Computing Matching Statistics and Maximal Exact Matches on Compressed Full-Text Indexes

Enno Ohlebusch, Simon Gog, and Adrian Kügel

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm
{Enno.Ohlebusch,Simon.Gog,Adrian.Kuegel}@uni-ulm.de

Abstract. Exact string matching is a problem that computer programmers face on a regular basis, and full-text indexes like the suffix tree or the suffix array provide fast string search over large texts. In the last decade, research on compressed indexes has flourished because the main problem in large-scale applications is the space consumption of the index. Nowadays, the most successful compressed indexes are able to obtain almost optimal space and search time simultaneously. It is known that a myriad of sequence analysis and comparison problems can be solved efficiently with established data structures like the suffix tree or the suffix array, but algorithms on compressed indexes that solve these problem are still lacking at present. Here, we show that *matching statistics* and *maximal exact matches* between two strings S^1 and S^2 can be computed efficiently by matching S^2 backwards against a compressed index of S^1.

1 Introduction

The suffix tree of a string S of length n is an index structure that can be computed and stored in $O(n)$ time and space; see the seminal paper of Weiner [1]. Once constructed, it can be used to efficiently solve a myriad of string processing problems [2,3]. Although being asymptotically linear, the space consumption of a suffix tree is quite large (about $20n$ bytes). This is a drawback in actual implementations. Thus, nowadays many string algorithms are based on suffix arrays instead. The suffix array specifies the lexicographic ordering of all suffixes of S, and it was introduced by Manber and Myers [4]. Almost a decade ago, Ferragina and Manzini [5] invented the FM-index, which is based on the Burrows-Wheeler transform [6] and the LF-mapping, and showed that it is possible to search a pattern backwards in the suffix array SA of string S using the FM-index instead of SA. In contrast to traditional data structures like the suffix tree, this compressed index supports backward search much better than forward search. Needless to say that one needs new algorithms to exploit this. In this paper, we present the first algorithms for computing matching statistics and maximal exact matches that use backward search instead of forward search. Matching statistics were introduced by Chang and Lawler [7] to solve the approximate string matching problem. Among other things, they were used in the computation of string kernels [8] and the design of DNA chips [9]. Matching statistics can also be used in a space-efficient computation of longest common substrings [3, Section 7.9]

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 347–358, 2010.

i	SA	LCP	PSV	NSV	BWT	$S_{\mathsf{SA}[i]}$
1	11	-1			t	\$
2	3	0	1	12	c	$aaacatat$\$
3	4	2	2	4	a	$aacatat$\$
4	1	1	2	8	\$	$acaaacatat$\$
5	5	3	4	6	a	$acatat$\$
6	9	1	2	8	t	at\$
7	7	2	6	8	c	$atat$\$
8	2	0	1	12	a	$caaacatat$\$
9	6	2	8	10	a	$catat$\$
10	10	0	1	12	a	t\$
11	8	1	10	12	a	tat\$
12		-1				

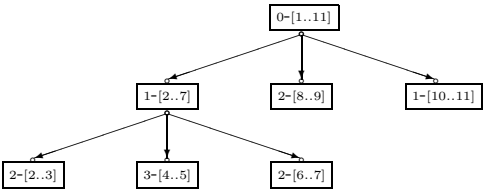

Fig. 1. Left: The suffix array of the string $S = acaaacatat\$$ and auxiliary arrays. Right: The lcp-interval tree (without singleton intervals) of the lcp-array.

and in the preprocessing phase of an algorithm that determines longest common prefix queries in constant time [3, Section 9.1]. Maximal exact matches play a key role in genome-genome comparisons (see e.g. [10,11]) and recently they were used to seed alignments of high-throughput reads for genome assembly. Our experiments show that the new algorithms outperform the current state-of-the-art algorithms based on forward search.

2 Preliminaries

Let Σ be an ordered alphabet whose smallest element is the so-called sentinel character \$. In the following, S is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the ith suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$; see Fig. 1 for an example. In the following, SA^{-1} denotes the inverse of the permutation SA. In 2003, it was shown independently and contemporaneously by three research groups that a direct linear time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms are known; see [12].

The Burrows and Wheeler transform converts a string S into the string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 1. In virtually all cases, the Burrows-Wheeler transformed string compresses much better than the original string; see [6]. The permutation LF, defined by $LF(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$

Algorithm 1. Given $c \in \Sigma$ and an ω-interval $[i..j]$, $backwardSearch(c, [i..j])$ returns the $c\omega$-interval if it exists, and $\perp$ otherwise

```
backwardSearch(c, [i..j])
  i ← C[c] + Occ(c, i − 1) + 1
  j ← C[c] + Occ(c, j)
  if i ≤ j then return [i..j]
  else return ⊥
```

and $LF(i) = 1$ otherwise, is called LF-mapping. The LF-mapping can be implemented by

$$LF(i) = C[c] + Occ(c, i), \text{ where } c = \mathsf{BWT}[i],$$

$C[c]$ is the overall number (of occurrences) of characters in S which are strictly smaller than c, and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$.

In the following, the ω-interval of a substring ω of S is the interval $[i..j]$ in the suffix array SA such that ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \leq k \leq j$, but ω is not a prefix of any other suffix of S. For example, the ca-interval in the suffix array of Fig. 1 is the interval [8..9]. Ferragina and Manzini [5] showed that it is possible to search a pattern character-by-character backwards in the suffix array SA of string S, without storing SA; see Algorithm 1.

Searching backwards in the string $S = acaaacatat\$$ for the pattern ca works as follows. By definition, backward search for the last character of the pattern starts with the ε-interval $[1..n]$, where ε denotes the empty string. In our example $n = 11$ and $backwardSearch(a, [1..11])$ returns the a-interval [2..7] because $C[a] + Occ(a, 1 - 1) + 1 = 1 + 0 + 1 = 2$ and $C[a] + Occ(a, 11) = 1 + 6 = 7$. Similarly, $backwardSearch(c, [2..7])$ delivers the ca-interval [8..9] because $C[c] + Occ(c, 2 - 1) + 1 = 7 + 0 + 1 = 8$ and $C[c] + Occ(c, 7) = 7 + 2 = 9$.

A space efficient data structure that supports backward search and the LF-mapping (plus a certain navigational operation in the lcp-interval tree as detailed below) will be called *compressed full-text index* in this paper. In our implementation, we use the wavelet tree of Grossi et al. [13] but there are alternatives; see the review paper of Navarro and Mäkinen [14] for details. With the wavelet tree, both a backward search step and the computation of $LF(i)$ take constant time; see [13].[1] The wavelet tree uses $n \log |\Sigma| + o(n \log |\Sigma|)$ bits of space.

The suffix array SA is often enhanced with the so-called lcp-array LCP containing the lengths of longest common prefixes between consecutive suffixes in SA; see Fig. 1. Formally, the lcp-array is an array such that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n + 1]$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leq i \leq n$, where $\mathsf{lcp}(u, v)$ denotes the longest common prefix between two strings u and v. Kasai et al. [15] showed that the lcp-array can be computed in linear time from the suffix array and

[1] Strictly speaking, it takes $\mathcal{O}(\log |\Sigma|)$ time, but here we assume a constant alphabet. In fact it is possible to get rid of the $\log |\Sigma|$ factor, trading space for time; see [14].

its inverse. Sadakane [16] describes an encoding of the lcp-array that uses only $2n + o(n)$ bits. Abouelhoda et al. [17] introduced the concept of lcp-intervals. An interval $[i..j]$, where $1 \leq i < j \leq n$, in an lcp-array LCP is called an *lcp-interval of lcp-value* ℓ (denoted by ℓ-$[i..j]$) if

1. $\mathsf{LCP}[i] < \ell$,
2. $\mathsf{LCP}[k] \geq \ell$ for all k with $i + 1 \leq k \leq j$,
3. $\mathsf{LCP}[k] = \ell$ for at least one k with $i + 1 \leq k \leq j$,
4. $\mathsf{LCP}[j + 1] < \ell$.

An lcp-interval m-$[p..q]$ is said to be *embedded* in an lcp-interval ℓ-$[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leq p < q \leq j$) and $m > \ell$. The interval $[i..j]$ is then called the interval *enclosing* $[p..q]$. If $[i..j]$ encloses $[p..q]$ and there is no interval embedded in $[i..j]$ that also encloses $[p..q]$, then $[p..q]$ is called a *child interval* of $[i..j]$. This parent-child relationship constitutes a tree which we call the *lcp-interval tree* (without singleton intervals); see Fig. 1.

An interval $[k..k]$ is called *singleton interval*. The parent interval of such a singleton interval is the smallest lcp-interval $[i..j]$ which contains k. The parent interval of an lcp-interval $[i..j] \neq [1..n]$ with $\mathsf{LCP}[i] = p$ and $\mathsf{LCP}[j + 1] = q$ can be determined as

$$parent([i..j]) = \begin{cases} p\text{-}[\mathsf{PSV}[i]..\mathsf{NSV}[i] - 1] & , \text{if } p \geq q \\ q\text{-}[\mathsf{PSV}[j + 1]..\mathsf{NSV}[j + 1] - 1], & \text{if } p < q \end{cases}$$

where, for any index $2 \leq i \leq n$,

$$\mathsf{PSV}[i] = \max\{k \mid 1 \leq k < i \text{ and } \mathsf{LCP}[k] < \mathsf{LCP}[i]\}$$
$$\mathsf{NSV}[i] = \min\{k \mid i < k \leq n + 1 \text{ and } \mathsf{LCP}[k] < \mathsf{LCP}[i]\}$$

3 Matching Statistics by Backward Search

In the following, let S^1 and S^2 be strings of length n_1 and n_2, respectively. We tacitly assume that S^1 has the sentinel character at the end (and nowhere else). As already mentioned, Chang and Lawler [7] introduced matching statistics in the context of approximate string matching. For each position p_2 in the string S^2, they searched for the longest match of $S^2[p_2..n_2]$ with a substring of S^1 by matching S^2 in *forward* direction against the *suffix tree* of S^1. This takes only linear time if suffix links are used as shortcuts in the traversal of the suffix tree; cf. [3, Section 7.8]. By contrast, we here match S^2 in *backward* direction against a compressed index of S^1. Our algorithm does not rely on suffix links but on the ability to determine parent intervals of lcp-intervals efficiently. Sadakane's [16] compressed suffix tree requires $4n + o(n)$ bits and allows one to determine a parent interval in constant time. Recently, we presented a balanced parentheses data structure that can be constructed in linear time from the lcp-array, uses only $2n + o(n)$ bits, and—in combination with the lcp-array—allows us to compute $\mathsf{NSV}[i]$ in constant time and $\mathsf{PSV}[i]$ in $\mathcal{O}(\log |\Sigma|)$ time [18]. In other words, a parent interval can be computed in constant time (for a constant alphabet).

Algorithm 2. Computing matching statistics by backward search

```
p_2 ← n_2
(q, [i..j]) ← (0, [1..n_1])
while p_2 ≥ 1 do
  [lb..rb] ← backwardSearch(S^2[p_2], [i..j])
  if [lb..rb] ≠ ⊥ then
    q ← q + 1
    ms[p_2] ← (q, [lb..rb])
    [i..j] ← [lb..rb]
    p_2 ← p_2 − 1
  else if [i..j] = [1..n_1] then
    ms[p_2] ← (0, [1..n_1])
    p_2 ← p_2 − 1
  else
    q-[i..j] ← parent([i..j])
```

An even more space-efficient implementation was proposed by Fischer et al. [19]. Their method to identify parent intervals also uses PSV and NSV values and runs in sublogarithmic time (in n). Either of the three methods can be used in our algorithms.

Before explaining our new Algorithm 2, we define matching statistics slightly more general than Chang and Lawler did.

Definition 1. *A* matching statistics *of S^2 w.r.t. S^1 is an array ms such that for every entry $ms[p_2] = (q, [lb..rb])$, $1 \leq p_2 \leq n_2$, the following holds:*

1. *$S^2[p_2..p_2 + q - 1]$ is the longest prefix of $S^2[p_2..n_2]$ which is substring of S^1.*
2. *$[lb..rb]$ is the $S^2[p_2..p_2 + q - 1]$-interval in the suffix array of S^1.*

Algorithm 2 shows how matching statistics can be computed by backward search. To exemplify it, we match the string $S^2 = caaca$ backwards against the compressed full-text index of $S^1 = acaaacatat\$$; cf. Fig. 1. Starting with the last character of S^2 and the ε-interval $[1..n_1]$, backward search returns the a-interval [2..7], and Algorithm 2 sets $ms[5] = (1, [2..7])$. Similary, it determines $ms[4] = (2, [8..9])$, $ms[3] = (3, [4..5])$, and $ms[2] = (4, [3..3])$. The procedure call $backwardSearch(c, [3..3])$ returns $\perp$, indicating that $S^2[1..5] = caaca$ is not a substring of S^1. In this case—if a mismatch occurs—the algorithm determines the parent interval of the current interval [3..3], which in our example is the aa-interval 2-[2..3]. The next procedure call $backwardSearch(c, [2..3])$ returns the caa-interval [8..8], and $ms[1]$ is set to $(3, [8..8])$.

We prove the correctness of Algorithm 2 by finite induction on the length $n_2 - p_2 + 1$ of the suffix $S^2[p_2..n_2]$ of S^2. If the length equals 1, i.e., $p_2 = n_2$ then there are two possibilities. The character $c = S^2[n_2]$ either (a) occurs in S^1 or (b) it does not. In case (a), Algorithm 2 sets $ms[n_2] = (1, [lb..rb])$, where $[lb..rb]$ is the c-interval. This is certainly correct. In case (b), Algorithm 2 sets $ms[n_2] = (0, [1..n_1])$, where

$[1..n_1]$ is the ε-interval. This is also correct. As induction hypothesis, we may assume that for some fixed position $p_2 + 1$ with $1 \leq p_2 < n_2$, Algorithm 2 correctly computed the matching statistic $ms[p_2 + 1] = (q, [i..j])$, i.e.,

1. $\omega = S^2[p_2 + 1..p_2 + q]$ is the longest prefix of $S^2[p_2 + 1..n_2]$ that occurs as a substring of S^1.
2. $[i..j]$ is the ω-interval in the suffix array of S^1.

In the inductive step, we must show that Algorithm 2 correctly computes $ms[p_2]$. Let $c = S^2[p_2]$. If $c\omega$ is a substring of S^1, then $backwardSearch(c, [i..j])$ yields the $c\omega$-interval $[lb..rb]$ in the suffix array of S^1. It is readily verified that $c\omega = S^2[p_2..p_2 + q]$ is the longest prefix of $S^2[p_2..n_2]$ that occurs as a substring of S^1. Consequently, $ms[p_2] = (q + 1, [lb..rb])$. Otherwise, if $c\omega$ is not a substring of S^1, then $backwardSearch(c, [i..j])$ returns $\bot$. We consider the two subcases (a) $[i..j] = [1..n_1]$ and (b) $[i..j] \neq [1..n_1]$.
(a) If $[i..j] = [1..n_1]$, i.e., $\omega = \varepsilon$, then the character c does not occur in S^1. This means that the longest prefix of $S^2[p_2..n_2]$ that occurs as a substring of S^1 is the empty string ε and $ms[p_2] = (0, [1..n_1])$.
(b) If $[i..j] \neq [1..n_1]$, then $\omega \neq \varepsilon$. Because $c\omega$ is not a substring of S^1, we must search for the longest prefix u' of ω such that cu' is a substring of S^1. Let $[i'..j']$ be the parent lcp-interval of $[i..j]$. The lcp-interval $[i'..j']$ is the u-interval of a proper prefix u of ω. Suppose that b is the character immediately following u in ω, i.e., $\omega = ubv$ for some string v. Because the u-interval $[i'..j']$ is the parent lcp-interval of the ω-interval $[i..j]$, every substring ω' of S^1 that has ub as a prefix must also have ω as a prefix. We claim that the string cub cannot occur in S^1. To prove the claim, suppose to the contrary that cub is a substring of S^1. Because every substring ω' of S^1 that has ub as a prefix must also have ω as a prefix, it follows that $c\omega$ must be a substring of S^1. This contradicts the fact that $c\omega$ is not a substring of S^1 and thus proves the claim that the string cub cannot occur in S^1. Consequently, u is the longest prefix of ω such that cu is a possible substring of S^1. Observe that the algorithm checks in the next iteration of the while-loop whether or not cu is indeed a substring of S^1. If so, then u is the longest prefix of ω such that cu is a substring of S^1. If not, the algorithm continues with the parent interval of the u-interval $[i'..j']$, and so on, until either backward search succeeds or the interval $[1..n_1]$ is found. In both cases $ms[p_2]$ is assigned correctly.

We use an amortized analysis to derive the worst-case time complexity of Algorithm 2. Each statement in the while-loop takes only constant time (assuming a constant alphabet). We claim that the number of iterations of the while-loop over the entire algorithm is bounded by $2n_2$. In each iteration of the while-loop, either the position p_2 in S^2 is decreased by one or the search interval $[i..j]$ is replaced with its parent interval. Clearly, p_2 is decreased n_2 times and we claim that at most n_2 many search intervals can be replaced with its parent interval. To see this, let the search interval $[i..j]$ be the ω-interval and let $[i'..j']$ denote its parent interval. The lcp-interval $[i'..j']$ is the u-interval of a proper prefix u of ω. Consequently, each time a search interval is replaced with its parent interval, the length of the search string ω is shortened by at least one. Since the overall

length increase of all search strings is bounded by n_2, the claim follows. Thus, in our implementation, Algorithm 2 has a worst-case time complexity of $O(n_2)$.

4 Computing Maximal Exact Matches by Backward Search

The starting point for any comparison of large genomes is the computation of exact matches between their DNA sequences S^1 and S^2. In our opinion, maximal exact matches—exact matches that cannot be extended in either direction towards the beginning or end of S^1 and S^2 without allowing for a mismatch—are most suitable for this task.

Definition 2. *An exact match between two strings S^1 (where S^1 ends with \$) and S^2 of lengths n_1 and n_2 is a triple (q, p_1, p_2) such that $S^1[p_1..p_1 + q - 1] = S^2[p_2..p_2 + q - 1]$. An exact match is called right maximal if $p_2 + q - 1 = n_2$ or $S^1[p_1+q] \neq S^2[p_2+q]$. It is called left maximal if $p_2 = 1$ or $\mathsf{BWT}[p_1] \neq S^2[p_2-1]$. A left and right maximal exact match is called maximal exact match (MEM).*

In genome comparisons, one is merely interested in MEMs (q, p_1, p_2) that exceed a user-defined length threshold ℓ, i.e., $q \geq \ell$. In the software-tool CoCoNUT [11], maximal exact matches between S^1 and S^2 are computed by matching S^2 in *forward* direction against an enhanced suffix array of S^1. The bottleneck in large-scale applications like genome comparisons is often the space requirement of the software-tool. If the index structure (e.g. an enhanced suffix array) does not fit into main memory, then it is worthwhile to use a compressed index structure instead. Our new Algorithm 3 computes maximal exact matches by matching S^2 in *backward* direction against a *compressed full-text index* of S^1.

Algorithm 3 proceeds as in the computation of the matching statistics by backward search, i.e., for each position p_2 in S^2, it computes the longest match of $S^2[p_2..n_2]$ with a substring of S^1 of length q, and the matching $S^2[p_2..q-1]$-interval $[lb..rb]$. This time, however, it keeps track of the longest matching path. To be precise, it stores the matching statistics $ms[p_2] = (q, [lb..rb])$ of each position p_2 satisfying $q \geq \ell$ as a triple $(q, [lb..rb], p_2)$ in a list called *path* until a mismatch occurs (i.e., until backward search returns $\perp$). Then, it computes MEMs from the triples in the list *path* (in its outer for-loop). If all elements of the list *path* have been processed, it computes the next longest matching path, and so on.

By construction (or more precisely, by the correctness of Algorithm 2), if the triple $(q, [lb..rb], p_2)$ occurs in some matching path, then $ms[p_2] = (q, [lb..rb])$ and $q \geq \ell$. (Note that for each position p_2 in S^2 at most one triple $(q, [lb..rb], p_2)$ appears in the matching paths.) Clearly, this implies that each $(q, \mathsf{SA}[k], p_2)$ is a longest right maximal exact match at position p_2 in S^2, where $lb \leq k \leq rb$. Now Algorithm 3 tests left maximality by $\mathsf{BWT}[k] \neq S^2[p_2 - 1]$. If $(q, \mathsf{SA}[k], p_2)$ is left maximal, then it is a maximal exact match between S^1 and S^2 with $q \geq \ell$, and the algorithm outputs it. After that, it considers the parent lcp-interval of $[lb..rb]$. Let us denote this parent interval by q'-$[lb'..rb']$. For each k with $lb' \leq k < lb$ or

Algorithm 3. Computing MEMs of length $\geq \ell$ by backward search

```
p_2 ← n_2
(q, [i..j]) ← (0, [1..n_1])
while p_2 ≥ 1 do
  path ← [ ]
  [lb..rb] ← backwardSearch(S^2[p_2], [i..j])
  while [lb..rb] ≠ ⊥ and p_2 ≥ 1 do
    q ← q + 1
    if q ≥ ℓ then
      add(path, (q, [lb..rb], p_2))
    [i..j] ← [lb..rb]
    p_2 ← p_2 − 1
    [lb..rb] ← backwardSearch(S^2[p_2], [i..j])
  for each (q', [lb'..rb'], p'_2) in path do
    [lb..rb] ← ⊥
    while q' ≥ ℓ do
      for each k ∈ [lb'..rb'] \ [lb..rb] do
        if p'_2 = 1 or BWT[k] ≠ S^2[p'_2 − 1] then
          output (q', SA[k], p'_2)
      [lb..rb] ← [lb'..rb']
      q'-[lb'..rb'] ← parent([lb'..rb'])
  if [i..j] = [1..n_1] then
    p_2 ← p_2 − 1
  else
    q-[i..j] ← parent([i..j])
```

$rb < k \leq rb'$, the triple $(q', \mathsf{SA}[k], p_2)$ is a right maximal exact match because $S^1[\mathsf{SA}[k]..\mathsf{SA}[k]+q'-1] = S^2[p_2..p_2+q'-1]$ and $S^1[\mathsf{SA}[k]+q'] \neq S^2[p_2+q']$. So if $q' \geq \ell$ and $\mathsf{BWT}[k] \neq S^2[p_2-1]$, then the algorithm outputs $(q', \mathsf{SA}[k], p_2)$. Then it considers the parent lcp-interval of $[lb'..rb']$ and so on. To sum up, Algorithm 3 checks every right maximal exact match exceeding the length threshold ℓ for left maximality. It follows as a consequence that it detects every maximal exact match of length $\geq \ell$.

We exemplify the algorithm by matching the string $S^2 = caaca$ backwards against the compressed full-text index of $S^1 = acaaacatat\$$. For the length threshold $\ell = 2$, the first matching path is $(2, [8..9], 4)$, $(3, [4..5], 3)$, $(4, [3..3], 2)$. The triple $(2, [8..9], 4)$ yields no output, but for the triple $(3, [4..5], 3)$, the algorithm outputs the MEM $(3, 1, 3)$. (Note that the parent intervals of $[8..9]$ and $[4..5]$ are not considered because their lcp-value is smaller than $\ell = 2$.) The triple $(4, [3..3], 2)$ yields the output $(4, 4, 2)$, and when its parent interval 2-$[2..3]$ is considered, the algorithm does not output the right maximal exact match $(2, 3, 2)$ because it is not left maximal. Now all triples in the matching path have been considered, and the algorithm computes the next longest matching path starting at position $p_2 = 1$ and the parent interval 2-$[2..3]$ of $[i..j] = [3..3]$. This new path consists of the triple $(3, [8..8], 1)$ resulting in the output $(3, 2, 1)$ and $(2, 6, 1)$.

Let us analyse the worst-case time complexity of Algorithm 3. If the outer for-loop was not there, it would run in $O(n_2)$ time; see the run-time analysis of Algorithm 2. In each execution of the while-loop within the outer for-loop, Algorithm 3 tests a right maximal exact match of length $\geq \ell$ for left maximality by $\mathsf{BWT}[k] \neq S^2[p'_2 - 1]$. As a matter of fact, it is not necessary to store the Burrows-Wheeler transformed string $\mathsf{BWT}[1..n_1]$ of S^1. This is because the wavelet tree allows to access the LF-mapping without it, and we have $\mathsf{BWT}[k] \neq c$ if and only if $LF(k) \notin [i..j]$, where $[i..j]$ is the c-interval (e.g., $backwardsearch(c, [1..n_1])$ returns $[i..j]$). In other words, the test $\mathsf{BWT}[k] \neq S^2[p'_2 - 1]$ in Algorithm 3 can be replaced with the test $LF(k) \notin [i..j]$, where $[i..j]$ is the $S^2[p'_2 - 1]$-interval, and this test takes only constant time. Therefore, the algorithm runs in $O(n_2 + z + occ \cdot t_{\mathsf{SA}})$ time, where occ (z) is the number of (right) maximal exact matches of length $\geq \ell$ between the strings S^1 and S^2, and t_{SA} is the access time to the compressed full-text index.

5 Experimental Results

Our implementations are available under the GNU General Public License at `http://www.uni-ulm.de/in/theo/research/seqana`. As already mentioned, the wavelet tree of a string S^1 of length n_1 needs only $n_1 \log |\Sigma| + o(n_1 \log |\Sigma|)$ bits (about $1.25 n_1 \log |\Sigma|$ bits in practice). The balanced parentheses data structure [18] to determine parent intervals requires $2n_1 + o(n_1)$ bits (about $3n_1$ bits in practice). The lcp-array is stored as suggested in [17], i.e., values < 255 are stored in one byte and larger values are stored in an index sorted array—so that larger values are retrieved in logarithmic time. Thus, the lcp-array uses n_1 to $4n_1$ bytes (n_1 to $2n_1$ bytes in practice). Alternatively, one could use Sadakane's [16] encoding of the lcp-array, which uses only $2.1n_1$ bits in practice. However, this slows down the computation considerably. So here is room for improvement.

We conducted experiments to compare our algorithms using backward search with "standard" algorithms using forward search. (Very recently, Russo et al. [20] presented algorithms based on forward search for parallel and distributed compressed indexes. To the best of our knowledge, implementations of their algorithms are not available.)

Test setup: All programs were compiled using `g++` version 4.1.2 with options `-O3 -DNDEBUG` on a 64 bit Linux (Kernel 2.6.16) system equipped with a Dual-Core AMD Opteron processor with 3 GHz and 3GB of RAM. For each test set we measure the real runtime (user time plus system time) of the programs in order to show the effects of swapping, which occurs when a program does not fit into main memory. All programs construct an index (suffix array, wavelet tree) in a first phase and then perform their task based on the index. Because (a) the index can be reused and (b) different programs are used to construct the index, we solely focus on the second phase.

Matching statistics: In machine learning, string kernels in combination with SVM provide string classification algorithms. After a preprocessing phase, an

Size S^1	backwardSK	sask-sl
40MB	144MB	840MB
80MB	277MB	1680MB
120MB	412MB	2520MB
160MB	539MB	3360MB

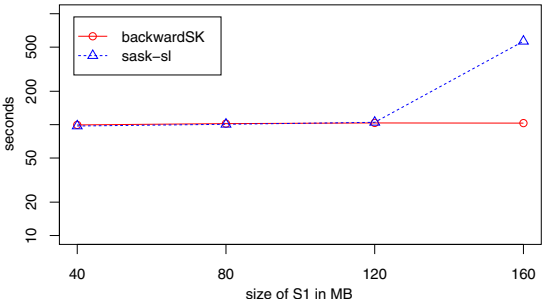

Fig. 2. Left: The amount of memory used by the programs (without the construction phase because the programs use different construction algorithms). Right: Runtime in seconds of the programs to calculate the matching statistics. Note that the y-axis is in log scale.

efficient computation of a string kernel boils down to the computation of matching statistics; see [8]. In this context, Teo and Vishwanathan [8] implemented the linear time algorithm of Abouelhoda et al. [17]. Their program sask-sl is available at `http://users.cecs.anu.edu.au/~chteo/SASK.html`. It uses $18n_1$ to $21n_1$ bytes (n_1 bytes for the string S^1, $4n_1$ bytes for the suffix array, n_1 to $4n_1$ bytes for the lcp-array, $4n_1$ bytes for the child table, and $8n_1$ bytes for the suffix links). In our experiments it used $21n_1$ bytes. We compared their implementation of matching statistics computation with our implementation *backwardSK*. As data we used a concatenation of books from project Gutenberg. Fig. 2 shows the memory usage and the real runtime of the programs to compute the matching statistics, given precalculated supporting data structures of S^1. We evaluated the effect of increasing the size of S^1 while keeping S^2 constant (S^2 has size 20 MB). For sizes up to 120 MB, the runtimes of the programs do not differ significantly. When the size of S^1 reaches 160 MB, however, sask-sl slows down drastically because it does not fit into main memory anymore (so parts of it are swapped out of main memory).

Maximal exact matches: Besides the data structures mentioned above, we used a compressed suffix array based on a wavelet tree that occupies $(n_1 \log n_1)/k$ bits [14]. Its size and the access time t_{SA} to it depend on the parameter $k \geq 1$. For $k = 1$, it is the uncompressed suffix array and the access time is constant. For $k > 1$, only every kth entry of the suffix array is stored and the remaining entries are reconstructed in $k/2$ steps (on average) with the *LF*-mapping (which can be computed with the wavelet tree). In order to compare our implementation *backwardMEM* with other software-tools computing MEMs, we chose the one developed by Khan et al. [21], called *sparseMEM* (`http://compbio.cs.princeton.edu/mems`). Their method computes MEMs between S^1 and S^2 by matching S^2 in *forward* direction against a *sparse* suffix array of S^1, which stores every Kth suffix of S^1, where K is a user-defined parameter. (The reader should be aware of the difference between a *sparse* and

a *compressed* suffix array: A sparse suffix array stores each Kth suffix of S^1, while a compressed suffix array stores each kth entry of the suffix array of S^1.) Our choice is justified by the fact that the sequential version of *sparseMEM* beats the open-source software-tool *MUMmer* [10] and is competitive with the closed-source software-tool *vmatch* (`http://www.vmatch.de/`).

For a fair comparison, we used the sequential version of *sparseMEM* and the same input and output routines as *sparseMEM*. In the experiments, we used the program parameters `-maxmatch -n -l` ℓ, which set the length threshold on the MEMs to ℓ. We ran both programs on DNA-sequences of different species. In the uncompressed case ($k = K = 1$), the memory consumption of *backwardMEM* is smaller than that of *sparseMEM*, but *sparseMEM* is faster. In the compressed cases, *backwardMEM* performs quite impressively, in most cases much better than *sparseMEM*. For example, *backwardMEM* takes only 57s (using 235 MB for $k = 16$) to compare the human chromosome 21 with the mouse chromosome 16, whereas *sparseMEM* takes 10m34s (using 255 MB for $K = 8$); see Table 1.

The space consumption of *sparseMEM* decreases faster with K as that of *backwardMEM* with k, but its running time also increases faster. While the experiments show a clear space-time tradeoff for *sparseMEM*, this is fortunately not the case for *backwardMEM*. Sometimes its running time increases with increasing compression ratio, and sometimes it does not. This is because the algorithm is output-sensitive. More precisely, before a MEM $(q, \mathsf{SA}[i], p_2)$ can be output, the algorithm first has to determine the value $\mathsf{SA}[i]$. While this takes only constant time in an uncompressed suffix array, it takes t_{SA} time in a compressed suffix array, and the value of t_{SA} crucially depends on the compression ratio.

Table 1. For each pair of DNA-sequences (cf. http://compbio.cs.princeton.edu/mems), the time (in minutes and seconds) and space consumption (in MByte) of the programs are shown (without the construction phase). We tested different values of K and k to demonstrate the time-space tradeoff of the algorithms. The value ℓ is the length threshold on the MEMs.

S^1	$\lvert S^1\rvert$	S^2	$\lvert S^2\rvert$	ℓ						
sparseMEM	Mbp		Mbp		$K=1$		$K=4$		$K=8$	
A.fumigatus	29.8	*A.nidulans*	30.1	20	23s	307	3m59s	108	6m13s	74
M.musculus16	35.9	*H.sapiens21*	96.6	50	1m15s	430	10m52s	169	19m56s	163
H.sapiens21	96.6	*M.musculus16*	35.9	50	32s	957	5m08s	362	10m34s	255
D.simulans	139.7	*D.sechellia*	168.9	50	2m17s	1489	21m09s	490	49m34s	326
D.melanogaster	170.8	*D.sechellia*	168.9	50	2m37s	1861	28m49s	588	55m43s	386
D.melanogaster	170.8	*D.yakuba*	167.8	50	2m49s	1860	32m57s	587	61m39s	384
backwardMEM	Mbp		Mbp		$k=1$		$k=8$		$k=16$	
A.fumigatus	29.8	*A.nidulans*	30.1	20	43s	187	49s	89	50s	82
M.musculus16	35.9	*H.sapiens21*	96.6	50	2m09s	261	2m09s	142	2m16s	134
H.sapiens21	96.6	*M.musculus16*	35.9	50	51s	576	59s	258	57s	235
D.simulans	139.7	*D.sechellia*	168.9	50	5m42s	859	17m35s	399	32m39s	366
D.melanogaster	170.8	*D.sechellia*	168.9	50	4m33s	1074	11m19s	504	20m38s	464
D.melanogaster	170.8	*D.yakuba*	167.8	50	3m50s	1068	5m18s	502	7m35s	463

References

1. Weiner, P.: Linear pattern matching algorithms. Proc. 14th IEEE Annual Symposium on Switching and Automata Theory. 1–11 (1973)
2. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words, pp. 85–96. Springer, Heidelberg (1985)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
4. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
5. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
6. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
7. Chang, W., Lawler, E.: Sublinear approximate string matching and biological applications. Algorithmica 12(4/5), 327–344 (1994)
8. Teo, C., Vishwanathan, S.: Fast and space efficient string kernels using suffix arrays. In: Proc. 23rd Conference on Machine Learning, pp. 929–936. ACM Press, New York (2003)
9. Rahmann, S.: Fast and sensitive probe selection for DNA chips using jumps in matching statistics. In: Proc. 2nd IEEE Computer Society Bioinformatics Conference, pp. 57–64 (2003)
10. Kurtz, S., Phillippy, A., Delcher, A., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.: Versatile and open software for comparing large genomes. Genome Biology 5, R12 (2004)
11. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: CoCoNUT: An efficient system for the comparison and analysis of genomes. BMC Bioinformatics 9, 476 (2008)
12. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
13. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article 2 (2007)
15. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
16. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41, 589–607 (2007)
17. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
18. Ohlebusch, E., Gog, S.: A compressed enhanced suffix array supporting fast string matching. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 51–62. Springer, Heidelberg (2009)
19. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theoretical Computer Science 410(51), 5354–5364 (2009)
20. Russo, L., Navarro, G., Oliveira, A.: Parallel and distributed compressed indexes. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 348–360. Springer, Heidelberg (2010)
21. Khan, Z., Bloom, J., Kruglyak, L., Singh, M.: A practical algorithm for finding maximal exact matches in large sequence data sets using sparse suffix arrays. Bioinformatics 25, 1609–1616 (2009)