTCoding (TC). We varied the length of the truncation factor k and also reported the results for the full suffix tree, obtained by building the tree with k equals to the sequence length. We remind here that

the enhanced suffix array requires always 9 bytes per symbol. We scanned both random sequences generated following the uniform distribution, and genomic sequences.

Random sequences								
uniform distribution								
Len	1024		1048576		16777216		67108864	
k	OC	TC	OC	TC	OC	TC	OC	TC
2	4.13	3.13	4.00	3.00	4.00	3.00	4.00	3.00
3	4.35	3.40	4.00	3.00	4.00	3.00	4.00	3.00
4	5.33	4.33	4.00	3.00	4.00	3.00	4.00	3.00
5	8.64	5.81	4.00	3.00	4.00	3.00	4.00	3.00
6	11.34	7.08	4.02	3.02	4.00	3.00	4.00	3.00
7	12.16	7.49	4.08	3.10	4.01	3.00	4.00	3.00
8	12.41	7.63	4.32	3.41	4.03	3.03	4.01	3.00
9	12.46	7.65	5.28	4.39	4.08	3.12	4.02	3.03
10	12.47	7.66	8.55	5.85	4.32	3.59	4.08	3.25
11	12.48	7.67	11.46	7.24	5.28	4.95	4.32	4.04
12	12.49	7.67	12.31	7.69	8.55	6.50	5.28	5.30
13	12.49	7.68	12.51	7.81	11.47	7.58	8.55	6.79
14	12.49	7.68	12.56	7.83	12.32	8.17	11.47	8.26
15	12.49	7.68	12.57	7.84	12.52	8.27	12.32	8.72
16	12.49	7.68	12.57	7.84	12.56	8.29	12.52	8.84
17	12.49	7.68	12.57	7.84	12.58	8.30	12.56	8.86
18	12.49	7.68	12.58	7.84	12.58	8.30	12.58	8.87
19	12.49	7.68	12.58	7.84	12.58	8.30	12.58	8.87
20	12.49	7.68	12.58	7.84	12.58	8.30	12.58	8.87
25	12.49	7.68	12.58	7.84	12.58	8.30	12.58	8.87
30	12.49	7.68	12.58	7.84	12.58	8.30	12.58	8.87
Full	12.49	7.68	12.58	7.84	12.58	8.30	12.58	8.87

	T.Guttata		O.Sativa		H.Sapiens		C.Lupus		T.Guttata	
Chr	13		4		19		15		1	
Len	16962381		35498469		55858983		64211953		118548696	
k	OC	TC	OC	TC	OC	TC	OC	TC	OC	TC
2	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00
3	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00
4	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00
5	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00
6	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00
7	4.00	3.01	4.00	3.00	4.00	3.00	4.00	3.00	4.00	3.00
8	4.02	3.03	4.01	3.01	4.01	3.01	4.01	3.01	4.00	3.00
9	4.08	3.13	4.04	3.07	4.02	3.05	4.02	3.05	4.01	3.03
10	4.30	3.50	4.15	3.35	4.10	3.25	4.08	3.21	4.05	3.15
11	5.11	4.29	4.60	4.09	4.37	3.76	4.32	3.73	4.17	3.35
12	6.79	5.38	6.06	5.19	5.20	4.58	5.06	4.63	4.58	4.25
13	8.70	6.41	8.24	6.35	6.91	5.57	6.83	5.76	5.68	5.28
14	10.15	7.12	9.69	7.10	8.62	6.46	9.18	6.97	7.98	6.58
15	11.03	7.52	10.34	7.44	9.56	6.97	10.67	7.75	10.29	7.73
16	11.51	7.75	10.62	7.59	9.99	7.22	11.33	8.10	11.47	8.33
17	11.77	7.87	10.76	7.67	10.21	7.35	11.60	8.26	11.90	8.57
18	11.90	7.94	10.86	7.72	10.36	7.45	11.72	8.33	12.05	8.65
19	11.96	7.97	10.92	7.75	10.48	7.52	11.80	8.38	12.11	8.69
20	12.00	7.99	10.98	7.78	10.58	7.59	11.85	8.41	12.15	8.71
25	12.09	8.04	11.20	7.90	11.01	7.88	12.02	8.52	12.22	8.75
30	12.13	8.06	11.35	7.98	11.34	8.08	12.12	8.58	12.26	8.77
Full	12.47	8.24	12.43	8.48	12.28	8.61	12.47	8.79	12.57	8.96

Fig. 3. Space occupation (bytes per symbol) comparison between original coding (OC) and TruSTCoding (TC) indexing random sequences (left) and genomic sequences (right). An enhanced suffix array always uses 9 bytes per symbol.

It can be noticed that TrustCoding always performs better than the original coding. Moreover, it tends to reach the space occupation of the enhanced suffix array only in the worst case.

6.2 Comparison of Time Efficiency

In this section we show the results on running time of TruSTsearch, the full suffix tree and PossumSearch. We report results on real biological sequences, under two different choices of p-values for the threshold: 10^{-6} and 10^{-3}. Since the actual matrices that are used in the scanning are the result of a pre-processing stage in which the count matrices are transformed into PSSM applying pseudo-counts, logarithms, rounding, etc., the matrices scanned by PoSSuMsearch might be slightly different from the ones we used. As a consequence the number of hits might also be different. Reporting a hit is a time consuming operation (which include writing to disk), so for a fair comparison we divided the time of computation by the number of matches found by each algorithm and report the time required to find 1000 matches.

Name	T.Guttata	O.Sativa	H.Sapiens
Chr	13	4	19
Length	16962381	35498469	55858983
ESAsearch	0.48	0.49	0.19
TruSTsearch/k			
5	0.35	0.33	0.24
6	0.28	0.29	0.20
7	0.25	0.18	0.16
8	0.21	0.13	0.12
9	0.15	0.14	0.09
10	0.12	0.11	0.08
11	0.13	0.11	0.07
12	0.13	0.10	0.07
13	0.13	0.10	0.07
14	0.14	0.10	0.07
15	0.13	0.14	0.07
16	0.13	0.19	0.07
17	0.13	0.37	0.07
18	0.16	0.49	0.07
19	0.13	0.46	0.07
20	0.18	0.46	0.07
25	0.14	0.48	0.07
30	0.13	0.49	0.07
Full	0.16	0.51	0.08

Name	T.Guttata	O.Sativa	H.Sapiens
Chr	13	4	19
Length	16962381	35498469	55858983
ESAsearch	33.26	27.27	39.59
TruSTsearch/k			
5	15.59	31.92	18.63
6	13.93	24.12	16.27
7	11.32	21.59	13.92
8	9.16	18.92	11.58
9	7.15	14.32	9.81
10	6.37	11.15	7.78
11	5.83	9.96	6.67
12	5.93	9.18	6.38
13	6.55	9.80	6.20
14	6.90	10.00	6.50
15	6.90	10.60	6.70
16	6.96	10.60	6.63
17	7.01	9.61	6.80
18	7.10	10.88	6.66
19	6.94	10.84	6.66
20	6.89	10.30	6.61
25	7.10	10.91	6.64
30	6.97	11.07	6.90
Full	7.11	11.51	6.94

Fig. 4. Running Time per 1000 matches (sec) on biological sequences for p-value=10^{-6} (left) and p-value=10^{-3} (right).

It can be noticed that TruSTsearch reaches the best performances around a factor k equal to 10 or 11, which is in accordance to the average length of the matrices in the database. For smaller truncation factors, in fact, the algorithms works more as a filter and the remaining positions are checked scanning one symbol at the time guided by lookahead. In most cases however, in our experiments on the Jaspar Core the performances of TruSTsearch were better than POSSUMSearch.

Conclusions

We presented TruSTsearch a profile matching algorithm based on the truncated suffix tree. We also propose a specific coding that allows us to obtain considerable space saving not only with respect to the adaption of Kurtz's coding for truncated suffix tree, but also with respect to enhanced suffix array. Time performances on the tested genomic sequences showed that our algorithm is faster than other suffix-based algorithms, reaching the best performances with a truncation factor around 10. In the future we plan to experiment also on different contests, such as protein sequences, to see how the size of the alphabet influences the performances of our method, and on different matrices databases, to measure the impact of the matrix length distribution on the performances.

References

1. Abouelhoda M., Kurtz S., and Ohlebush E., Replacing Suffix Trees with Enhanced Suffix Arrays, *Journal of Discrete Algorithms*, 2004; 2:53–86.
2. Allali J., Sagot M.-F., The at most k-deep factor tree, tech report 2004-03, Institut Gaspard Monge, Université de Marne la Vallée.

3. 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; 7(1):389.
4. Dorohonceanu B., and Neville-Manning C.G., Accelerating Protein Classification Using Suffix Trees, in *Proc. of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, 2000; 128–133.
5. Freschi V., and Bogliolo A., Using Sequence Compression to Speedup Probabilistic Profile Matching, *Bioinformatics*, 2005; 21(10):2225–2229.
6. Kärkkäinen J., and Sanders P., Simple Linear Work Suffix Array Construction, in *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, LNCS 2719, 943–955, Springer-Verlag 2003.
7. Kurtz S., Reducing the Space Requirements of Suffix Trees, *Software - Practice and Experience*, 1999; 29(13):1149–1171.
8. Liefhooghe A., Touzet H. and Varré J.-S.: Large Scale Matching for Position Weight Matrices. In: *Proc. of CPM 2006*, LNCS 4006, 401–412, Springer-Verlag 2006.
9. Liefooghe A., Touzet H., Varré J.-S., Self-overlapping Occurrences and Knuth-Morris-Pratt Algorithm for Weighted Matching. *Proc. of LATA 2009*, LNCS 5457, 481–492, Springer-Verlag 2009.
10. Na J.C., Apostolico A., Iliopoulos C.S., Park K., Truncated Suffix Trees and their Application to Data Compression, *Theoretical Computer Science*, 2003; 304(1-3):87–101.
11. Pizzi C., Rastas P., and Ukkonen E., Fast Search Algorithms for Position Specific Scoring Matrices, in *Bioinformatics Research and Development Conference (BIRD 2007)*, LNBI 4414, 239–250, Springer-Verlag 2007.
12. Pizzi C., Rastas P., and Ukkonen E., Finding significant matches of position weight matrices in linear time, in *IEEE Transaction on Bioinformatics and Computational Biology*, 12 Mar. 2009. IEEE computer Society Digital Library. IEEE Computer Society, http://doi.ieeecomputersociety.org/10.1109/TCBB.2009.35
13. Pizzi C., Ukkonen E., Fast Profile Matching Algorithms - A survey. *Theoretical Computer Science*, 2008; 395(2-3): 137–157.
14. 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.
15. 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.
16. Salmela L. and Tarhio J., Algorithms for weighted matching, in Proc. SPIRE 2007, LNCS 4726, 276–286. Springer-Verlag 2007.
17. 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* 2004; 32: D91–D94.
18. Scordis P., Flower D.R., and Attwood T., FingerPRINTScan: Intelligent Searching of the PRINTS Motif Database, *Bioinformatics*, 1999; 15(10):799–806.
19. Staden R., Methods for Calculating the Probabilities of Finding Patterns in Sequences, *CABIOS*, 1989; 5(2):89–96.
20. Ukkonen, E., On-line Construction of Suffix Trees, *Algorithmica*, 1995; 14(3): 249–260.
21. 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.
22. 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.