Space-Efficient Computation of Maximal and Supermaximal Repeats in Genome Sequences

Timo Beller, Katharina Berger, and Enno Ohlebusch

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm
{Timo.Beller,Katharina.Berger,Enno.Ohlebusch}@uni-ulm.de

Abstract. The identification of repetitive sequences (repeats) is an essential component of genome sequence analysis, and the notions of maximal and supermaximal repeats capture all exact repeats in a genome in a compact way. Very recently, Külekci et al. (Computational Biology and Bioinformatics, 2012) developed an algorithm for finding all maximal repeats that is very space-efficient because it uses the Burrows-Wheeler transform and wavelet trees. In this paper, we present a new space-efficient algorithm for finding maximal repeats in massive data that outperforms their algorithm both in theory and practice. The algorithm is not confined to this task, it can also be used to find all supermaximal repeats or to solve other problems space-efficiently.

1 Introduction

In the analysis of a genome, a basic task is to locate and characterize the repetitive sequences (repeats). While bacterial genomes usually do not contain large amounts of repetitive sequences, a considerable portion of the genomes of higher organisms is composed of repeats. For example, more than half of the 3 billion basepairs of the human genome consists of repeats. Clearly, one needs extensive algorithmic support for a systematic study of repetitive DNA on a genomic scale. Although there are already several software tools for finding repeats in genome sequences (see the overview in [11]), the space consumption of these tools—the main issue in large scale applications—can still be improved. Recently, for example, Külekci et al. [13] provided an algorithm that uses only $2n$ bytes to compute all maximal repeats in a DNA sequence of length n. The high-level idea of their algorithm is similar to an algorithm proposed by Becher et al. [2] which uses suffix arrays for the same task. In this paper, we present a new space-efficient algorithm for finding maximal repeats in massive data that outperforms the algorithm of Külekci et al. (called KVX-algorithm in the follwing) both in theory and practice. Our algorithm is based on [3]; there it was shown that the longest common prefix array (LCP-array) of a string S can be computed on the wavelet tree of the Burrows-Wheeler transform of S. So the new algorithm has the usage of this data structure in common with the KVX-algorithm, but it does not need the auxiliary data structures $\mathcal{W}_{lcp}$ and B_{lcp} used by the latter. In contrast to

L. Calderón-Benavides et al. (Eds.): SPIRE 2012, LNCS 7608, pp. 99–110, 2012.

i	SA	LCP	BWT	$S_{\mathsf{SA}[i]}$	lcp-intervals
1	12	−1	i	\$	
2	11	0	p	i\$	
3	8	1	s	$ippi$\$	
4	5	1	s	$issippi$\$	1
5	2	4	m	$ississippi$\$	4
6	1	0	\$	$mississippi$\$	0
7	10	0	p	pi\$	1
8	9	1	i	ppi\$	
9	7	0	s	$sippi$\$	2
10	4	2	s	$sissippi$\$	1
11	6	1	i	$ssippi$\$	3
12	3	3	i	$ssissippi$\$	

Fig. 1. Suffix array, LCP-array, BWT and lcp-intervals of the string $S = mississippi\$$

the KVX-algorithm, which takes $O(n \log n)$ time to *find* all maximal repeats, our algorithm needs only $O(n \log \sigma)$ time, where σ is the size of the underlying alphabet Σ. So from a theoretical point of view, our algorithm is better than the KVX-algorithm. Experiments confirm that it is also much faster in practice; in fact it is more than one order of magnitude faster.

Our algorithm is not limited to maximal repeat finding, it can also be used to find all supermaximal repeats or to solve other problems space-efficiently. This is because on a high level our new algorithm can be viewed as a top-down traversal of a (virtual) suffix tree, and it is well-known that many problems can be solved by traversing suffix trees (see [10]). So the new algorithm is interesting in its own right.

2 Preliminaries

Let Σ be an ordered alphabet of size σ 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 i-th 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 S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$; see Fig. 1 for an example. We refer to the overview article [17] for suffix array construction algorithms (some of which have linear runtime). In the following, ISA denotes the inverse of the permutation SA.

The Burrows and Wheeler transform [4] 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. The permutation LF, defined by $LF(i) = \mathsf{ISA}[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $LF(i) = 1$ otherwise, is called LF-mapping. The LF-mapping can be implemented by $LF(i) = C[c] + Occ(c, i)$, 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 $\mathsf{BWT}[1..i]$.

Ferragina and Manzini [6] showed that it is possible to search a pattern backwards, character-by-character, in the suffix array SA of string S, without storing SA. Let $c \in \Sigma$ and ω be a substring of S. Given the ω-interval $[i..j]$ in the suffix array SA of S (i.e., 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), $backwardSearch(c, [i..j])$ returns the $c\omega$-interval $[C[c] + Occ(c, i-1) + 1 \;..\; C[c] + Occ(c, j)]$. A space efficient data structure that supports backward search and the LF-mapping in $\mathcal{O}(\log \sigma)$ time is the wavelet tree of Grossi et al. [9]. With the wavelet tree it is possible to generalize backward search: for an ω-interval $[i..j]$, a slight modification of the procedure $getIntervals([i..j])$ presented in [3] returns the list of all $c\omega$-intervals, where $c \in \Sigma \setminus \{\$\}$; see also [5]. It has a worst-case time complexity of $O(k \log \sigma)$, where k is the number of elements in the output list.

The suffix array SA is often enhanced with the so-called LCP-array containing the lengths of longest common prefixes between consecutive suffixes in SA; see Fig. 1. Formally, the LCP-array is an array so 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. [12] showed that the LCP-array can be computed in linear time from the suffix array and its inverse. Abouelhoda et al. [1] introduced the concept of lcp-intervals. An interval $[i..j]$, where $1 \leq i < j \leq n$, in the LCP-array 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. $\mathsf{LCP}[j+1] < \ell$.

Every index k, $i+1 \leq k \leq j$, with $\mathsf{LCP}[k] = \ell$ is called ℓ-index. Note that each lcp-interval has at least one and at most $\sigma - 1$ many ℓ-indices. There is a one-to-one correspondence between the set of all lcp-intervals and the set of all internal nodes of the suffix tree of S; see [1]. Consequently, there are at most $n-1$ lcp-intervals for a string of length n.

A substring ω of S is a *repeat* if it occurs at least twice in S. As an example, consider the string $S = mississippi\$$. The substring $issi$ is a repeat of length 4, iss and ssi are repeats of length 3, etc. A repeat ω of S is a *maximal repeat* if any extension of ω occurs fewer times in S than ω. In our example, the substring $issi$ is a maximal repeat but the substrings iss and ssi are not. For instance, the substring ssi is left-extendible: if we extend it by the character i to the left, then the resulting substring $issi$ occurs as often in S as ssi. Analogously, the substring iss is right-extendible. So a repeat is maximal if and only if it is non-left-extendible (left-maximal) and non-right-extendible (right-maximal). A *supermaximal repeat* is a maximal repeat which is not a proper substring of another maximal repeat.

3 Related Work

Gusfield [10, 7.12.1] describes a linear-time algorithm to find all maximal repeats in S, using the suffix tree of S. Subsequently, several other authors provided algorithms for the same task, using different data structures: Raffinot [19] uses a compact suffix automaton of S, Franek et al. [7] use the suffix arrays of both S and its reversed string, Narisawa et al. [14] use the suffix array, the inverse suffix array, and the LCP-array of S, Prieur and Lecroq [16] use a compact suffix vector of S, and Puglisi et al. [18] use the suffix array and the LCP-array of S (essentially using the method of simulating a bottom-up traversal of the suffix tree of S, a method developed by Abouelhoda et al. [1]). Many of these algorithms (implicitly or explicitly) use the fact that lcp-intervals induce candidate repeats which are non-right-extendible (right-maximal).

More recently, two software tools have been developed with the purpose to find maximal repeats in whole genomes. Becher et al. [2] presented an algorithm that accesses LCP-values in increasing order to identify lcp-intervals in increasing order of their lcp-values by using a dynamic data structure: a balanced binary tree of height $\log n$ that is queried and updated in $O(\log n)$ time. Consequently, their maximal repeat finding algorithm has a worst-case time complexity of $O(n \log n)$. The test whether a candidate repeat is non-left-extendible (left-maximal) is done with the aid of the suffix array and the inverse suffix array of S. Their algorithm is not space-efficient: it uses approximately $18n$ bytes. By contrast, Külekci et al. [13] provided an algorithm that uses only $2n$ bytes. Their algorithm shares the same high level idea and the $O(n \log n)$ time complexity with that of Becher et al., but their implementation uses succinct data structures. Moreover, they introduce a clever method to decide whether a candidate repeat is non-left-extendible (left-maximal) that works with just one bit vector; see Section 5 for details.

Gusfield [10, 7.12.2] also presented a linear-time algorithm to find all supermaximal repeats in S, again using the suffix tree of S. Abouelhoda et al. [1] sketched a solution for this task that is based on the suffix array and the LCP-array of S, and Puglisi et al. [18] improved that solution.

4 Space Efficient Enumeration of All lcp-intervals

It is our first goal to show that Algorithm 1 enumerates all lcp-intervals. (At the moment, we ignore all statements in the algorithm that deal with the Boolean variable $locMax$ because $locMax$ is used solely in the computation of supermaximal repeats.) Algorithm 1 maintains a bit array B and, for each character $c \in \Sigma$, a queue Q_c. In the initialization phase, all entries of B are set to zero, except for $B[1]$ and $B[n+1]$. Each queue Q_c initially contains the c-interval. Furthermore, ℓ is set to 0 and the two variables $last_{lb}$ and $last_{idx}$ get the undefined value $\perp$.

As in [3], one can show that Algorithm 1 computes LCP-values in increasing order. To be precise, for increasing values of ℓ, it determines all indices i with $\mathsf{LCP}[i] = \ell$ (for space reasons, the proof is not repeated here). However, because we are not interested in the computation of the LCP-array, we just set

Algorithm 1. Space efficient enumeration of lcp-intervals

initialize a bit vector $B[1..n+1]$ /* i.e., $B[i] = 0$ for all $1 \leq i \leq n+1$ */
$B[1] \leftarrow 1$
$B[n+1] \leftarrow 1$
for each c **in** Σ **do**
  initialize an empty queue Q_c
  $enqueue(Q_c, [C[c]+1..C[c+1]])$ /* the c-interval */
$\ell \leftarrow 0$
$last_{lb} \leftarrow \perp$
$last_{idx} \leftarrow \perp$
$locMax \leftarrow true$
while there is a non-empty queue **do**
  for each c **in** Σ **do**
    $size[c] \leftarrow |Q_c|$ /* current size of the queue Q_c */
  for each c **in** Σ **do** /* in alphabetical order */
    while $size[c] > 0$ **do**
      $[lb..rb] \leftarrow dequeue(Q_c)$ /* $[lb..rb]$ is the ω-interval for some ω */
      $size[c] \leftarrow size[c] - 1$
      if $B[rb+1] = 0$ **then** /* case 1: $rb+1$ is an ℓ-index */
        $B[rb+1] \leftarrow 1$
        if $lb \neq rb$ **then**
          $locMax \leftarrow false$
        if $last_{lb} = \perp$ **then**
          $last_{lb} \leftarrow lb$
        $last_{idx} \leftarrow rb+1$
        $list \leftarrow getIntervals([lb..rb])$
        for each $(c, [i..j])$ **in** $list$ **do**
          $enqueue(Q_c, [i..j])$
      else if $last_{idx} = lb$ **then** /* case 2: $last_{idx}$ is last ℓ-index */
       /* the lcp-interval ℓ-$[last_{lb}..rb]$ has not been considered before */
        if $lb \neq rb$ **then**
          $locMax \leftarrow false$
        $process(\langle \ell, last_{lb}, rb, locMax \rangle)$
        $last_{lb} \leftarrow \perp$
        $last_{idx} \leftarrow \perp$
        $locMax \leftarrow true$
        $list \leftarrow getIntervals([lb..rb])$
        for each $(c, [i..j])$ **in** $list$ **do**
          $enqueue(Q_c, [i..j])$
      else nothing to do /* case 3 */
  $\ell \leftarrow \ell + 1$

ℓ	$Q_\$$	Q_i	Q_m	Q_p	Q_s
0	$\$_{[1..1]}$	$i_{[2..5]}$	$m_{[6..6]}$	$p_{[7..8]}$	$s_{[9..12]}$
1		$i\$_{[2..2]}, ip_{[3..3]}, is_{[4..5]}$	$mi_{[6..6]}$	$pi_{[7..7]}, pp_{[8..8]}$	$si_{[9..10]}, ss_{[11..12]}$
2		$ipp_{[3..3]}, iss_{[4..5]}$	$mis_{[6..6]}$	$pi\$_{[7..7]}, ppi_{[8..8]}$	$sip_{[9..9]}, sis_{[10..10]}, ssi_{[11..12]}$
3					$ssip_{[11..11]}, ssis_{[12..12]}$
4		$issip_{[4..4]}, issis_{[5..5]}$			
5			$missis_{[6..6]}$		$sissip_{[10..10]}$

Fig. 2. Contents of the queues for increasing values of ℓ when Algorithm 1 is applied to the example of Fig. 1. Intervals belonging to case 1 are wavy underlined, intervals belonging to case 2 are underlined, and intervals belonging to case 3 are not underlined.

$B[i] = 1$ whenever such an index i is detected. In contrast to the algorithm in [3], Algorithm 1 enforces that—for a fixed value of ℓ—the indices $i_1, \ldots, i_q$ with $\mathsf{LCP}[i_k] = \ell$ are found in increasing order $i_1 < \ldots < i_q$. Moreover, our new algorithm must consider more intervals because of case 2.

Let us illustrate this with the example of Fig. 1. After the initialization phase, $Q_\$$ contains the \$-interval [1..1], Q_i contains the i-interval [2..5], and so on; see row $\ell = 0$ in Fig. 2. In the while-loop of Algorithm 1, $size[c]$ is set to the current size of queue Q_c for each $c \in \Sigma$; in the first iteration, we have $size[c] = 1$ for each $c \in \Sigma$. Then, the algorithm accesses the queues in alphabetical order. In our example, it first removes the interval $[lb..rb] = [1..1]$ from $Q_\$$. Since $B[rb+1] = B[2] = 0$ (case 1), the algorithm has detected the first index $i_1 = 2$ with $\mathsf{LCP}[i_1] = \ell = 0$. It sets $B[2] = 1$ to mark that this LCP-entry is now known. Furthermore, it sets $last_{lb} = lb = 1$ and $last_{idx} = rb + 1 = 2$. The procedure call $getInterval([1..1])$ returns a list that contains just the $i\$$-interval [2..2], which is added to the queue Q_i; see Fig. 2. The intervals [2..5], [6..6], and [7..8] are processed similarly (in this order), and the algorithm detects that the indices $i_2 = 6$, $i_3 = 7$, and $i_4 = 9$ satisfy $\mathsf{LCP}[i_k] = \ell = 0$. Thus, afterwards $B[6] = B[7] = B[9] = 1$ and $last_{idx} = 9$ holds. Finally, when the interval $[lb..rb] = [9..12]$ is processed, we have $B[rb+1] = 1$ (recall that $\mathsf{LCP}[n+1] = -1$) and $last_{idx} = lb$. So case 2 applies, and the algorithm has found an lcp-interval, namely the interval $[last_{lb}..rb] = [1..12]$ of lcp-value $\ell = 0$. This is because $\mathsf{LCP}[1] = -1 < 0$, $\mathsf{LCP}[2] = \mathsf{LCP}[6] = \mathsf{LCP}[7] = \mathsf{LCP}[9] = 0$, and $\mathsf{LCP}[13] = -1$. Note that for $k \in \{3, 4, 5, 8, 10, 11, 12\}$ the inequality $\mathsf{LCP}[k] > 0$ must hold because $B[k] = 0$. The generic procedure *process* "processes" the lcp-interval, the variables $last_{lb}$ and $last_{idx}$ are reset to $\perp$, and—as in case 1—new intervals are generated and added to the queues. In the last statement of the while-loop ℓ is incremented by one. The contents of the queues at this point in time is depicted in row $\ell = 1$ of Fig. 2. The reader is invited to compute the first lcp-interval [2..5] of lcp-value 1 by "executing" the algorithm with the intervals in the queue Q_i.

In general, for a fixed value of ℓ, the while-loop of Algorithm 1 computes the indices $i_1, \ldots, i_q$ with $\mathsf{LCP}[i_k] = \ell$ in increasing order $i_1 < \ldots < i_q$. When i_1, the first of these indices, is detected, the variable $last_{lb}$ memorizes the left boundary of the interval under consideration. If there are further ℓ-indices, say $i_2, \ldots, i_p$, then these are identified one after the other (by case 1) until the last ℓ-index is found (by case 2). Recall that $B[rb+1] = 1$ means that the index $rb+1$ has an LCP-value that is strictly smaller than ℓ; so it is the right boundary of the lcp-interval that started at $last_{lb}$. Now, the lcp-interval ℓ-$[last_{lb}..rb]$ can be processed. It should be pointed out that all lcp-intervals of lcp-value ℓ are found in this way (if $i_p \neq i_q$, then i_{p+1} is the first ℓ-index of the next lcp-interval of lcp-value ℓ, etc.). Since the algorithm proceeds in this way for increasing values of ℓ, it enumerates all lcp-intervals. It may happen, however, that the procedure *getIntervals* generates intervals that do not lead to a new value in the LCP-array. In our example, the mi-interval [6..6] is such an interval; see row $\ell = 1$ of Fig. 2. Immediately before Algorithm 1 processes this interval, the lcp-interval [2..5] of lcp-value 1 was detected, and $last_{lb}$ and $last_{idx}$ were reset to $\perp$. For the mi-interval $[lb..rb] = [6..6]$, we have $B[rb+1] = B[7] = 1$ (the value $\mathsf{LCP}[7] = 0$ was detected before) and $last_{idx} \neq lb$ (no ℓ-index of the next lcp-interval was found yet), so none of the cases 1 or 2 applies. In this case 3, the algorithm does nothing.

An amortized analysis will show that Algorithm 1 has a worst-case time complexity of $\mathcal{O}(n \log \sigma)$. We prove that each of the cases 1, 2, and 3 can occur at most n times. Case 1 occurs as often as a bit of B is set to 1 in the while-loop, and this happens exactly $n-1$ times. Whenever case 2 occurs, the algorithm processes a different lcp-interval. As there are at most $n-1$ lcp-intervals, this happens at most $n-1$ times. It remains to analyse how often case 3 can occur. We claim that for a fixed position j, $1 \leq j \leq n$, there is at most one substring $\omega = S[i..j]$ ending at j for which the ω-interval $[lb..rb]$ belongs to case 3. If i is the largest position with $\omega = S[i..j]$ so that the ω-interval $[lb..rb]$ belongs to case 3, then none of the left-extensions of ω is generated. More precisely, none of the ω'-intervals, where $\omega' = S[i'..j]$ with $1 \leq i' < i$, will be enqueued. This proves the claim. As there are only n possibilities for j, it follows that case 3 also occurs at most n times. In summary, the procedure *getIntervals* can create at most $3n$ intervals because every interval belongs to exactly one case. Each interval can be generated in $\mathcal{O}(\log \sigma)$ time, so the runtime of Algorithm 1 is $\mathcal{O}(n \log \sigma)$.

Algorithm 1 uses $nH_0(\mathsf{BWT}) + 5n + o(n)$ bits of space. The wavelet tree of a text T of length n uses $nH_0(T) + O(n \log\log n / \log_\sigma n)$ bits, where $H_0(T)$ is the 0-order empirical entropy of T; see [13]. Clearly, the bit vector B uses n bits. Finally, the queues can be implemented with only $4n + o(n)$ bits. Essentially, this is because the left (right, respectively) boundaries of generated intervals form a sequence of strictly increasing numbers (details omitted).

5 Finding Maximal and Supermaximal Repeats

We start with characterizations of maximal and supermaximal repeats. Lemma 1 can be found in [13, Lemma 6] (using a different terminology though), and Lemma 2 was proved in [1].

Lemma 1. *A substring ω of S is a maximal repeat if and only if the ω-interval $[i..j]$ is an lcp-interval of lcp-value $\ell = |\omega|$, and the characters $\mathsf{BWT}[i], \mathsf{BWT}[i+1], \ldots, \mathsf{BWT}[j]$ are not all the same.*

Lemma 2. *A substring ω of S is a supermaximal repeat if and only if the ω-interval $[i..j]$ is a* local maximum *in the LCP-array (i.e., $[i..j]$ is an lcp-interval of lcp-value $\ell = |\omega|$ so that $\mathsf{LCP}[k] = \ell$ for all $i+1 \leq k \leq j$), and the characters $\mathsf{BWT}[i], \mathsf{BWT}[i+1], \ldots, \mathsf{BWT}[j]$ are pairwise distinct.*

In view of Lemmas 1 and 2, we say that an lcp-interval ℓ-$[i..j]$ *induces* a maximal (supermaximal) repeat if the string $\omega = S[\mathsf{SA}[i]..\mathsf{SA}[i] + \ell - 1]$ is a maximal (supermaximal) repeat.

Let us turn to the problem of finding all maximal repeats. Algorithm 1 enumerates all lcp-intervals, and when the procedure *process* is called with the lcp-interval ℓ-$[i..j]$ (ignore the Boolean parameter $locMax$), then it must be tested whether this interval induces a maximal repeat. According to Lemma 1, this is the case if and only if the characters in $\mathsf{BWT}[i..j]$ are not all the same. Using a clever idea of [13], this test can be done in constant time with a bit vector $B_{\mathsf{BWT}}[1..n]$ which initially contains a series of zeros. In a linear scan of the BWT, we set $B_{\mathsf{BWT}}[i] = 1$ if $\mathsf{BWT}[i] \neq \mathsf{BWT}[i-1]$. Then, the bit vector is preprocessed so that rank queries can be answered in constant time. A rank query $rank_b(B_{\mathsf{BWT}}, i)$ returns the number of occurrences of bit b in $B_{\mathsf{BWT}}[1..i]$. Clearly, the characters in $\mathsf{BWT}[i..j]$ are not all the same if and only if $rank_1(B_{\mathsf{BWT}}, j) - rank_1(B_{\mathsf{BWT}}, i) > 0$. It is useful to restrict the output to maximal repeats that have a certain minimum length ml (which usually can be defined by the user) because short repeats are somewhat meaningless. Moreover, it may (or may not) be useful to restrict the output to maximal repeats that occur at least mo times in the string S. These constraints can easily be incorporated by testing whether an lcp-interval ℓ-$[i..j]$ satisfies $\ell \geq ml$ and $j - i + 1 \geq mo$. Algorithm 2 first tests whether the lcp-interval satisfies these constraints. If so, it checks whether it induces a supermaximal repeat. If this is not the case, it tests whether it induces a maximal repeat.

It remains to address the problem of finding all supermaximal repeats. According to Lemma 2, an lcp-interval ℓ-$[i..j]$ induces a supermaximal repeat if and only if (a) it is a local maximum in the LCP-array (i.e., $\mathsf{LCP}[k] = \ell$ for all $i+1 \leq k \leq j$) and (b) the characters $\mathsf{BWT}[i], \mathsf{BWT}[i+1], \ldots, \mathsf{BWT}[j]$ are pairwise distinct.

(a) Since Algorithm 1 successively considers the intervals $[i..i_1 - 1], [i_1..i_2 - 1], \ldots, [i_q..j]$, where $i_1, i_2, \ldots, i_q$ are exactly the ℓ-indices of $[i..j]$, the interval $[i..j]$ is a local maximum if and only if each of these intervals is a singleton interval. The statements in Algorithm 1 that deal with the Boolean variable

Algorithm 2. Procedure $process(\langle \ell, i, j, locMax \rangle)$ tests whether the lcp-interval ℓ-$[i..j]$ induces a supermaximal or a maximal repeat of length at least ml, having at least mo occurrences in S. If so, it reports the repeat.

```
if ℓ ≥ ml and j − i + 1 ≥ mo then
  if locMax = true and superMax([i..j]) then        /* short-circuit evaluation */
      report that ℓ-[i..j] induces a supermaximal repeat
  else if rank_1(B_BWT, j) − rank_1(B_BWT, i) > 0 then
    report that ℓ-[i..j] induces a maximal repeat
```

Algorithm 3. Procedure $superMax([i..j])$ returns true if and only if the characters in $\mathsf{BWT}[i..j]$ are pairwise distinct

```
pd ← true
list ← getIntervals([i..j])
for each interval [p..q] in list do
  if p ≠ q then pd ← false
return pd
```

$locMax$ make sure that when the procedure $process$ is called with the parameters ℓ, i, j and the Boolean parameter $locMax$, we have $locMax = true$ if and only if the lcp-interval ℓ-$[i..j]$ is a local maximum.
(b) The procedure $superMax([i..j])$ in Algorithm 3 returns true if and only if the characters in $\mathsf{BWT}[i..j]$ are pairwise distinct. This can be seen as follows. $[i..j]$ is the ω-interval for some substring ω of S. A character c occurs exactly once in $\mathsf{BWT}[i..j]$ if and only if the $c\omega$-interval is a singleton interval. So the characters in $\mathsf{BWT}[i..j]$ are pairwise distinct if and only if each interval in the list returned by the procedure $getIntervals([i..j])$ is a singleton interval. Of course, in an actual implementation it is best to integrate the functionality of the procedure $superMax$ into the procedure $getIntervals$.

Algorithms 1 - 3 simultaneously compute all supermaximal and maximal repeats. We claim that the overall runtime is $O(n \log \sigma)$. We have seen that the worst-case time complexity of Algorithm 1 is $O(n \log \sigma)$. The procedure $process$ is called at most $n-1$ times (there are at most $n-1$ lcp-intervals) and the execution of each statement of procedure $process$ takes only constant time, except for the call to the procedure $superMax$. We use an amortized analysis to show that the overall time consumed by all calls to the procedure $superMax$ is $O(n \log \sigma)$. Let $[i_1..j_1], [i_2..j_2], \ldots, [i_q..j_q]$ be the local maxima in the LCP-array. The procedure $superMax$ applied to one of the intervals, say $[i_p..j_p]$, takes $O(k \log \sigma)$ time, where k is the size of the list returned by $getIntervals([i_p..j_p])$. Clearly, $k \leq j_p - i_p + 1$ (equality holds if all characters in $\mathsf{BWT}[i_p..j_p]$ are pairwise distinct). The key observation is that any two local maxima must be disjoint. We conclude that the overall time consumed by all calls to the procedure $superMax$ is $O((\sum_{p=1}^{q}(j_p - i_p + 1)) \log \sigma) = O(n \log \sigma)$.

6 Experimental Results

We compared our maximal repeat finding algorithm with the findpat algorithm of Becher et al. [2] and the KVX-algorithm of Külekci et al. [13], which were kindly provided by the authors. The algorithm PSY1 of Puglisi et al. [18] is faster than those of [7,14], but unfortunately their implementation is not available any more. That is why we use our own implementation of the bottom-up method. In fact, we use an algorithm that is simpler than that of [18].[1]

All programs were compiled with the options `-O9 -funroll-loops -msse4.2` using `gcc` version 4.4.3 on a 64 bit Ubuntu (Kernel 2.6.32) system equipped with a six-core AMD Opteron processor 2431 with 2.4 GHz (but no parallelism was used) and 32GB of RAM. We tested our algorithm on the data that was used in [13]: the DNA of the human genome,[2] the proteins file from the Pizza&Chili Corpus,[3] and English texts from a wikipedia dump.[4] Table 1 shows the results for the 400MB and 1GB prefixes of these files; note that we used the whole file of the human genome instead of its 1GB prefix. The bottom-up method needs the arrays SA, BWT, and LCP. The suffix array construction was done by Mori's libdivsufsort-algorithm;[5] because the 32 bit version (which needs $5n$ bytes in total) is limited to files of size $\leq 2^{31}$, we had to use the 64 bit version (which needs $9n$ bytes in total) for the human genome. Furthermore, the *go*Φ algorithm[6] of Gog and Ohlebusch [8] was used to compute the BWT and the LCP-array. Both our new algorithm and the KVX-algorithm require the BWT as starting point. In our experiments, we used the implementation of Okanohara and Sadakane [15] to directly construct the BWT.

Table 1 shows the experimental results for finding all lcp-intervals that induce maximal repeats of minimum length $\lceil \log n \rceil$. Let us concentrate on the 2729MB DNA file containing the human genome because repeat finding in genome sequences is the main application. The bottom-up method is very fast provided that the needed data structures (SA&BWT&LCP) are available, but the construction of these data structures takes a lot more time than building the BWT directly. Although the human genome is quite large and the construction of its suffix array requires $9n$ bytes, it still fits into the 32GB of RAM of our computer. However, there are many organisms that have larger genomes; e.g. the largest vertebrate genome known (*Protopterus aethiopicus*) has 130GB. The main issue in such large scale applications is the space consumption, and that is why Külekci et al. [13] developed their space-efficient algorithm. Our algorithm is as space-efficient as their algorithm, but it is much faster. In case of the human genome, for example, the KVX-algorithm takes over 19 hours, whereas our algorithm takes only 46 minutes.

[1] For space reasons, supermaximal repeats are not treated here.
[2] `ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/`
[3] `http://pizzachili.dcc.uchile.cl/texts.html`
[4] `http://download.wikimedia.org/enwiki/20100730/enwiki-20100730-pages-articles.xml.bz2`
[5] `http://code.google.com/p/libdivsufsort/`
[6] `https://github.com/simongog/sdsl`

Table 1. For each file, the first column shows the real runtime in seconds and the second column shows the maximum memory usage per character. As an example, the construction of the BWT of the 400MB DNA file takes 179 sec. and $1.8n$ bytes (720MB). The first two rows refer to the construction time of the needed data structures. Rows 3-5 show the runtime and space usage for finding maximal repeats under the assumption that the needed data structures have already been build. The output consists solely of the lcp-intervals that induce the maximal repeats. The last three rows show the overall runtime and maximum memory usage per character for computing the needed data structures and the maximal repeats.

	DNA ($\sigma = 4$)		proteins ($\sigma = 27$)		English ($\sigma = 211$)	
	400MB	2729MB	400MB	1000MB	400MB	1000MB
SA&BWT&LCP	256 5.0	3,220 8.9	302 5.0	898 5.0	229 5.0	716 5.0
BWT	179 1.8	1,705 1.9	286 2.4	776 2.3	209 2.1	679 2.1
bottom-up	22 4.6	209 5.0	21 4.6	63 4.8	16 4.6	60 4.8
KVX-algorithm	6,072 1.5	68,595 1.6	6,950 2.6	20,514 2.6	7,739 2.7	23,173 2.7
new algorithm	145 1.1	1,001 1.1	294 1.4	819 1.4	195 1.6	520 1.6
bottom-up	278 5.0	3,428 8.9	322 5.0	961 5.0	245 5.0	777 5.0
KVX-algorithm	6,251 1.8	70,300 1.9	7,235 2.6	21,290 2.6	7,948 2.7	23,851 2.7
new algorithm	324 1.8	2,706 1.9	579 2.4	1,595 2.3	404 2.1	1,199 2.1

Table 1 does not include the time to explicitly output all maximal repeats and their starting positions in the text. For example, an lcp-interval ℓ-$[i..j]$ induces the maximal repeat $S[\mathsf{SA}[i]..\mathsf{SA}[i] + \ell - 1]$ with the starting positions $\mathsf{SA}[k]$, $i \leq k \leq j$. Table 2 shows the results that include this output. It can be seen that the bottom-up method has an advantage over the space-efficient methods. This is because it keeps the suffix array in main memory, whereas the other methods must use a sampled (compressed) suffix array instead, in which only each k-th entry is directly accessible; see [13] for details. Both the KVX-algorithm and our algorithm have a memory peak at $1.9n$ bytes during the computation of the BWT of the human genome, and both use less space in the repeat finding phase. However, the KVX-algorithm uses a sample rate of $k = 32$, while our algorithm can afford a sample rate of $k = 8$ to stay below the $1.9n$ bytes limit.

Table 2. Real runtime in seconds and maximum memory usage per character for computing lcp-intervals *and* the maximal repeats they induce *and* their starting positions in the text. The construction of needed data structures is included. Note that findpat needs more than 32GB of RAM to cope with the human genome.

	DNA ($\sigma = 4$)		proteins ($\sigma = 27$)		English ($\sigma = 211$)	
	400MB	2729MB	400MB	1000MB	400MB	1000MB
findpat	1,907 18.2	- -	2,257 18.2	6,897 18.2	1,559 18.2	4,877 18.2
bottom-up	332 5.0	4,162 8.9	412 5.0	1,227 5.0	255 5.0	837 5.0
KVX-algorithm	10,001 1.8	113,557 1.9	17,578 2.6	51,876 2.6	11,276 2.7	35,997 2.7
new algorithm	817 1.8	9,248 1.9	3,453 2.4	10,102 2.3	1,354 2.1	5,372 2.1

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
2. Becher, V., Deymonnaz, A., Heiber, P.: Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. Bioinformatics 25(14), 1746–1753 (2009)
3. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the Longest Common Prefix Array Based on the Burrows-Wheeler Transform. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 197–208. Springer, Heidelberg (2011)
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
5. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k Ranked Document Search in General Text Databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
6. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
7. Franek, F., Smyth, W.F., Tang, Y.: Computing all repeats using suffix arrays. Journal of Automata, Languages and Combinatorics 8(4), 579–591 (2003)
8. Gog, S., Ohlebusch, E.: Lightweight LCP-array construction in linear time (2011), `http://arxiv.org/pdf/1012.4263`
9. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
10. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
11. Haas, B.J., Salzberg, S.L.: Finding repeats in genome sequences. In: Lengauer, T. (ed.) Bioinformatics — From Genomes to Therapies, Volume 1: Molecular Sequences and Structures, ch. 7, Wiley-VCH Verlag (2007)
12. 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)
13. Külekci, M.O., Vitter, J.S., Xu, B.: Efficient maximal repeat finding using the Burrows-Wheeler transform and wavelet tree. IEEE/ACM Transactions on Computational Biology and Bioinformatics 9(2), 421–429 (2012)
14. Narisawa, K., Inenaga, S., Bannai, H., Takeda, M.: Efficient Computation of Substring Equivalence Classes with Suffix Arrays. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 340–351. Springer, Heidelberg (2007)
15. Okanohara, D., Sadakane, K.: A Linear-Time Burrows-Wheeler Transform Using Induced Sorting. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 90–101. Springer, Heidelberg (2009)
16. Prieur, E., Lecroq, T.: On-line construction of compact suffix vectors and maximal repeats. Theoretical Computer Science 407(1-3), 290–301 (2008)
17. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
18. Puglisi, S.J., Smyth, W.F., Yusufu, M.: Fast, practical algorithms for computing all the repeats in a string. Mathematics in Computer Science 3(4), 373–389 (2010)
19. Raffinot, M.: On maximal repeats in strings. Information Processing Letters 80(3), 165–169 (2001)