Discrete Mathematics and Theoretical Computer Science DMTCS vol. **15:2**, 2013, 59–70

Efficient repeat finding in sets of strings via suffix arrays

Pablo Barenbaum[1] Verónica Becher[1,2] Alejandro Deymonnaz[1]
Melisa Halsband[1] Pablo Ariel Heiber[1,2†]

[1]*Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires*
[2]*CONICET Argentina*

received 6th April 2012, *revised 3rd April 2013*, *accepted 21st April 2013*.

We consider two repeat finding problems relative to sets of strings: (a) Find the largest substrings that occur in every string of a given set; (b) Find the maximal repeats in a given string that occur in no string of a given set. Our solutions are based on the suffix array construction, requiring $\mathcal{O}(m)$ memory, where m is the length of the longest input string, and $\mathcal{O}(n \log m)$ time, where n is the the whole input size (the sum of the length of each string in the input). The most expensive part of our algorithms is the computation of several suffix arrays. We give an implementation and experimental results that evidence the efficiency of our algorithms in practice, even for very large inputs.

Keywords: stringology, repeats, suffix array, longest maximal substring

1 Introduction

The difficulty of finding repeats within a *set of strings*, as opposed to within a single string, is that the size of the input can grow significantly. A typical setting happens when the total size of the set exceeds the available memory, while each element in the set fits in it. One would like a solution that does not rely on secondary memory and has an acceptable, close to linear, running time. In this paper we address two problems relative to sets of strings:

(a) Find the largest substrings that occur in every string of a given set.

(b) Find the maximal repeats in a given string that occur in no string of a given set. We consider the variants of this second problem for two notions maximality: one is maximality with respect to the substring relation; the other says that a repeat is maximal if any of its extensions occur fewer times.

We define a data structure that allows us to solve these problems efficiently. It stores which substrings of a given string occur in all the elements of a set, and which occur in none. To build this structure no more than two strings are processed at the same time. Our algorithms are based on the *suffix array* construction [8] and require $\mathcal{O}(n \log m)$ time and $\mathcal{O}(m)$ memory, where n is the size of the input (sum of the all the strings' lengths) and m is the length of the longest input string. The most expensive part of our algorithms is the computation of several suffix arrays. The use of suffix trees instead of suffix arrays may lower the asymptotic time complexity bounds to be linear on n. However, as it is widely known, the large constants associated to the suffix tree construction make the suffix array approach better in most cases. For comparison on the theoretical and practical utility of both structures see [1, 9].

To our knowledge, no non-trivial solution has been proposed for this exact problem. However, a very related problem on sets of strings, also solved with suffix arrays, is treated by Babenko and Starikovskaya in [2]. They tackle, in an ingenious way, the problem of finding the *longest* repeat that appears in at least k strings of a set, for each k. Their algorithm, however, requires to store all strings simultaneously in memory, yielding a $\mathcal{O}(n)$ memory requirement and $\mathcal{O}(n \log n)$ time. In contrast, our algorithms just require the suffix arrays of pairs of strings and thus have lower time and memory bounds.

The problems we solve in this paper were motivated by applications in comparative genomic sequence analysis, where the huge volume of the input data and the abundance of repeated subsequences demand

†emails: {pbarenbaum,vbecher,adeymo,mhalsband,pheiber}@dc.uba.ar.

algorithms that are efficient in the trade-off complexity of time and memory. Nevertheless, applications for cross-search on sets of strings arise naturally in the context of text search applications. While heuristics are the usual tool to tackle this problems in massive real life applications like search engines, research of efficient exact solutions is of interest, at least to provide a basis for comparison, but also to eventually combine the new techniques and ideas found with the methods used in practice. Applications in genomics also rise interesting open questions regarding how to incorporate the possibility of errors in the data, thus requiring inexact matching algorithms.

In the last section, we describe an implementation of our algorithms and their performance on real datasets, including various possible scenarios such as genomic data, source code and literary work.

2 Notation and definitions

Notation. Assume the alphabet $\mathcal{A}$, a finite set of symbols. A string is a finite sequence of symbols in $\mathcal{A}$. The length of a string w, denoted by $|w|$, is the number of symbols in w. We address the positions of a string w by counting from 1 to $|w|$. The symbol in position i is denoted $w[i]$, and $w[i..j]$ represents the substring of length $j - i + 1$ that starts in position i of w. We say u is a substring of w if $u = w[i..j]$ for some i, j. If u is a substring of w we say that u occurs in w at position i if $u = w[i..i + |u| - 1]$. A prefix of a string w is an initial segment of w, $w[1..i]$; a suffix of w is a final segment $w[i..|w|]$. When u is a substring of w we call w an extension of u. Given a string w and a set of strings X, we say that w occurs in X if w is a substring of some $x \in X$.

2.1 Maximal repeats

We use the nomenclature given by Gusfield [5], and extend it to definitions over sets of strings. We use a different language than that on Gusfield's book, however, the definitions are equivalent.

Definition 1 (Maximal and supermaximal repeats [5])

1. *A* maximal repeat *in a string w is a substring that occurs more than once in w, and each of its extensions occur fewer times in w.*
2. *A* supermaximal repeat *in a string w is a substring that occurs more than once in w, and each of its extensions occur at most once in w.*

Definition 2 (Supermaximal repeat in a set, exclusive maximal and supermaximal repeat)

1. *Given a set X with at least two strings, a supermaximal repeat in X is a substring of each $x \in X$, such that none of its extensions occur in every $x \in X$.*
2. *Given a string w and a set of strings X, an* exclusive maximal (respectively supermaximal) repeat *in w with respect to X is a maximal (respectively supermaximal) repeat in w that does not occur in X.*

Example 3 The maximal repeats in $w = abcdeabcdfbcde$ are $abcd$, $bcde$, and bcd. Clearly $abcd$ and $bcde$ are maximal repeats, occurring twice. But also bcd is a maximal repeat because it occurs three times in w, and every extension of bcd occurs fewer times. There are no other maximal repeats in w (bc, for example, occurs three times, but since bcd occurs the same number of times, bc is not a maximal repeat). Of these, only $abcd$ and $bcde$ are supermaximal repeats. The only exclusive maximal repeat in $w = abcdeabcdfbcde$ with respect to $S = \{fabcd, bcdf, abce\}$ is $bcde$, which is also the only exclusive supermaximal repeat. There is just one supermaximal repeat in S, the string bc.

This example already shows that maximal repeats can be nested and overlapping, and the same applies to exclusive maximal repeats. Supermaximal repeats (in a set and exclusive), however, can be overlapping but not nested.

Theorem 4 ([5], Theorem 7.12.1) *The number of supermaximal repeats in a string is less than or equal to the number of maximal repeats in a string, which is, in turn, less than or equal to the string length.*

Proposition 5 *The set of exclusive maximal (respectively supermaximal) repeats in a given string w with respect to any set of strings, is included in the set of maximal (respectively supermaximal) repeats of w.*

Proof: Immediate from the definitions. □

Proposition 6

1. *The number of supermaximal repeats in a set of strings is not greater than the minimum of all string lengths.*
2. *The number of exclusive maximal (respectively supermaximal) repeats in a given string with respect to a set is not greater than the string length.*

Proof: Supermaximal repeats occur in every string in the given set, and they can not be nested with other supermaximal repeats. This means that at each position in a given string in the set, at most one supermaximal repeat starts. Hence, the length of a shortest string in the set gives an upper bound to the total number of supermaximal repeats; this proves point 1. Point 2 is immediate from Theorem 4 and Proposition 5. □

2.2 Suffix array and Longest Common Prefix

Let w be a string of length $n = |w|$. The suffix array [8] of w is a permutation r of the indices $1...n$ such that for each $i < j$, $w[r[i]..n]$ is lexicographically less than $w[r[j]..n]$. Thus, a suffix array represents the lexicographic order of all suffixes of the input w. For convenience, sometimes we also store the inverse permutation of r and call it p, namely, $p[r[i]] = i$. In our procedures, we use the fast algorithm of Larsson and Sadakane [7] to build suffix arrays of some of the strings of the input. This algorithm profits from the fact that the elements to be ordered are suffixes. Its worst case time complexity is $\mathcal{O}(n \log n)$.

Each substring of w can be seen as a prefix of a suffix of w. Suppose a maximal repeat u occurs k times in w; then, it is a prefix of k different suffixes of w. Since the suffix array r records the lexicographical order of the suffixes of w, the maximal repeat u can be seen as a string of length $|u|$ addressed by k consecutive indices of r. Namely, there will be an index i such that u occurs at positions $r[i], r[i+1]$,..., and $r[i+k-1]$ of w (see [4] for a detailed analysis of this point).

We write $lcp(u, v)$ to denote the length of the longest common prefix of the strings u and v. We use the linear time algorithm of Kasai et al. [6] to compute the lcp value of each pair of consecutive suffixes of w in the lexicographic order. This elegant algorithm builds the LCP (Longest Common Prefix) array by examining the suffixes of the input w in decreasing length order, and by comparing each suffix to its adjacent entry on the suffix array.

Definition 7 *For each position* $1 \leq i < n$*,* $LCP[i] = lcp(w[r[i]..n], w[r[i+1]..n])$.

Proposition 8 *For any* i, j *such that* $1 \leq i < j \leq n$*,*

$$lcp(w[r[i]..n], w[r[j]..n]) = \min\{LCP[k] \ : \ i \leq k < j\}.$$

Proof: Let $t = lcp(w[r[i]..n], w[r[j]..n])$. Since r is lexicographically sorted, the longest common prefix of $w[r[i]..n]$ and $w[r[j]..n]$ is also a prefix of each suffix in between. Therefore, $LCP[k] \geq t$ for each k in range $[i, j]$. By way of contradiction, assume that for all such k, $LCP[k] > t$. Consider the pairs of strings $w[r[k]..r[k]+t]$ and $w[r[k+1]..r[k+1]+t]$ of length $t+1$. Since $LCP[k] \geq t+1$, all these pairs are equal, so, the strings $w[r[i]..r[i]+t] = w[r[j]..r[j]+t]$ are equal, which contradicts $t = lcp(w[r[i]..n], w[r[j]..n])$. Thus, the fact that for all k, $LCP[k] > t$ is false, hence, there is at least one k for which $LCP[k] = t$. □

Proposition 9

1. *If* $lcp(w[r[i]..n], w[r[i+k]..n]) < lcp(w[r[i]..n], w[r[i+j]..n])$ *then* $j < k$.
2. *If* $lcp(w[r[i]..n], w[r[i-k]..n]) < lcp(w[r[i]..n], w[r[i-j]..n])$ *then* $j < k$.

Proof: For point 1, by Proposition 8, $lcp(w[r[i]..n], w[r[i+j]..n]) = \min\{LCP[t] : \ i \leq t < j\}$ and $lcp(w[r[i]..n], w[r[i+k]..n]) = \min\{LCP[t] : \ i \leq t < k\}$. If $j \geq k$, the set in the second equality is included in the set in the first one, so the minimum of the first set is not greater than the minimum of the second. Point 2 is analogous. □

3 The base algorithm

The two problems, supermaximal repeats in a set and exclusive supermaximal/maximal repeats in a string with respect to a set, are dual in the sense that the first requires a maximal string occurring in *each* string in the set, while the second requires a maximal string occurring in *no* string in the set. To solve both problems we use a base algorithm **longest_common_substring**, which takes two input strings w and s and outputs an integer array of length $|w|$. This array indicates for each suffix of w, the length of its longest prefix occurring in s.

Definition 10 *Given two strings w and s, for $1 \leq i \leq |w|$,*

$$m[i] = \max\{\ell : w[i..i+\ell-1] \text{ occurs in } s\}.$$

Proposition 11 *Given two strings w and s, for $1 \leq i \leq |w|$,*

$$m[i] = \max\{lcp(w[i..|w|], s[j..|s|]) : 1 \leq j \leq |s|\}.$$

Proof: Immediate from the definitions. □

We write $w\$s$ for the concatenation of w and s having a separator symbol \$ not in alphabet $\mathcal{A}$, and let r now be the the suffix array of $w\$s$. We use Proposition 9 applied to $w\$s$ to find the suffix of s having the longest common prefix with the suffix of w addressed by index i in r: it is addressed by closest index to i (upwards or downwards) that corresponds to a suffix of s.

Proposition 12 *Let r be the suffix array of $w\$s$. If i is an index of w in r then $m[r[i]] = \max(up_i, down_i)$ where*

> $up_i = lcp(w\$s[r[i]..|w|], w\$s[r[i-j]..|w|+|s|+1])$ *for j the lowest value such that $i-j$ is an index of s in r.*
> $down_i = lcp(w\$s[r[i]..|w|], w\$s[r[i+k]..|w|+|s|+1])$ *for k the lowest value such that $i+k$ is an index of s in r (let each of them be 0 when such j or k does not exist).*

Proof: Follows directly by Proposition 9 and Proposition 11. □

Algorithm 1 computes the array m by scanning the LCP array built from the suffix array of the string $w\$s$. Observe that the longest common substrings between w and s can be obtained by iterating over the array m, reporting the positions that these common substrings have in string w.

4 An algorithm for supermaximal repeats in a set

Since supermaximal repeats in a set of strings X (cf. Definition 2) are substrings occurring at least once in each $x \in X$, it is convenient to select a shortest string in X, called w from now on, and use it as *base string* for the algorithm. The idea is to iterate through each position of w checking whether a maximal repeat starts or not in the current position (see Proposition 6). From now on, we will then work on a set of the form $X \cup \{w\}$ where we assume the set X does not include w and that w is not longer than any of the strings on X. We will compute the supermaximal repeats in the set $X \cup \{w\}$.

Each supermaximal repeat is a prefix of a suffix of w, that also occurs in every $x \in X$ but such that any extension fails to occur in some $x \in X$. We use an array N of length $|w|$ to indicate, for each position i of w, the length of the longest prefix of the suffix $w[i..n]$ that also occurs in every $x \in X$.

Definition 13 *For $1 \leq i \leq |w|$, $N[i] = \max\{\ell :\ w[i...i+\ell-1]$ occurs in every $x \in X\}$.*

The pseudocode to construct this array is given in Algorithm 2, and we call it **minimum_length**. It calls the previously introduced **longest_common_substring** (Algorithm 1) for the base string w and each element of X, keeping always the smaller values. The whole N array is initialized with the length of each suffix of w, that is $|w| - i + 1$.

For notational convenience we introduce this definition.

Definition 14 *Let $w^{(i)} = w[r[i]...r[i] + N[r[i]] - 1]$.*

Clearly, all supermaximal repeats are among the $w^{(i)}$'s. In order to find the actual supermaximal repeats we will filter out the $w^{(i)}$'s that are not.

Algorithm 1 longest_common_substring (input: string w, string s, output: array m)

Initialize array $m[1..|w|]$ in 0
$r :=$ suffix array of $w\$s$
$LCP :=$ longest common prefix array of $w\$s$
–set $M[r[i]] = down_i$ by using Propositions 8 and 12.
for $i := 2$ to $|w\$s| - 1$ such that $r[i]$ is an index in w **do**
  if $r[i-1]$ is an index in w **then**
    $m[r[i]] := \min(m[r[i-1]], LCP[i-1])$
  else
    —$r[i-1]$ is an index in s—
    $m[r[i]] := LCP[i-1]$
  end if
end for
–calculate $acc = up_i$ as before and then update m
for $i := |w\$s| - 2$ to 1 such that $r[i]$ is an index in w **do**
  if $r[i+1]$ is an index in w **then**
    $acc := \min(acc, LCP[i]))$
  else
    —$r[i+1]$ is an index in s—
    $acc := LCP[i]$
  end if
  $m[r[i]] := \max(m[r[i]], acc)$
end for

Algorithm 2 minimum_length(input: string w, set of strings X, output: array N)

Initialize array $N[1..|w|]$ such that $N[i] = |w| - i + 1$
for each $x \in X$ **do**
  $m :=$ **longest_common_substring**(w, x)
  for $i := 1$ to $|w|$ **do**
    $N[i] := \min(N[i], m[i])$
  end for
end for

Observation 15

1. *For all i,* $w^{(i)}$ *occurs in every* $x \in X \cup \{w\}$.
2. *Every supermaximal repeat in* $X \cup \{w\}$ *is of the form* $w^{(i)}$ *for some i,* $1 \le i \le |w|$.

Proof: Immediate from Definitions 2, 13 and 14. □

Given the array N for the selected w, we define an equivalence relation on the indices of the suffix array r of w, such that contiguous indices with the same value in N define an equivalence class.

Definition 16 $i \equiv^{\dagger} j \;\Leftrightarrow\; \forall k \in [min(i,j), max(i,j)]\; N[r[i]] = N[r[k]]$.

Proposition 17 $\equiv^{\dagger}$ *is an equivalence relation on* $[1..n]$.

Proof: *Reflexivity* and *Symmetry*: Immediate from definition. *Transitivity*: Suppose $i \equiv^{\dagger} h$ and $h \equiv^{\dagger} j$. If $h \notin [i,j]$ then transitivity follows directly. Otherwise, without loss of generality assume $i \le h \le j$. Then $N[r[i]] = N[r[k]]\ \forall k \in [i,h]$ and $N[r[h]] = N[r[k]]\ \forall k \in [h,j]$. Since h is in both intervals, $N[r[i]] = N[r[h]] = N[r[k]]$, $\forall k \in [i,j]$. □

We now refine the relation $\equiv^\dagger$ and define the relation $\equiv^\ddagger$. It groups the indices that address consecutive suffixes of w in r whose longest common prefix is greater than the maximum allowed by the values of N. It will sub-partition each equivalence class of $\equiv^\dagger$ so that the longest common prefix of any two elements in the new partition is longer than the N value of the class, keeping all the elements in the new equivalence classes consecutive in r.

Definition 18 $i \equiv^\ddagger j \iff i \equiv^\dagger j$ *and* $\forall k \in [min(i,j), max(i,j)-1]\ LCP[k] \geq N[r[i]]$.

Proposition 19 $\equiv^\ddagger$ *is an equivalence relation on* $[1..|w|]$.

Proof: *Reflexivity* and *Symmetry*: Immediate from definition. *Transitivity*: Suppose $i \equiv^\ddagger h$ and $h \equiv^\ddagger j$. If $h \notin [i,j]$ then transitivity follows directly. Otherwise, without loss of generality assume $i \leq h \leq j$. This implies $i \equiv^\dagger h$ and $h \equiv^\dagger j$, therefore $i \equiv^\dagger j$. Thus, the value of $N[r[k]]$ is the same for every $k \in [i,j]$. Since $\forall k \in [i, h-1]$, $LCP[k] \geq N[r[k]]$ and $\forall k \in [h, j-1]$, $LCP[k] \geq N[r[k]]$, then $\forall k \in [i, h-1] \cup [h, j-1], LCP[k] \geq N[r[k]]$. We conclude $i \equiv^\ddagger j$. □

Observation 20 *Each equivalence class defined by* $\equiv^\dagger$ *or* $\equiv^\ddagger$ *is an integer interval.*

From the previous observation, we can name the equivalence classes as integer intervals.

Lemma 21 *For each equivalence class* $[i,j]$ *of* $\equiv^\ddagger$, $w^{(i)} = w^{(k)},\ \forall k \in [i,j]$.

Proof: If $i \equiv^\ddagger j$ then $N[r[i]] = N[r[k]], \forall k \in [i,j]$. Furthermore, if $i \equiv^\ddagger j$, $LCP[k] \geq N[r[i]]$, for $i \leq k < j$. Therefore, all the suffixes addressed by the interval $r[i...j]$ share, at least, their first $N[r[i]]$ symbols. □

We base our algorithm on the following characterization.

Definition 22 *A substring of* w *is supermaximal to the left (respectively right) iff it occurs in every* $x \in X \cup \{w\}$ *and none of its extensions to the left (respectively right) occur in every* $x \in X \cup \{w\}$.

Observation 23 *A substring of* w *is supermaximal iff it is supermaximal to the left and supermaximal to the right.*

Lemma 24 *For an equivalence class* $[i,j]$ *defined by* $\equiv^\ddagger$, $w^{(i)}$ *is supermaximal to the right if and only if the following two conditions hold:*

1. $i = 1$ *or* $LCP[i-1] < N[r[i]]$
2. $j = n$ *or* $LCP[j] < N[r[j]]$

Proof: If $w^{(i)}$ is supermaximal to the right, it is not a prefix of any other string $w^{(k)}$ with $k \notin [i,j]$. In particular, it is not a proper prefix of $w^{(i-1)}$ nor $w^{(j+1)}$ (when they exist). It is also not equal to $w^{(i-1)}$ nor $w^{(j+1)}$, because otherwise they would be in the same equivalence class. Therefore, the length of the common prefix of $w^{(i)}$ with $w^{(i-1)}$ or $w^{(j+1)}$ is strictly less than its length.

Now, if $w^{(i)}$ is not supermaximal to the right, then there is some extension to the right. This extension is either supermaximal, or can be extended to a supermaximal substring. Therefore, there is some supermaximal repeat that extends $w^{(i)}$ to the right. By Observation 15, there is some k such that $w^{(k)}$ extends $w^{(i)}$ to the right, so its length $N[r[k]] > N[r[i]]$. By Lemma 21, $k \notin [i,j]$. Since $w^{(k)}$ extends $w^{(i)}$, $lcp(w^{(k)}, w^{(i)}) = N[r[i]]$. There are two cases:

Case $k < i$. Since $k < i$, we have $i \neq 1$. Using Proposition 8 we get $N[r[i]] = lcp(w^{(k)}, w^{(i)})$ and $lcp(w^{(k)}, w^{(i)}) \leq lcp(w[r[k]..|w|], w[r[i]..|w|]) = \min\{LCP[l] : k \leq l < i\}$, so $LCP[i-1] \geq N[r[i]]$. This case, then, implies the negation of the first condition.

Case $k > j$ analogously implies the negation of the second condition. □

Lemma 25 *For an equivalence class* $[i,j]$ *defined by* $\equiv^\ddagger$, *let* $w^{(i)}$ *be supermaximal to the right. Then,* $w^{(i)}$ *is also supermaximal to the left if and only if* $\forall k \in [i,j]$, $r[k] = 1$ *or* $N[r[k]-1] \leq N[r[k]]$.

Proof: Assume $w^{(i)}$ supermaximal to the right. There is no extension to the right of $w^{(i)}$ that occurs in every the element of X. Then, there is no other equivalence class $[i', j']$ such that $w^{(i)}$ is a proper prefix of $w^{(i')}$. Thus, there is also no other class $[i', j']$ such that $w^{(i)} = w^{(i')}$, because those two classes can not be neighbors (otherwise they would be the same class) and can not be separated because of the lexicographic ordering and the previous claim. Then, all the occurrences of $w^{(i)}$ in w are addressed by the indices in r in the equivalence class $[i, j]$.

For the left to right implication assume there is k in $[i, j]$ such that $r[k] > 1$ and $N[r[k] - 1] > N[r[k]]$. By Lemma 21, $w^{(k)} = w^{(i)}$. Let us call u the substring of w starting at position $r[k] - 1$ with length $N[r[k] - 1]$. Since $N[r[k] - 1] \geq N[r[k]] + 1$, u extends $w^{(k)}$ to the left. Since u occurs in every $x \in X \cup \{w\}$ (by definition of N), $w^{(k)}$ is not supermaximal to the left. Conversely, assume $w^{(i)}$ is not supermaximal to the left. Then, there is an extension of one symbol to the left, that occurs in every $x \in X$. Let us call that extension u and assume it occurs at position t in w. Then, $w^{(i)}$ occurs at position $t + 1$, and therefore, as proved in the first paragraph, there is $k \in [i, j]$ such that $t + 1 = r[k]$, or $t = r[k] - 1$, so $r[k] > 1$. Since u occurs in every $x \in X$, $N[t] \geq |u| = N[r[k]] + 1$, so $N[t] = N[r[k] - 1] > N[r[k]]$. □

Algorithm 3 mrset(input: set of strings Y)

w:= a shortest $y \in Y$
$X := Y \setminus \{w\}$ —remove w from Y—
$N :=$ **minimum_length**(w, X)
$r :=$ suffix array of w
$LCP :=$ longest common prefix array of r
for each equivalence class $[i, j]$ defined by $\equiv^{\ddagger}$ **do**
  if ($i = 1$ or $LCP[i - 1] < N[r[i]]$) and
    ($j = n$ or $LCP[j] < N[r[j]]$) **then**
    if for each k in $[i, j]$, ($r[k] = 1$ or $N[r[k] - 1] \leq N[r[k]]$) **then**
      report $w^{(i)}$
    end if
  end if
end for

Theorem 26 *Algorithm 3, called* **mrset**, *computes the supermaximal repeats in a given set.*

Proof: Algorithm 3 iterates through all equivalence classes of $\equiv^{\ddagger}$ (Definition 18). The first conditional filters out exactly the substrings that are not maximal to the right, and the second the ones that are not maximal to the left, according to Lemmas 24 and 25. Since all supermaximal repeats in a set must be of the form $w^{(i)}$ (by Observation 15), the algorithm reports all possible repeats, as wanted. □

5 An algorithm for exclusive supermaximal/maximal repeats

To obtain the exclusive maximal or supermaximal repeats in a given string w with respect to a set X, we compute the maximal or supermaximal repeats in w and filter out those that occur in some $x \in X$ (cf. Definitions 1 and 2). We use an array M to store, for each position i in w, the length of the longest prefix of $w[i..n]$ that occurs in some $x \in X$. That is, for each i, $w[i..i + M[i] - 1]$ occurs in some $x \in X$, but $w[i..i + M[i]]$ does not occur in any $x \in X$.

Definition 27 *For* $1 \leq i \leq |w|$, $M[i] = \max\{\ell : w[i..i + \ell - 1]$ *occurs in some* $x \in X\}$.

Algorithm 4, called **maximum_length**, gives the pseudocode of how to construct the array M using **longest_common_substring** (Algorithm 1) and updating it for each element of X, keeping always the larger values. It is to dual to **minimal_length** (Algorithm 2), because it replaces $\min$ with $\max$, and the whole array M is initialized in 0.

Proposition 28 *Let r be the suffix array of a string w, and let M be the array as in Definition 27 for a given set X. The following conditions are equivalent:*

Algorithm 4 maximum_length(input: string w, set of strings X, output: array M)

```
Initialize array M[1..|w|] in 0
for each x ∈ X do
   m := longest_common_substring(w, x)
   for i := 1 to |w| do
      M[i] := max(M[i], m[i])
   end for
end for
```

1. *A string s is an exclusive maximal (respectively supermaximal) repeat of w with respect to set X.*
2. *s is a maximal (respectively supermaximal) repeat in w with k occurrences, for some $k \geq 2$, in positions $r[i], ..., r[i+k-1]$, and the length of s is greater than each of $M[r[i]], .., M[r[i+k-1]]$.*
3. *s is a maximal (respectively supermaximal) repeat in w, one of its occurrences is in position $r[i]$ and the length of s is greater than $M[r[i]]$.*

Proof: The equivalence between 1 and 2 follows from the definition of M and the already mentioned properties of the occurrences of a maximal (respectively supermaximal) repeat. It is trivial that 2 implies 3, because it is a particular case. We can also see that 3 implies 2, because if $M[r[i]] < |s|$, then s does not occur in any $x \in X$, and then $M[k] < |s|$ for any position k at which s occurs in w. □

Our algorithms **findmaxr** and **findsmaxr**, presented in [4, 3], report each repeat (respectively maximal and supermaximal) in a concise way, indicating the index in r of its first occurrence i, the number of repetitions k and the length of the repeat ℓ. Given the aforementioned results, to filter out the non exclusive repeats, all that is needed is to report the repeats such that $M[i] < \ell$.

Correctness of the two modified algorithms follow directly from the correctness of the cited algorithms and the results above.

6 Complexity of the algorithms

As it is usual in the literature on algorithms, we express the time and space complexity assuming integer values can be stored in a unit, and integer additions and multiplications can be done in $\mathcal{O}(1)$. These assumptions make sense because the integer values involved in the algorithms fit into the processor word size for practical cases. Although our algorithms are scalable for any input size, the derived complexity bounds are guaranteed only if the input size remains under the machine addressable size. Otherwise, the classical logarithmic complexity charge for each integer operation becomes mandatory.

We now prove that the algorithms presented in the previous sections, supermaximal repeats in a set and exclusive maximal/supermaximal repeats in a string with respect to a set, require $\mathcal{O}(n \log m)$ time and $\mathcal{O}(m)$ memory, where n is the sum of the all strings' length in the input, and m is the length of the longest input string.

To bound the time complexity of our algorithms we use that the output can be represented in a concise way, as follows:

Proposition 29 *Given a set of strings X.*

1. *The set of all supermaximal repeats in X is representable in space $\mathcal{O}(\min\{|x| : x \in X\})$.*
2. *The set of all exclusive maximal or supermaximal repeats in w with respect to X and all their occurrences is representable in space $\mathcal{O}(|w|)$.*

Proof: We argue first for Point 2. As stated in Proposition 6, exclusive maximal/supermaximal repeats in w are a subset of the maximal/supermaximal repeats in w, which are representable in space $\mathcal{O}(|w|)$, cf. [4]. For the sake of completeness in the complexity argument we include the proof.

Each reported maximal repeat and all its occurrences can be represented with three unsigned integers: an index i in the suffix array r, a length ℓ, and the number of occurrences k. The reported maximal repeat is the prefix of length ℓ of the suffix at position $r[i]$. Its k occurrences are respectively in positions $r[i]$,..,$r[i+k-1]$. Each of these integers is at most $|w|$ (where $|w|$ is at most the maximum addressable memory) and we need at most $|w|$ of them (Proposition 6). Assuming that these integer values can be stored in fixed number of bits, this output requires size $\mathcal{O}(|w|)$. Finally, we need to store the suffix array r, which contains $|w|$ integer values that are a permutation of $1..|w|$, so it also requires $\mathcal{O}(|w|)$. The input w also takes $\mathcal{O}(|w|)$ space, since each symbol in $\mathcal{A}$ also takes $\mathcal{O}(1)$ because $|\mathcal{A}| \leq |w|$.

Point 1 is analogous, each repeat is reported by giving an interval in the suffix array (two integers) and a length. Notice that to report the repeats in a set X our algorithm **mrset** uses a witness string in X, and only reports the lengths and positions of maximal repeats in it, instead of reporting also the occurrences in each of the strings in X. □

Of course, if instead of charging a fixed number of bits to store an integer, we count the length of its bit representation, the total needed output space to report all maximal repeats and occurrences in a given input string of length n bits becomes $\mathcal{O}(n \log n)$. The input in this case would have an $\mathcal{O}(n \log |\mathcal{A}|)$ bound, which has $\mathcal{O}(n \log n)$ as worst case, but probably takes a lot less because alphabet sizes are usually small compared with n.

Proposition 30 *For two given strings w and s,* **longest_common_substring** *(Algorithm 1) requires time $\mathcal{O}(n \log n)$ and space $\mathcal{O}(n)$, where $n = |w| + |s|$. More specifically, the total memory required is bounded by $(|w| + |s| + 1)(2\ word\_size + \log |\mathcal{A}|) + |w| word\_size + \mathcal{O}(1)$ bits.*

Proof: Consider Algorithm 1. Its input is stored in exactly $(|w| + |s|) \log |\mathcal{A}|$ bits. The suffix array r of $w\$s$, and its LCP array require $|w\$s|$ word_size memory each. The suffix array takes $\mathcal{O}(n \log n)$ to be calculated, and the LCP array takes $\mathcal{O}(n)$. The output array has length $|w|$. No other data structures are used, and the auxiliary variables are accounted for in the $\mathcal{O}(1)$ term. In the *for* loops the condition of $r[i]$ being an index in s or w can be checked in $\mathcal{O}(1)$ by comparing $r[i]$ with $|w|$. Hence, the total time of Algorithm 1 is $\mathcal{O}(n \log n)$. □

Proposition 31 *Let w be a string and let X be a set of strings.*

1. *If $|w| < \min\{|x| : x \in X\}$,* ***minimum_length*** *(Algorithm 2) can be implemented to construct array N in time $\mathcal{O}(n \log m)$, and requires $(|w| + m)\ (2\ word\_size + \log |\mathcal{A}|) + 2\ |w|\ word\_size + \mathcal{O}(1)$ bits of memory, where $n = \sum_{x \in X \cup \{w\}} |x|$ and $m = \max\{|x| : x \in X \cup \{w\}\}$.*
2. ***maximum_length*** *can be implemented to construct array M with the same time and memory bounds.*

Proof: The output array N or M and the auxiliary array m require, each, $|w|$ word_size bits. For each $x \in X$, the computation of **longest_common_substring** temporarily requires constant space for local variables plus $(|w| + |x|)(2\ \text{word\_size} + \log |\mathcal{A}|)$ bits of memory. Thus, total memory space required is $(m + |w|)\ (2\ \text{word\_size} + \log |\mathcal{A}|) + 2\ |w|\ \text{word\_size} + \mathcal{O}(1)$ bits, where $m = \max\{|x| : x \in X \cup \{w\}\}$. The upper bound for the needed time is the sum of the time required by Algorithm 1 with arguments w, and x, for each $x \in X$. This is $\mathcal{O}(\sum_{x \in X}(|w| + |x|) \log(|w| + |x|))$. For **minimum_length**, the additional hypothesis $|w| < \min\{|x| : x \in X\}$ implies a time bound of $\mathcal{O}(n \log m)$. For **maximum_length**, on the other hand, while $|w|$ could be greater than $|x|$ for some $x \in X$, we can ensure the given bounds by considering a variant of the set X: Group the elements in X in a greedy way to define a new set Y composed of the concatenation (with a separator) of as many as possible elements of X up to length at most m. It is easy to see that at most one element of Y will have a length lower than $m/2$ (otherwise, we can still concatenate those elements). It is also clear that occurring in X is equivalent to occurring in Y, so the problems of computing the algorithms with respect to X or to Y are equivalent. Finally, $\sum_{y \in Y \cup \{w\}}(|w| + |y|) \log(|w| + |y|)) = \mathcal{O}(|Y| m \log m) = \mathcal{O}(\frac{n}{m/2} m \log m) = \mathcal{O}(n \log m)$. □

Theorem 32 **mrset** *(Algorithm 3) computes the supermaximal repeats in a set Y in time $\mathcal{O}(n \log m)$ and $\mathcal{O}(m)$ memory, where $n = \sum_{y \in Y} |y|$ and $m = \max\{|y| : y \in Y\}$. More precisely, it requires $(m + |w|)(2\ word\_size + \log |\mathcal{A}|) + 2\ \min\{|y| : y \in Y\} word\_size + \mathcal{O}(1)$ bits of memory, where w is a shortest string in the set.*

Proof: Consider Algorithm 3. Let w be the smallest element in Y and let $X = Y \setminus \{w\}$. The total time is the sum of the time to compute the three data structures N, r and LCP, plus the time needed by the *for* loop. By Proposition 31, **minimum_length**(w, X) computes N in time $\mathcal{O}(n \log m)$. This subsumes the $\mathcal{O}(|w| \log |w|)$ time needed for the suffix array r plus the linear time for the LCP. Since equivalence classes in $\equiv^{\ddagger}$ are intervals, cf. Observation 20, the *for* loop can iterate through the indices in increasing order, and check the partitions in linear time. The check for each k inside the *for* loop has also one check for each index, so it is still linear overall. All the other instructions in the loop take time $\mathcal{O}(1)$, hence, the *for* loop requires only time linear in $|w|$. The needed memory is the maximum between: the space for the data structures N, r, LCP and the input w, and the memory required by **minimum_length**(w, X). This, using Proposition 31, is constant space plus the maximum between $|w|(3 \text{ word\_size} + \log |\mathcal{A}|)$ and $(m + |w|)(2 \text{ word\_size} + \log |\mathcal{A}|) + 2|w| \text{ word\_size}$. The latter is clearly larger, and it is $\mathcal{O}(m)$. □

Theorem 33 *To compute the exclusive supermaximal/maximal repeats in a string w with respect to a set X requires $\mathcal{O}(n \log m)$ time and $\mathcal{O}(m)$ memory, where $n = \sum_{x \in X \cup \{w\}} |x|$ is the total length of the input and $m = \max\{|x| : x \in X \cup \{w\}\}$ is the maximum length of an input string. More precisely, it requires $(m + |w|)\ (2\ \textit{word\_size} + \log |\mathcal{A}|) + 2\ |w|\ \textit{word\_size} + \mathcal{O}(1)$ bits of memory.*

Proof: The precomputation of array M is done in **maximum_length** takes $\mathcal{O}(n \log m)$ time and uses $\mathcal{O}(m)$ memory, as shown in Proposition 31. Let's see that this dominates the total time and memory required to compute the exclusive supermaximal/maximal repeats in a w with respect to X. An $\mathcal{O}(|w| \log |w|)$ time bound to compute the maximal or supermaximal repeats in w is proved in [4] for algorithms **findmaxr** and **findsmaxr**, and both algorithms have an $\mathcal{O}(|w|)$ memory bound. Only $\mathcal{O}(1)$ time must be added to check whether each repeat must be filtered out; thus, the overall time bound remains $\mathcal{O}(n \log m)$. Over the execution of **findmaxr** or **findsmaxr** an $\mathcal{O}(|w|)$ memory must be added to store the array M, so the memory bound remains $\mathcal{O}(m)$. The actual memory required is the maximum between the memory used by **findmaxr** or **findsmaxr** plus an extra $|w|$ word_size for the array M, and the memory needed to calculate M. The requirements for each part are:

findmaxr:	$|w|(4word\_size + \log |\mathcal{A}| + 2) + \mathcal{O}(1)$,
findsmaxr:	$|w|(3word\_size + \log |\mathcal{A}| + 2) + |\mathcal{A}| + \mathcal{O}(1)$,
maximum_length:	$(m + |w|)\ (2 \text{ word\_size} + \log |\mathcal{A}|) + 2\ |w| \text{ word\_size} + \mathcal{O}(1)$,

We conclude the actual total memory requirement is that of **maximum_length**. □

7 Implementation and Experiments

We implemented all algorithms in C (ANSI C99), for a 32 or 64 bits machine. The complete source code of the algorithms and an example tool to run them can be downloaded from
`http://www.dc.uba.ar/people/profesores/becher/software/findrepset.tar.bz2`.

We tested this implementation on large inputs, using an Intel® Core$^{\text{TM}}$2 Duo E6300 (only one core), running at 1.86GHz with 8GB RAM (DDR2-800) under Ubuntu linux for 64 bits. The programs were compiled with the GCC compiler version 4.2.4, with option -O2 for normal optimization.

We performed tests on the input files described in Table 1. We used the Canterbury corpus [(i)] and the human genome NCBI 36.49 FASTA files [(ii)] with headers, enters, and unknown base marks (letter N) removed.

Each run consists on a set with an element selected to be the base. In the case of exclusive maximal or supermaximal repeats, the base is the string in which the repeats are searched. For supermaximal repeats in a set, any element can be selected as the base string and the same result is obtained, but size of the chosen element greatly affects the running time, see the introduction in section 4 and the proof of Theorem 32. We tried both the largest and smallest elements as bases to compare. The reported times are user times, counting only the time consumed by the algorithm, not including the needed time the load the input from disk. In Table 2 we describe each run and the aggregated time for each part of the process: suffix-array constructions

[(i)] `http://corpus.canterbury.ac.nz/descriptions/#cantrbry`
[(ii)] `ftp://ftp.ensembl.org/pub/release-49/fasta/homo_sapiens/dna/`

Tab. 1: Input files

Set	Description
as	2 files with letter ‘a’ repeated 2 million and 65536 times
txts-big	2 largest files in the txts set form large from Canterbury corpus
txts	4 text files containing English texts from Canterbury corpus
linux-hs	7043 .h files in the Linux Kernel 2.6.31 tar file
linux-cs	9985 .c files in the Linux Kernel 2.6.31 tar file
HS-genome-big	3 FASTA files of human chromosomes 1, 2 and 3 from NCBI 36.49
HS-genome	24 FASTA files of all human chromosomes NCBI 36.49

Input	Set	Set total size (bytes)	Base	Base size (bytes)
1	as	2 065 536	a64K.txt	65536
2	as	2 065 536	a2M.txt	2 000 000
3	txts-big	6 520 792	world192.txt	2 473 400
4	txts-big	6 520 792	bible.txt	4 047 392
5	txts	6 798 060	asyoulik.txt	125 179
6	txts	6 798 060	bible.txt	4 047 392
7	linux-hs	54 582 944	genapic.h	22
8	linux-hs	54 582 944	me4000_firmware.h	781 415
9	linux-cs	197 594 330	regs.c	32
10	linux-cs	197 594 330	nls_cp949.c	875 265
11	HS-genome-big	2 975 638 422	chr3.actgn	200 851 322
12	HS-genome-big	2 975 638 422	chr1.actgn	252 811 345
13	HS-genome	3 139 901 384	chr21.actgn	48 817 465
14	HS-genome	3 139 901 384	chr1.actgn	252 811 345

Tab. 2: Running times in seconds of each process

Input	suffix array	LCP array	max/min array	filtered findmaxr	filtered findsmaxr	mrset
1	3.02	0.06	0.01	0.03	0.00	0.00
2	5.33	0.11	0.04	1.22	0.01	0.01
3	8.73	0.81	0.23	1.10	0.13	0.08
4	8.90	0.83	0.29	1.66	0.22	0.12
5	5.45	0.53	0.05	0.03	0.00	0.00
6	16.88	1.59	0.91	1.77	0.24	0.13
7	18.33	2.01	0.20	0.00	0.00	0.00
8	5 421.00	352.39	294.96	0.29	0.02	0.02
9	59.66	6.87	0.66	0.00	0.00	0.00
10	4 987.65	347.68	275.65	0.23	0.02	0.01
11	2 844.43	204.80	85.64	114.28	22.45	19.11
12	3 224.98	222.79	106.63	154.73	26.92	24.34
13	9 552.86	676.26	190.50	24.40	4.00	2.58
14	23 738.29	1 608.39	1 085.60	151.85	26.84	21.86

(sum of all needed constructions), LCP array calculation, minimum/maximum array calculation (both take the same time), **findmaxr** and **findsmaxr** each with the filter **maximum_length**, and **mrset**.

The experiments confirm the theoretical complexity bounds. The times in first three columns of Table 2 are bounded by the total size of the set, while the last three are bounded by the size of the base (considerably smaller in most cases). The first column, suffix array constructions, accounts for most of the total time because it is the only superlinear time. Of the last three columns, **findmaxr** is the slowest, also due to the extra $\log$ factor. As a final comment, note in the linux-cs and linux-hs set, the time increase when choosing a large file as the base. This difference in practice illustrates the need to choose a short base (for **mrset**) or to do the concatenation trick in the complexity bounds proof (for the **maximum_length** filter).

Acknowledgments

We thank the anonymous referees for their comments and suggestions. The group has received support from Biosidus and IBM Argentina.

References

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86, 2004.

[2] Maxim Babenko and Tatiana Starikovskaya. Computing longest common substrings via suffix arrays. In *CSR 2008, LNCS 5010*, page 6475, Heidelberg, 2008. Springer-Verlag Berlin.

[3] Verónica Becher, Alejandro Deymonnaz, and Pablo Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics (Oxford, England)*, 25(14), July 2009.

[4] Verónica Becher, Alejandro Deymonnaz, and Pablo Heiber. Efficient repeat finding via suffix arrays. manuscript, arXiv:1304.0528, 2012.

[5] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[6] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc.12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, London, UK, 2001. Springer-Verlag.

[7] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[8] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. SODA ’90: Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 319–327, San Francisco, 1990.

[9] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.