BIOINFORMATICS ORIGINAL PAPER

Vol. 30 no. 4 2014, pages 464–471
doi:10.1093/bioinformatics/btt706

Sequence analysis

Advance Access publication December 11, 2013

slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array

Francisco Fernandes[1] and Ana T. Freitas[1,2,*]

[1]Knowledge Discovery and Bioinformatics Group (KDBIO), Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento (INESC-ID), Rua Alves Redol, 9, 1000-029 Lisbon and [2]Department of Computer Science and Engineering, Instituto Superior Técnico (IST) – Universidade de Lisboa, Avenida Rovisco Pais, 1, 1049-001 Lisbon, Portugal

Associate Editor: Martin Bishop

ABSTRACT

Motivation: Maximal exact matches, or just MEMs, are a powerful tool in the context of multiple sequence alignment and approximate string matching. The most efficient algorithms to collect them are based on compressed indexes that rely on longest common prefix array-centered data structures. However, their space-efficient representations make use of encoding techniques that are expensive from a computational point of view. With the deluge of data generated by high-throughput sequencing, new approaches need to be developed to deal with larger genomic sequences.

Results: In this work, we have developed a new longest common prefix array-sampled representation, optimized to work with the backward search method inherently used by the FM-Index. Unlike previous implementations that sacrifice running time to have smaller space, ours lead to both a fast and a space-efficient approach. This implementation was used by the new software slaMEM, developed to efficiently retrieve MEMs. The results show that the new algorithm is competitive against existing state-of-the-art approaches.

Availability and implementation: The software is implemented in C and is operating system independent. The source code is freely available for download at http://github.com/fjdf/slaMEM/ under the GPLv3 license.

Contact: atf@inesc-id.pt

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on August 11, 2013; revised on November 20, 2013; accepted on November 28, 2013

1 INTRODUCTION

With the new high-throughput sequencing technologies becoming faster, cheaper and more accurate, the number of available genomes is growing fast. Metagenomics is also pushing forward the need to align new sequences to those already known to compare different strains or assemblies, build phylogenetic trees, identify new genes, identify mutations or polymorphisms, observe structural variations and perform other relevant operations. It is well known that dynamic programming approaches are prohibitive, both in terms of required memory and processing time, when aligning large genomes or a number of different genomes. To approach these problems, strategies using seeded alignments with shared segments, which are identical among the sequences and act as anchor points for the alignment, have been developed. These anchors can be fixed-length exact matches, or k-mers, as those used in the BLAST (Altschul *et al.*, 1990) tool. However, this type of match is inefficient because it can lead to an oversized number of hits, and these still have to be extended in both directions using pairwise comparisons, implying a significant processing time. Much more efficient is the identification of maximal unique matches (MUMs) that have been introduced first by MUMmer (Delcher *et al.*, 1999). MUMs are identical substrings that occur exactly once in each sequence and whose occurrences cannot be extended to either side without producing a mismatch. The second version of MUMmer (Delcher *et al.*, 2002) introduced a new more compact suffix tree (ST) representation, and the third and last one (Kurtz *et al.*, 2004) added the ability to output maximal exact matches (MEMs). These are similar to MUMs but can occur any number of times, which is useful when the number of MUMs is insufficient to produce enough anchors for a solid alignment, e.g. when many repeated regions exist. Also, using MEMs instead of MUMs multiplies the regions covered by anchors, reducing considerably the areas requiring further processing. However, the bottleneck of MUMmer is the memory requirements of its ST index structure, which can become problematic when it does not fit into the main memory. Other closed-source tools based on enhanced suffix arrays (ESAs) such as Vmatch (Abouelhoda *et al.*, 2004) and CoCoNUT (Abouelhoda *et al.*, 2008) have also been released, but they share the same problem. For this reason, and to deal with larger sequences, other approaches to find MEMs have been developed. The sparseMEM approach (Khan *et al.*, 2009) makes use of a sparse SA as an index, which trades memory space for extra computational time. Later, backwardMEM (Ohlebusch *et al.*, 2010) used a backward search method over a compressed ESA. More recently, essaMEM (Vyverman *et al.*, 2013) improved sparseMEM by enhancing it with a sparse child array that reduces computational time maintaining the same memory footprint. This method currently shows the best trade-off between time and memory consumption for MEM identification.

In this work, we propose another approach as an alternative to these previous tools. We have developed a new sampled representation of the longest common prefix (LCP) array, optimized to work with the backward search method inherent from the

*To whom correspondence should be addressed.

FM-Index. Results show the effectiveness of the new method for a number of different genomes.

1.1 Basic notions

STs (Weiner, 1973) are fundamental data structures in the field of string processing. However, despite being linear (Ukkonen, 1995), their space requirements are large. Consequently, they have been progressively replaced by SAs (Manber and Myers, 1993) and more recently by more space-efficient Burrows–Wheeler Transform (BWT) (Burrows and Wheeler, 1994)-based indexes, namely, the FM-Index (Ferragina and Manzini, 2000). Although these more advanced indexes work just fine for standard pattern matching, they lack certain functionalities originally present in STs, including the possibility to follow suffix links. To overcome these limitations, other data structures have been proposed, like the ESA (Abouelhoda *et al.*, 2004), which extends the original SA with additional information to simulate the behavior of STs. Other alternatives are reviewed in (Navarro and Mäkinen, 2007) and (Vyverman *et al.*, 2012).

Let $\Sigma = \{\alpha_1, \ldots, \alpha_{|\Sigma|}\}$ be a finite ordered *alphabet*, and Σ^* be the *set of all strings* over Σ, including the empty string ε. Let T be a string or *text* over Σ^*, which is always terminated by a special character \$, which is lexicographically smaller than any character in Σ and does not occur anywhere else in T. Let T[i] denote the *character at position i* in T, for $0 \le i < n$, where $n = |T|$. This way, we define T[i...j] as the *substring* of length $(j - i + 1)$ starting at the *i*th position and ending at the *j*th position of T, where $0 \le i \le j < n$. We call T_i the *i*th *suffix* of T, i.e. the substring $T[i \ldots (n-1)]$, with $0 \le i < n$. In the same way, the substring $T[0 \ldots i]$, $0 \le i < n$ corresponds to a *prefix* of T.

1.2 STs and SAs

The ST of T is a rooted tree that represents all the non-empty suffixes of T in the following compact way. Each node has a label corresponding to a substring that occurs in T. The special top node is called the root and corresponds to the empty string. Each internal node has at least two children, and no two children branching from the same node can have labels starting with the same character. Each node can also be identified by its path label, i.e. the string obtained by concatenating all the node labels on the path from the root down to that node. The tree has exactly *n* leaves, corresponding to the *n* suffixes of T, where the path label of the *i*th leaf spells the *i*th suffix. A suffix link is a pointer that connects a node to its subsequent suffix node, i.e. it associates each node with path label $\alpha\omega$, such that $\alpha \in \Sigma$ and $\omega \in \Sigma^*$, to the node whose path label is ω.

The SA of T is an array of size *n* of numbers corresponding to the lexicographical ordering of the *n* suffixes of T, i.e. a permutation of the integers $\{0, \ldots, (n-1)\}$ such that $T_{SA[0]} < T_{SA[1]} < \ldots < T_{SA[n-1]}$. The SA takes $O[n^*\log(n)]$ bits of space and can be built using linear time and space (Kärkkäinen and Sanders, 2003; Kim *et al.*, 2003; Ko and Aluru, 2003; Nong *et al.*, 2009). Taking advantage of the SA as an index structure, a pattern P can be matched in $O[m^*\log(n)]$ time using binary search. The term *ω-interval* (Abouelhoda *et al.*, 2004) is often used to denote the interval in the index obtained from matching the string ω.

1.3 BWT and FM-Index

The BWT of T is a permutation of the characters of T such that BWT[i] corresponds to the character preceding the *i*th lexicographically ordered *rotation* of T, i.e. $BWT[i] = T[SA[i]-1]$ if $SA[i] \neq 0$ and BWT[i] = \$ otherwise. If we consider the conceptual matrix M consisting of all the sorted rotations of T, the BWT array corresponds to the last column of M. In the example of Figure 2, the BWT of the previously illustrated string (Fig. 1) is 'TCACCG\$GAATAGC'. The BWT array takes $O[n^*\log(|\Sigma|)]$ space and can be constructed in linear time and space, e.g. using the induced sorting approach from Okanohara and Sadakane (2009), among others.

Using the BWT together with some extra information, we can build another index structure called the *FM-Index* (Ferragina and Manzini, 2000). One of its key concepts is the *Last-to-First column mapping* (LF-mapping), which finds, for each position i, the position j such that $SA[j] = (SA[i] - 1) \pmod n$. Like the name suggests, it simply maps the *k*th occurrence of each symbol in the last column L to the *k*th occurrence of the same symbol in the first column F. In other words, and noting that the BWT is in fact the last column L, if $L[i] = BWT[i] = T[(SA[i] - 1)(\mathrm{mod}\ n)] = c$ is the *k*th occurrence of the character c in the last column L, then we will have $LF[i] = j$ where $F[j] = T[SA[j]] = c$ is the *k*th occurrence of the same character in the first column F. For the example in Figure 2, $LF[0] = 12$ for character 'T' and $LF[1] = 5$ for character 'C'. The LF-mapping can be efficiently computed by setting: $LF[i] = C[c] + occ(c,i) - 1$, where:

- $c = BWT[i]$
- C[c] is the total number of occurrences in T of all the characters strictly smaller than c
- occ(c,i) is the number of occurrences of c in BWT[0...i]

Pattern matching on a pattern P of size m is done in O(m) time according to the *BackwardSearch* procedure (Ferragina and Manzini, 2005) detailed in the Supplementary Material. The search is performed backward by iteratively applying the LF-mapping rule to obtain the $P[i \ldots (m-1)]$-interval from the $P[(i+1) \ldots (m-1)]$-interval, for $0 \le i < (m-1)$.

i	SA[i]	$T_{SA[i]}$
0	13	\$
1	4	AACTGCAGT\$
2	5	ACTGCAGT\$
3	1	AGCAACTGCAGT\$
4	10	AGT\$
5	3	CAACTGCAGT\$
6	0	CAGCAACTGCAGT\$
7	9	CAGT\$
8	6	CTGCAGT\$
9	2	GCAACTGCAGT\$
10	8	GCAGT\$
11	11	GT\$
12	12	T\$
13	7	TGCAGT\$

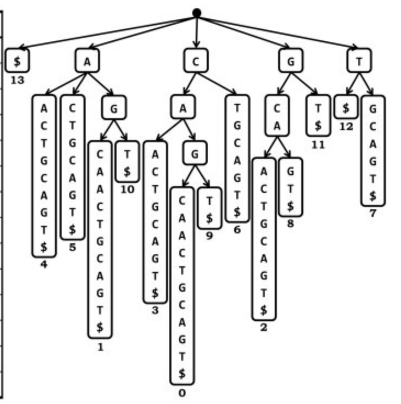
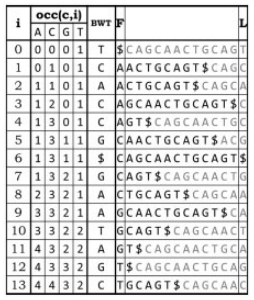
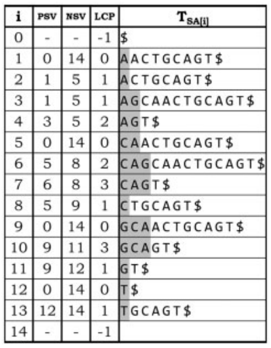


Fig. 1. SA and ST for the string *CAGCAACTGCAGT\$*. Each leaf node in the ST corresponds to an SA entry and vice versa, while internal nodes correspond to shared prefixes among consecutive ordered suffixes



i	occ(c,i)				BWT	F		L
	A	C	G	T				
0	0	0	0	1	T	$	CAGCAACTGCAG	T
1	0	1	0	1	C	A	ACTGCAGT$CAG	C
2	1	1	0	1	A	A	CTGCAGT$CAGC	A
3	1	2	0	1	C	A	GCAACTGCAGT$	C
4	1	3	0	1	C	A	GT$CAGCAACTG	C
5	1	3	1	1	G	C	AACTGCAGT$AC	G
6	1	3	1	1	$	C	AGCAACTGCAGT	$
7	1	3	2	1	G	C	AGT$CAGCAACT	G
8	2	3	2	1	A	C	TGCAGT$CAGCA	A
9	3	3	2	1	A	G	CAACTGCAGT$C	A
10	3	3	2	2	T	G	CAGT$CAGCAAC	T
11	4	3	2	2	A	G	T$CAGCAACTGC	A
12	4	3	3	2	G	T	$CAGCAACTGCA	G
13	4	4	3	2	C	T	GCAGT$CAGCAA	C

Fig. 2. Matrix of all the rotations of the string *CAGCAACTGCAGT$* evidencing its *BWT*, the first and last columns *F* and *L* and the counts of each alphabet character *c* along the BWT in *occ(c,i)*. The last characters of the rotations are grayed out to show the difference from each corresponding suffix



i	PSV	NSV	LCP	$T_{SA[i]}$
0	-	-	-1	$
1	0	14	0	AACTGCAGT$
2	1	5	1	ACTGCAGT$
3	1	5	1	AGCAACTGCAGT$
4	3	5	2	AGT$
5	0	14	0	CAACTGCAGT$
6	5	8	2	CAGCAACTGCAGT$
7	6	8	3	CAGT$
8	5	9	1	CTGCAGT$
9	0	14	0	GCAACTGCAGT$
10	9	11	3	GCAGT$
11	9	12	1	GT$
12	0	14	0	T$
13	12	14	1	TGCAGT$
14	-	-	-1	

Fig. 3. LCP, PSV and NSV arrays for the string *CAGCAACTGCAGT$*. The shaded characters represent the shared prefixes between adjacent suffixes

Because the BWT stores characters and not numbers, the FM-Index space requirements of O(n*log(|Σ|)) are much lower than those of ST and SA, O(n*log(n)). More specifically, these values are typically ~10*n–20*n bytes for ST and 5*n bytes for SA, assuming 32-bit integers (Kurtz, 1999).

1.4 LCP array and lcp-intervals

The ESA can simulate all the functionality of the original ST while improving the SA's pattern matching time to O(m), and consists of the basic SA augmented with two additional structures, the *LCP array* and the *lcp interval tree* represented by a *child table*.

The LCP of T is an array of numbers with size *n* that stores the length of the lcp between each suffix and the previous one, i.e.

- $\mathrm{LCP}[i] = |\ \mathrm{lcp}(T_{SA[i-1]}, T_{SA[i]})\ |$ for $0 < i \le n$
- $\mathrm{LCP}[0] = (-1)$

It can be built in both linear time and space using the SA (Kasai *et al.*, 2001). An *lcp-interval* with *lcp-value l* is named an *l-interval* and is denoted by $l-[i,j]$, where $0 \le i < j \le (n-1)$ and the following properties hold:

- $\mathrm{LCP}[i] < l$
- $\mathrm{LCP}[k] \ge l$ for all k with $(i + 1) \le k \le j$
- $\mathrm{LCP}[k] = l$ for at least one k with $(i + 1) \le k \le j$
- $\mathrm{LCP}[j+1] < l$ if $j \ne (n-1)$

Consequently, if we have an *lcp-interval* $l-[i,j]$, this means that the substring $T[(SA[i])\ldots(SA[i]+l-1)]$ of size *l* is the longest common prefix between all the $(j-i+1)$ suffixes $T_{SA[i]}\ldots T_{SA[j]}$ of that SA interval. Note that because the *lcp* is calculated between the current SA position and its predecessor, the first position *i* of an *lcp-interval* $l-[i,j]$ always has an *lcp-value* lower than *l*. In this way, it is sometimes useful to refer to the *depth* of an interval or a single position, which for an *lcp-interval* $l-[i,j]$ is always *l*, but for a general *ω*-interval given by [i,j], it can be obtained from the LCP as:

$$\mathrm{Depth}([i, j]) = \begin{cases} \max\{\mathrm{LCP}[i], \mathrm{LCP}[i + 1]\}, & \text{if } i = j \\ \min\{\mathrm{LCP}[i + 1], \mathrm{LCP}[j]\}, & \text{if } i \ne j \end{cases}$$

The *parent interval* of an *lcp-interval* $l-[i,j]$ is an *lcp-interval* $q-[r,s]$ such that $q < l$, $r \le i$ and $s \ge j$, and there is no other *lcp-interval* of *lcp-value t* enclosing $l-[i,j]$ such that $q < t < l$. Therefore, it corresponds to the first larger *lcp-interval* that encloses $l-[i,j]$. To calculate a parent interval, we can use the next smaller value (NSV) and the previous smaller value (PSV) arrays introduced in Fischer *et al.* (2009) and defined as:

- $\mathrm{PSV}[i] = \max\{k : (0 \le k < i \text{ and } \mathrm{LCP}[k] < \mathrm{LCP}[i]) \text{ or } k = 0\}$
- $\mathrm{NSV}[i] = \min\{k : (i < k \le n \text{ and } \mathrm{LCP}[k] < \mathrm{LCP}[i]) \text{ or } k = n\}$

As their names suggest, the PSV/NSV arrays contain the first position above/below in the LCP array that has an *lcp-value* lower than the current one, respectively. Because in an *lcp-interval* $l-[i,j]$ the nearest *lcp-values* lower than *l* are located at LCP[i] and LCP[j + 1], its first enclosing interval will have an *lcp-value* equal to the higher of these two values. Therefore, the resulting parent interval is defined by:

$\mathrm{Parent}(l - [i,j]) = (\mathrm{LCP}[k]) - [\mathrm{PSV}[k], (\mathrm{NSV}[k] - 1)]$, with

- $k = i$, if $\mathrm{LCP}[i] \ge \mathrm{LCP}[j + 1]$
- $k = j + 1$, if $\mathrm{LCP}[j + 1] > \mathrm{LCP}[i]$

Some illustrative *lcp-intervals* in Figure 3 make this more clear, e.g. the parent of the 3–[6,7] interval, corresponding to the 'CAG'-interval, is the 2–[5,7] interval corresponding to the 'CA'-interval, and *lcp-interval* 3–[9,10] has 1–[9,11] as parent, coinciding with the 'GCA' and 'G' intervals, respectively.

From the parent/child relations of the *lcp-intervals*, an *lcp-interval tree* can also be built to simulate the topography of the ST. Instead of pre-computing and storing the PSV/NSV arrays explicitly, they can also be calculated on the fly using Range Minimum Queries (RMQ) (Fischer and Heun, 2007) that rely on other auxiliary data structures. Although many applications exist that use these last two methods, they have not been applied in this work.

1.5 LCP array representations

Some direct representations of the LCP include storing each value using only 1 byte with the larger values going in a separate array (Abouelhoda *et al.*, 2004), or encoding it with a wavelet tree using the different lcp values as the alphabet (Kulekci *et al.*, 2012).

A succinct representation of the LCP exists (Sadakane, 2007), based on the fact that the property $\mathrm{LCP}[\mathrm{SA}^{-1}[j+1]] \geq \mathrm{LCP}[\mathrm{SA}^{-1}[j]] - 1$ always holds. In this case, $\mathrm{SA}^{-1}[j] = i$ gives the position i in the SA where the text position j is stored, i.e. $\mathrm{SA}[i] = j$. This means that if we rearrange the LCP entries in text order rather than lexicographic order, the values decrease by at most 1. From this observation, a new array of size *n* defined by $\mathrm{Hgt}[j] = (j + \mathrm{LCP}[\mathrm{SA}^{-1}[j]])$ and composed entirely of non-decreasing integers can be constructed and stored using only $2n + O(n)$ bits, with a special data structure used to encode sorted numbers. Given $\mathrm{SA}[i] = j$, LCP[i] can then be derived from Hgt[j]. This method of storing the LCP in text order and not in SA order originated the concept of the *permuted longest-common-prefix array* (Kärkkäinen *et al.*, 2009) defined by $\mathrm{PLCP}[j] = \mathrm{LCP}[\mathrm{SA}^{-1}[j]]$. The major drawback of this approach is that it is dependent on the time needed to retrieve $\mathrm{SA}[i] = j$, which can be expensive when SA is not available explicitly.

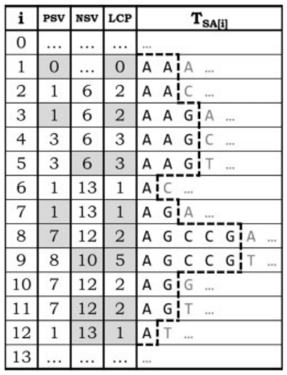
Another representation of the LCP array is the *sampled LCP* (SLCP) described in Sirén (2010). It introduces the notions of *maximal* or *irreducible* values, which satisfy $\mathrm{PLCP}[j] \neq (\mathrm{PLCP}[j-1] - 1)$, and *minimal* values, if either $j = (n-1)$ or $\mathrm{PLCP}[j+1]$ is maximal, i.e. if it is the last value of the array or the last one of a run of decreasing values flanked by a larger value on the right. The LCP is then sampled at these minimal PLCP values, which are as many as the number of equal letter runs in the BWT, and stored in SA order using a bit vector to mark their positions. To retrieve the value of LCP[i], the $\Psi(i)$ function is iterated k times until we fall over a sample, and finally it outputs $\mathrm{LCP}[i] = (\mathrm{LCP}[\Psi^k(i)] + k)$. The $\Psi(i)$ function is defined as $\Psi(i) = \mathrm{SA}^{-1}[\mathrm{SA}[i] + 1]$, meaning that it is the converse of the LF(i) function but returning the position in the SA of the suffix one text position to the right instead of to the left.

2 METHODS

2.1 SLCP array

The main motivation for our sampled version of the LCP is the observation that when performing the backward search over the BWT, if we want to retrieve the parent interval of the current BWT interval through the LCP, PSV and NSV arrays, we only need the values located at the two positions corresponding to both ends of the interval (more accurately, we need the top end and the position next to the bottom end) and not on any of the values in between. Therefore, it suffices to store the data only for positions that correspond to edges of non-singular BWT search intervals instead of storing it for all the positions of the BWT. This can be seen in Figure 4, where, for example, at the 'AG'-interval, given by 2−[7,11], we only need the values of the mentioned arrays at positions 7 and 12 = 11 + 1.

As mentioned before, to be able to retrieve LCP[i], previous LCP representations need to first perform a series of Ψ steps to compute the value of SA[i], and in virtually all implementations of compact indexes, the SA array is already stored in some sampled form. Unlike them, our approach does not require this time penalty cost for any supplemental computation of SA or LF values. Because we need to access the LCP array quite often for the purpose of resolving parent intervals, the retrieval of the lcp values should be as fast as possible. Therefore, the efforts in this work have been made in the direction of reducing the LCP space requirements with this SLCP approach, but without making use of any



i	PSV	NSV	LCP	$T_{SA[i]}$
0	...	...	...	...
1	0	...	0	A A A ...
2	1	6	2	A A C ...
3	1	6	2	A A G A ...
4	3	6	3	A A G C ...
5	3	6	3	A A G T ...
6	1	13	1	A C ...
7	1	13	1	A G A ...
8	7	12	2	A G C C G A ...
9	8	10	5	A G C C G T ...
10	7	12	2	A G G ...
11	7	12	2	A G T ...
12	1	13	1	A T ...
13	...	...	...	...

Fig. 4. Example of a hypothetical section of an index structure showing arrays LCP, PSV and NSV. The sampled values of these three arrays are emphasized in gray. Only top and bottom corner positions are sampled for the LCP. The PSV is only sampled at top corners and the NSV only at bottom corners

other space-oriented representations that would sacrifice speed, such as representing it with a wavelet tree (Kulekci *et al.*, 2012). Unlike the sampled version from Sirén (2010), our SLCP method takes the positions in the LCP that correspond to the boundaries of the search intervals in the BWT. More precisely, these positions of interest are the ones delimiting BWT ranges with the same depth, i.e. the positions whose lcp value differs from the next one: $\mathrm{SLCP} = \{\mathrm{LCP}[i] : i = (n-1) \text{ or } \mathrm{LCP}[i] \neq \mathrm{LCP}[i+1]\}$.

This is the same as discarding the positions with consecutive equal lcp values, similar to what is done in run length encoding, but without the need to store the length of each run. The rest of the positions, if needed, can be deduced from the sampled ones. We use the terms *top corner* and *bottom corner* to refer to the lower value/topmost position and to the higher value/bottommost position of the BWT search interval, respectively. Attending to this, every sampled position is either a top corner or a bottom corner. Formally, the sets of both types of corners are defined by:

- $\mathrm{TopCorners} = \{i : (i+1) \neq n \text{ and } \mathrm{LCP}[i] < \mathrm{LCP}[i+1]\}$
- $\mathrm{BottomCorners} = \{i : (i+1) = n \text{ or } \mathrm{LCP}[i] > \mathrm{LCP}[i+1]\}$

2.2 Sampled smaller values

While calculating parent intervals when executing a BWT search, we only need to perform PSV requests on the higher edge of each interval, i.e. on top corners, and NSV requests on the lower edge, i.e. on bottom corners. Therefore, it is sufficient to keep one single sampled smaller value (SSV) array instead of both PSV and NSV arrays. SSV[i] will automatically return the value of PSV[i] or NSV[i] if position i corresponds to a top corner or to a bottom corner, respectively. More correctly, SSV[i] of bottom corners stores (NSV[i + 1] − 1), as NSVs are always one position ahead of bottom corners. Hence, PSV and NSV values for sampled positions can be recovered from the SSV array using the relation:

- $\mathrm{SSV}[i'] = \mathrm{PSV}[i]$, if $\mathrm{SSV}[i'] < i$
- $\mathrm{SSV}[i'] = \mathrm{NSV}[i+1] - 1$, if $\mathrm{SSV}[i'] > i$

Where i' is the number of sampled positions in the interval $[0,(i-1)]$ because SSV does not have the same size as PSV/NSV, as it only stores the sampled positions.

Algorithm 1 associates each SLCP position with its corresponding PSV or NSV positions depending on whether it is a top or bottom corner, and stores it in the unified SSV array. It shares some resemblances with the procedure from (Abouelhoda *et al.*, 2004) to calculate the lcp-interval tree, as the PSV/NSV queries are the basis for the parent/child relationships in the tree's hierarchy. For simplicity, it uses the full LCP and SSV arrays with size *n*, but it is easy to adapt it to use the sampled versions instead.

Algorithm 1. GetSmallerValues( )

```
(01)  topCorners ← {(0, -1)};
(02)  bottomCorners ← { };
(03)  LCP[n] ← -1;
(04)  for i ← 1 ... n-1 do
(05)      if (LCP[i+1] > LCP[i]) then
(06)          SSV[i] ← topCorners.top().pos;
(07)          topCorners.push( i , LCP[i] );
(08)      else if (LCP[i+1] < LCP[i]) then
(09)          while ((not bottomCorners.empty()) and (bottomCorners.top().lcp > LCP[i+1]))
(10)              SSV[ bottomCorners.pop().pos ] ← i;
(11)          end
(12)          bottomCorners.push( i , LCP[i+1] );
(13)          while ((not topCorners.empty()) and (topCorners.top().lcp ≥ LCP[i+1]))
(14)              topCorners.pop();
(15)          end
(16)  end
```

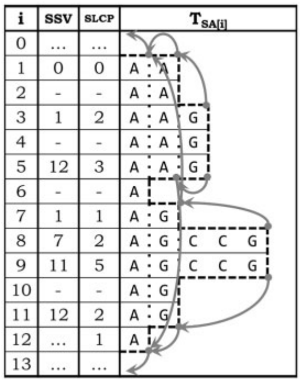
Algorithm 1 maintains two stacks, one for top corners (line 1) and another for bottom corners (line 2). Each stack stores pairs of values containing the position of the corner in the BWT and an lcp-value. This lcp-value is LCP[i] for top corners and LCP[i + 1] for bottom corners, as we are interested in how low the lcp-value was before and how low will it decrease next, respectively. This means the top corners stack stores {i,LCP[i]} and the bottom corners stack stores {i,LCP[i + 1]}. At every moment, the set of pairs stored in both stacks is always ordered by increasing positions and non-decreasing lcp-values. To fill the SSV array, the LCP is scanned from top to bottom. When a top corner is found (line 5), it is linked to the previous found top corner (line 6) and added to the stack (line 7) to be later linked to by another top corner. When a bottom corner is found (line 8), all the active previous bottom corners with an lcp-value higher than the current one (line 9) are linked to it and removed from the stack (line 10), as they were already used. This new bottom corner is saved (line 12) to be later linked to the next lower such corner. All the top corners with lcp-value higher or equal than the current bottom corner are also removed from its stack (lines 13 and 14), as these will not have a chance of being linked to anymore.

Algorithm 2 allows us to obtain the parent interval of a given interval by relying solely on the sampled arrays, SLCP and SSV.

Algorithm 2. GetParentInterval(*[i, j]*)

```
(01)  i' ← GetPositionInSampledArray( i );
(02)  j' ← GetPositionInSampledArray( j );
(03)  if (SLCP[i'] > SLCP[j'+1]) then
(04)      depth ← SLCP[i'];
(05)      topPos ← SSV[i'];
(06)      bottomPos ← j;
(07)  else if (SLCP[j'+1] > SLCP[i']) then
(08)      depth ← SLCP[j'+1];
(09)      topPos ← i;
(10)      bottomPos ← SSV[j'];
(11)  else
(12)      depth ← SLCP[i'];
(13)      topPos ← SSV[i'];
(14)      bottomPos ← SSV[j'];
(15)  return {depth, [topPos, bottomPos]}
```

For clarity reasons, the check to ensure that (j' + 1) does not go beyond the size of the SLCP was omitted from the pseudo-code above to facilitate its reading. First, the edge positions of our initial interval in the full arrays, *i* and *j*, are translated into positions in the sampled arrays, *i'* and *j'* (lines 1 and 2). At this point, we cannot simply set each corner to its SSV position because we also have to take into consideration the depth of the parent interval. Similarly to the *Parent* operation on lcp-intervals, we need to check which side of the interval will lead to a higher lcp-value (lines 3 and 7), because the lcp-value of the parent is always max{LCP[i'],LCP[j' + 1]}. The corner displaying this property is expanded and replaced by the corner at its SSV position and the other corner remains unchanged. If both corners share the same value (line 11), they are both updated. Finally, it outputs the parent interval in the BWT along with its destination depth (line 15).



i	SSV	SLCP	$T_{SA[i]}$
0	...	...	
1	0	0	A A
2	-	-	A A
3	1	2	A A G
4	-	-	A A G
5	12	3	A A G
6	-	-	A
7	1	1	A G
8	7	2	A G C C G
9	11	5	A G C C G
10	-	-	A G
11	12	2	A G
12	...	1	A
13	...	...	

Fig. 5. Parent interval relations in a section of a hypothetical index. The arrows connect the top/bottom corners of each interval to their respective parent interval's top/bottom corners at the appropriate destination depth. In the SLCP and SSV sampled arrays, dashes represent un-sampled positions

In Figure 5, the illustrated arrows are connecting the top/bottom corners of the child intervals with the top/bottom corners of their parent intervals at their corresponding depths. Note that because intervals with different depths can share one of their borders, these arrows do not necessarily correspond to the destination SSV positions. For example, in the succession of parent intervals 3−[3,5], 2−[1,5] and 1−[1,12], the first two share the same bottom corner at position 5, so SSV[5] points directly to position 12, the bottom corner of the last one.

When we have an interval composed of a single position, we find its parent interval by doing a simple scan in the SLCP for the closest top and bottom corners around that position. Because we use the SLCP instead of the LCP, the search is faster. The boundaries of our target interval are promptly determined by starting at those initial corners and iteratively following the SSV values until the required destination depth is reached.

Because in the current context only parent-interval queries are required and we have no need for child-interval queries, this new SSV array replaces the lcp-interval tree and it also eliminates the need for other representations of the ST topology such as balanced parenthesis (Geary *et al.*, 2006) or RMQs (Fischer and Heun, 2007). We give the term Sampled Search Intervals from Longest Common Prefixes (SSILCP) to the structure combining the described SLCP and SV arrays.

Each interval with no children intervals is of size no larger than |Σ|, as there are only |Σ| distinct characters (including the terminator character '$') that can be used to extend the common prefix in each suffix, otherwise if at least one of the characters was repeated, it would create a child interval. This means that in each childless interval, we will have at most (|Σ|−2) un-sampled positions, because we always need one sample for each edge. Therefore, theoretically, the maximum number of positions saved by using the SLCP instead of the full LCP will be of 3*(n/5) for DNA alphabet, which means, at most, a 60% size reduction, although in practice that value will be lower.

2.3 Finding MEMs

The MEM searching algorithm is based on the one proposed in Section 4 of (Ohlebusch *et al.*, 2010) and used in backwardMEM, as the underlying index structure of both works is supported by backward matching.

Algorithm 3. GetMEMs(*P* , *minLength*)

```
(01)  (l, [i, j]) ← (0, [0, n-1]);
(02)  for k ← m-1 ... 0 do
(03)      [nexti, nextj] ← BackwardSearchStep( P[k] , [i, j] )
(04)      while ([nexti, nextj] = [ ]) do
(05)          l-[i, j] ← Parent( l-[i, j] );
(06)          [nexti, nextj] ← BackwardSearchStep( P[k] , [i, j] );
(07)      end
(08)      (l, [i, j]) ← (l+1, [nexti, nextj]);
(09)      (l', [i', j']) ← (l, [i, j]);
(10)      (previ', prevj') ← (j', j'+1);
(11)      while (l' ≥ minLength) do
(12)          while (previ' ≥ i') do
(13)              if ((k = 0) or (BWT[previ'] ≠ P[k-1])) output (l', SA[previ'], k);
(14)              previ' ← previ'-1;
(15)          end
(16)          while (prevj' ≤ j') do
(17)              if ((k = 0) or (BWT[prevj'] ≠ P[k-1])) output (l', SA[prevj'], k);
(18)              prevj' ← prevj'+1;
(19)          end
(20)          l'-[i', j'] ← Parent( l'-[i', j'] );
(21)      end
(22)  end
```

Basically, Algorithm 3 processes the query sequence backward (line 2), and for each interval [i,j] found (line 3), it keeps following parent intervals [i′,j′] (line 20) until the length l' of the current match is lower than a given threshold *minLength* (line 11). Because the right-maximality is already assured (lines 4–7), we check for left-maximality by verifying that each position inside [i′,j′] can no longer be further extended to the left, i.e. if the character to the left in the text, given in the BWT array, is not the same as the character to the left in the pattern (lines 13 and 17). Because each new parent interval [i′,j′] encloses the previous one *[previ′,prevj′]*, only the newly found positions above (lines 12–15) and below (lines 16–19) are checked. This algorithm runs in $O(m + R_L + M_L{*}t_{SA})$ time, where m is length of the query sequence, R_L and M_L are the number of right maximal matches and MEMs, respectively, of size at least $L = minLength$ and t_{SA} is the time needed to obtain a value from the SA, which is constant in our case.

2.4 Implementation

The SSILCP is built from the full LCP, presenting a variable sampling rate of ~1.2 based on the test results of Table 1, and is used as a replacement for both the LCP and the PSV/NSV arrays. Following the same idea as in (Abouelhoda *et al.*, 2004), based on the observation that the vast majority of the values in the LCP array are small, the lcp values <255 are stored using 8 bits per number, while the remaining values go into a complementary table containing only larger numbers. We also use an additional space-saving trick based on the fact that most of the PSV[i] and NSV[i] values do not jump too far away from its position i. Therefore, the SSV array can also take advantage of a similar approach as the LCP array by storing the differential values between the source and destination positions, with negative values representing PSV jumps and positive values NSV jumps. Now the values represented using only 8 bits are within the range from −127 to +127.

To comply with the most commonly used computer memory architectures, the SA array coupled to the FM-Index uses a fixed sampling rate of 32. Further details about the index and SSILCP implementations are available in the Supplementary Material. All algorithms and data structures have been developed from scratch without relying on any other existing code base.

Table 1. LCP statistics for the used datasets

Dataset	Drosophila	Human versus mouse
Genome reference size	162 Mbp	2897 Mbp
Sampled positions	88.3%	86.9%
Oversized LCP samples	8.2%	1.6%
Oversized SV samples	2.6%	2.8%
Average lcp value	100	1059
Maximum lcp value	48 382	2 339 520
SSILCP structure size	416 MB	6680 MB
Index size	182 MB	3259 MB

Note: The reference genome size considers {A,C,G,T} chars only. Samples with an lcp value >254 and SV samples with an absolute value >127 are considered oversized. The maximum lcp value indicates the length of the largest repeat present in the genome. The full SSILCP size accounts for the SLCP and SSV arrays and all the supporting data structures, excluding the FM-Index. MB, Megabyte; Mbp, Mega base pair.

3 RESULTS

3.1 Datasets

As test suites, we have chosen two real-life scenarios that feature significantly sized genomes. The first dataset is constituted by two species of the fruit fly, *Drosophila melanogaster* and *Drosophila yakuba*, with 162 and 163 Mbp, respectively, as this setting was also featured in the publications of every other tested tool. The second dataset includes the complete genomes of *Homo sapiens*, build 19 (HG19) (Church *et al.*, 2011), and *Mus musculus*, build 10 (MM10) (Waterston *et al.*, 2002), with 3.1 and 2.7 Gbp, respectively. The chosen references for each dataset were *D. melanogaster* and HG19. Specific LCP-related characteristics for each one are depicted in Table 1.

Using this new SSILCP data structure, we get $O(1)$ time for LCP and parent operations. It also scales linearly with the size of the reference genome. The space requirements are typically around 2.2*n bytes for practical applications on DNA according to the tests presented in the second last row of Table 1, consuming ~19% of the space we would have used by storing the full LCP, PSV and NSV arrays in the naïve way.

3.2 Tested programs

MUMmer builds an ST for the reference using the compact representation from (Kurtz, 1999) that requires ~15.4 bytes per input character and streams the query sequences against it. sparseMEM indexes the reference with a sparse SA and uses the LCP and SA^{-1} arrays to simulate suffix links, which are essential to accelerate the computation of MEMs in that data structure. Its time complexity is O(m*log(n) + q) for a reference of size *n*, query of size *m* and *q* dependent on the sparseness factor and minimum matches length. A *sampled* SA approach keeps all the suffixes of the text but only stores each *k*th entry of the SA array, whereas the *sparse* SA approach only maintains each *K*th suffix of the text and their corresponding SA entry. essaMEM works over an enhanced sparse SA that replaces the SA^{-1} array of sparseMEM with a sparse child array, greatly reducing the time complexity by removing the log(n) term while maintaining a similar memory usage of (9/K + 1)*n bytes. backwardMEM uses an enhanced compressed SA supported by a BWT encoded as a wavelet-tree and adapted existing MEM locating algorithms to work with backward search. It features a balanced parentheses representation of the lcp-interval tree capable of constant time parent interval queries. Its memory requirements are ~(4/k + 2)*n bytes in practice.

When available, the tools were given different SA sampling values of powers of 2 ranging from 1 to 32, using the run-time parameter '*-k*' in sparseMEM and essaMEM and the compile-time flag '*BWTK*' in backwardMEM. All tools were run with the same parameters '*-maxmatch -n -l L*' to report all MEMs, with minimum length $L = 50$ for the *Drosophila* dataset and $L = 100$ for the human/mouse dataset.

The source code of each tool was edited to launch a process in the background that starts collecting the time and memory values right after the data structures were built and just before the actual MEM finding algorithms take place. Because each tool uses different index structures and different construction algorithms that would be difficult to compare as a whole, this allows the benchmarks to reflect the MEM retrieving efficiency only.

3.3 Benchmarks

Time corresponds to the elapsed real time and memory to the maximum physically resident memory, fields 'etime' and 'rss', respectively, in the Unix system command 'ps', and measured using the 'memusgpid' script included in the source package. All tests were run on a Linux server machine featuring an Intel Xeon CPU clocked at 2.13 GHz with 256 GB of RAM and 64 cores, but none of the tools was run with multi-threading options. The results are presented in Table 2 and Figures 6 and 7 and detailed in the Supplementary Material.

Because all the tools only index the reference genome, the used memory is determined by the reference size, with the addition of the currently loaded query. As the results show, slaMEM's approach consumes ~3.3*n bytes in practice. In the *Drosophila* dataset, it uses approximately the same memory as backwardMEM and sparseMEM, both with a sampling value of 32, while being almost 7 times faster than backwardMEM and 25 times faster than sparseMEM. Therefore, between the two backward searching-based methods, slaMEM achieves the best performance. It is only outperformed in terms of memory by essaMEM with K = 16 and K = 32, while still running slightly faster. Compared with the un-sampled approach used in MUMmer, slaMEM runs in half the time and uses almost four times less space. In the human/mouse dataset, MUMmer, backwardMEM and essaMEM with K = 1 all failed or crashed possibly due to the use of signed integer variable types, which do not support arrays with sizes larger than 2 billion positions. Theoretically, for a comparable memory usage, slaMEM is equivalent to a sampling rate of K = 6 in the sparse methods and still almost 8 times faster than the closest test results (K = 4 and K = 8). The best time/memory ratio belongs to essaMEM. Nevertheless, slaMEM achieves the fastest running times among all the tested tools and sampling values in both datasets.

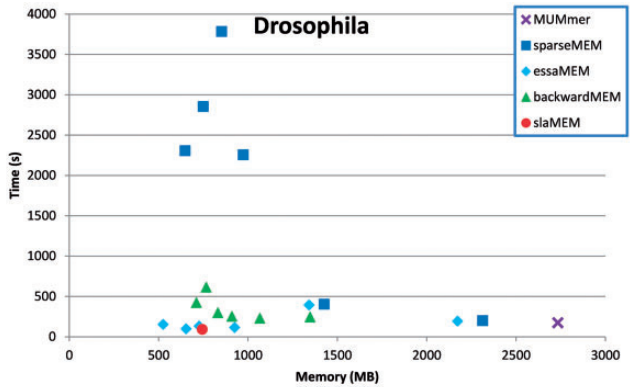


Fig. 6. Plot comparing the time and memory used by the different MEM locating tools in the *Drosophila* dataset. Multiple points for the same tool represent distinct values of K, if available

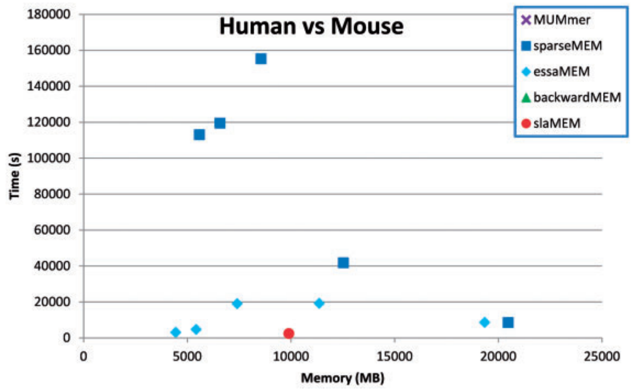


Fig. 7. Plot displaying the results of all the tested tools over the human dataset

Table 2. MEM statistics for the used datasets showing the number of found MEMs and their average size in each dataset

Dataset	Drosophila	Human versus mouse
Number of MEMs	1 461 805	537 438
Average MEM size	82	114

Note: Only MEMs with size at least 50 and 100, respectively, have been considered.

4 CONCLUSIONS

An algorithmic improvement that can be further explored is based on the observation that on applications that involve pursuing parent intervals only when a mismatch occurs, e.g. matching statistics (Chang and Lawler, 1994) or super-maximal matches (Gusfield, 1997), we only need to retrieve parent intervals when the current BWT search interval does not include at least one of the letters of the alphabet, which might be the one we were interested in following backward. This way, the SSILCP memory requirements can be lowered even further by ignoring the samples for LCP intervals that contain all the DNA letters. slaMEM could also take advantage of multi-threading to speed up computation by processing many queries or parts of the same query in parallel. Furthermore, by extending the current implementation to support some missing ST operations, the combination of the FM-Index with the SSILCP could be used as a full compressed ST representation.

We observed that because the boundaries of non-unitary BWT search intervals can only fall in certain positions, it is enough to sample the LCP, PSV and NSV arrays at those specific positions. Therefore, we engineered the SSILCP specifically to closely wrap around the string matching mechanism that characterizes the FM-Index. This added ability to the index enables faster parent interval queries, which makes it especially useful for calculating matching statistics and maximal exact matches, where its absence would otherwise render the MEM retrieval algorithm impractically slow. Unlike other representations of the LCP, ours does not depend on the calculation of any previous SA or LF values, resulting in a strategy with a good space/time tradeoff to replace both the LCP array and the lcp-interval tree. Results on real data show that, for this application, our new combined SLCP and PSV/NSV representation proves to be a competitive approach against other equivalent structures such as the lcp-interval tree, thus making slaMEM a useful backbone for any project in the field of comparative genomics that relies on MEMs.

ACKNOWLEDGEMENTS

The authors thank Arlindo Oliveira for proofreading and providing valuable suggestions for this document and Michaël Vyverman for his help on tracking down a bug in our sequence parsing code.

Funding: This work was supported by National Funds from FCT—Fundação para a Ciência e a Tecnologia, through projects PEst-OE/EEI/LA0021/2013 and PTDC/EIA-EIA/112283/2009. F.F. was supported by grant [SFRH/BD/45586/2008].

Conflict of Interest: none declared.

REFERENCES

Abouelhoda,M.I. *et al.* (2004) Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, **2**, 53–86.

Abouelhoda,M.I. *et al.* (2008) CoCoNUT: an efficient system for the comparison and analysis of genomes. *BMC Bioinformatics*, **9**, 476.

Altschul,S.F. *et al.* (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Burrows,M. and Wheeler,D.J. (1994) *A Block-Sorting Lossless Data Compression Algorithm*. Digital Systems Research Center, Palo Alto, CA.

Chang,W.I. and Lawler,E.L. (1994) Sublinear approximate string matching and biological applications. *Algorithmica*, **12**, 327–344.

Church,D.M. *et al.* (2011) Modernizing reference genome assemblies. *PLoS Biol.*, **9**, e1001091.

Delcher,A.L. *et al.* (1999) Alignment of whole genomes. *Nucleic Acids Res.*, **27**, 2369–2376.

Delcher,A.L. *et al.* (2002) Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.*, **30**, 2478–2483.

Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science, 2000*. pp. 390–398.

Ferragina,P. and Manzini,G. (2005) Indexing compressed text. *J. ACM*, **52**, 552–581.

Fischer,J. *et al.* (2009) Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, **410**, 5354–5364.

Fischer,J. and Heun,V. (2007) A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen,B. *et al.* (eds) *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, Berlin, pp. 459–470.

Geary,R.F. *et al.* (2006) A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, **368**, 231–246.

Gusfield,D. (1997) *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, MA.

Kärkkäinen,J. *et al.* (2009) Permuted longest-common-prefix array. In: *Combinatorial Pattern Matching*. Springer, Berlin, pp. 181–192.

Kärkkäinen,J. and Sanders,P. (2003) Simple linear work suffix array construction. In: Baeten,J. *et al.* (eds) *Automata, Languages and Programming*. Springer, Berlin, pp. 943–955.

Kasai,T. *et al.* (2001) Linear-time longest-common-prefix computation in suffix arrays and its applications. In: *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*. pp. 181–192.

Khan,Z. *et al.* (2009) A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, **25**, 1609–1616.

Kim,D.K. *et al.* (2003) Linear-time construction of suffix arrays. In: *Combinatorial Pattern Matching*. pp. 186–199.

Ko,P. and Aluru,S. (2003) Space efficient linear time construction of suffix arrays. In: Baeza-Yates,R. *et al.* (eds) *Combinatorial Pattern Matching*. Springer, Berlin, pp. 200–210.

Kulekci,M.O. *et al.* (2012) Efficient maximal repeat finding using the Burrows-Wheeler transform and wavelet tree. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, **9**, 421–429.

Kurtz,S. (1999) Reducing the space requirement of suffix trees. *Softw. Pract. Exp.*, **29**, 1149–1171.

Kurtz,S. *et al.* (2004) Versatile and open software for comparing large genomes. *Genome Biol.*, **5**, R12.

Manber,U. and Myers,G. (1993) Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, **22**, 935–948.

Navarro,G. and Mäkinen,V. (2007) Compressed full-text indexes. *ACM Comput. Surv.*, **39**, 2.

Nong,G. *et al.* (2009) Linear suffix array construction by almost pure induced-sorting. In: *Data Compression Conference, 2009. DCC'09*. pp. 193–202.

Ohlebusch,E. *et al.* (2010) Computing matching statistics and maximal exact matches on compressed full-text indexes. In: Chavez,E. and Lonardi,S. (eds) *String Processing and Information Retrieval*. Springer, Berlin, pp. 347–358.

Okanohara,D. and Sadakane,K. (2009) A linear-time burrows-wheeler transform using induced sorting. In: *String Processing and Information Retrieval*. pp. 90–101.

Sadakane,K. (2007) Compressed suffix trees with full functionality. *Theory Comput. Syst.*, **41**, 589–607.

Sirén,J. (2010) Sampled longest common prefix array. In: *Combinatorial Pattern Matching*. pp. 227–237.

Ukkonen,E. (1995) On-line construction of suffix trees. *Algorithmica*, **14**, 249–260.

Vyverman,M. *et al.* (2012) Prospects and limitations of full-text index structures in genome analysis. *Nucleic Acids Res.*, **40**, 6993–7015.

Vyverman,M. *et al.* (2013) essaMEM: finding Maximal Exact Matches using enhanced sparse suffix arrays. *Bioinformatics*, **29**, 802–804.

Waterston,R.H. *et al.* (2002) Initial sequencing and comparative analysis of the mouse genome. *Nature*, **420**, 520–562.

Weiner,P. (1973) Linear pattern matching algorithms. In: *IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory, 1973. SWAT'08*. pp. 1–11.