xkcd-repeats: a taxonomy of repeats defined by their context diversity

Matthias Gallé[a,*], Matías Tealdi[b]

[a]*Naver Labs Europe, France*
[b]*Amazon*

Abstract

The context in which a substring appears is an important notion to identify – for example – its semantic meaning. However, existing definitions from stringology fail to model the context explicitly. We introduce here *xkcd*-repeats, a new family of repeats characterized by the number of different symbols at the left and right of their occurrences. These repeats include as special extreme cases the well-known classes of maximal and super-maximal repeats.

We give sufficient and necessary condition to bound their number linearly in the size of the original string, and show an optimal algorithm that computes them in linear time – given a suffix array –, independent of the size of the alphabet, as well as two other algorithms that are faster in practice.

We extend this in two ways: first we show how to generalize the notion of context. Instead of reducing it to the single symbol before and after an occurrence, we propose the notion of unbounded context, while keeping linear representation and computing time. Secondly, we provide a general framework which allows to compute these (and other) repeats incrementally; opening the door to streaming application.

Keywords: repeats, stringology, text algorithms, suffix array

1. Introduction

Inferring *constituents* is a basic step for many applications involving textual documents. Constituents are the semantic blocks that define the meaning of a document, and can be single words, multi-words expressions or even partial (or several) sentences. They can be used to represent the document, and an accurate description is crucial to tasks such as classification, clustering, topic detection or knowledge extraction. It has long been known that the importance of a constituent does not rely only on its intrinsic properties (like frequency, lengths

*Corresponding author
Email addresses: `matthias.galle@naverlabs.com` (Matthias Gallé), `matiastealdi@gmail.com` (Matías Tealdi)

or composition) but also on the *context* it appears in. J.R. Frith famously said "You shall know a word by the company it keeps" and the "distributional" approach has been successfully used in natural-language applications [25], with a renewed interest in recent years thanks to the success of word embeddings [17, 21].

These constituents are also crucial in the more fundamental task of inferring the structure of a document. In grammatical inference for instance – where it is supposed that the document samples were generated by a formal grammar – prior to detecting how different rules are related to each other, one has to decide which of the substrings of the document may or may not correspond to the same constituent. A crucial step for this is the notion of the context in which a substring appears in. In the most basic setting this context is just the character to the left and to the right of the occurrence of a given word. ADIOS with its MEX procedure for instance [26] uses as fundamental signal the fraction of different context a substring appears in to decide on the set of constituents. Strongly related to the idea of context of a constituent is the idea of Zellig Harris' substitutability theory, which got implemented in diverse ways: for example in ABL [28] or through the mutual information criterion of Clark [8, 9]. Selecting which are the right substrings to consider, prior to decide how they relate to each other hierarchically, is also important in the associated Smallest Grammar Problem [6].

Unfortunately, existing notions from stringology offer very limited links to these theories. Existing classes of words only tangentially take into account the notion of context; and applied algorithms, like those cited above, rely on trivial and brute-force algorithms to detect them.

We present here a new class of repeats, named *xkcd*-repeats, that is explicitly defined by the number of different context a substring appears in (Sect. 3). These repeats define a family of classes where maximal and super-maximal repeats are extreme cases. We give bounds on their number with respect to the size of the string and give three algorithms to compute them relying on different ideas. These algorithms run in $\mathcal{O}(|\Sigma|n)$, $\mathcal{O}(n \log n / \log\log n)$ and $\mathcal{O}(n)$ time respectively, and in Sect. 4 we compare empirically their efficiency on different datasets.

In this vanilla definition of *xkcd*-repeats, the context is considered as the immediate symbols to the left and right of the occurrences of a substring. While aligned with the definitions of maximal and super-maximal repeats, this seems an unnatural constraint in real applications. All repeats will have their context size bounded by $|\Sigma|$, regardless of the number of occurrences, which seems particularly ill-suited in domains where the alphabet size is small (like DNA strings). In a first extension (Sect. 5) we expand the notion of context to include all substring to the left/right and show how to compute those *unbounded xkcd repeats* ($\infty-$xkcd) still in linear time with respect to the size of the original string.

An independent second extension (Sect. 6) concerns a general framework to compute *xkcd* and other repeats incrementally when documents arrive one after the other. For this, we define the notion of *cover* of a document d with respect to a collection S as a certain subset of nodes of the suffix tree of $S \cup \{d\}$ which contains all substrings whose condition may change due to the inclusion of d. We propose algorithms that solve the incremental computation problem by traversing this *cover* set, and we study theoretically and empirically the size of this set with respect to the minimal (optimal) set of changes to be performed.

A preliminary version of this paper was presented as [13].

2. Definitions

A *string* s is a concatenation of symbols $s[1] \dots s[n]$, with $s[i] \in \Sigma$, the *alphabet*. The *length* of s, $|s|$ is the numbers of symbols, which we will generally denote by n. When necessary we will suppose that s starts and ends with different unique symbols ($s[0] = \$_1, s[n+1] = \$_2, \$_1 \neq \$_2$ and $\$_1, \$_2 \notin \Sigma$). Another string $\omega \in \Sigma^*$ is said to *occur* in s at position i if $\omega[j] = s[i+j-1]$ for $j = 1 \dots |\omega|$. The set of occurrences of ω in s is denoted by $occ_s(\omega)$ (or just $occ(\omega)$ if s is clear from the context), and is the set of all starting positions of ω in s. If $|occ_s(\omega)| \geq 2$, ω is called a *repeat* of s and $\mathcal{R}(s)$ is the set of all repeats of the string s.

The size of the left (right) context of a word ω in s is defined as the number of different symbols appearing to the left (right) of all occurrences of ω: $lc_s(\omega) = |\{s[i-1] : i \in occ(\omega)\}|$ ($rc_s(\omega) = |\{s[i+|\omega|] : i \in occ(\omega)\}|$).

Most of the algorithms presented here use the *suffix array* and other associated data structures:

2.1. Suffix Array

The suffix array is part of the suffix-tree data structure family. It consists of a lexicographically ordered array of all suffixes of the input string. The suffixes themselves are not stored but instead their starting positions.

Definition 1 (Suffix Array). *Consider a string s of length n over an alphabet Σ with an order $\prec$. The lexicographical extension to Σ^* will also be denoted by $\prec$, and $n = |s|$.*

The suffix array, denoted by sa, is an integer array such that $sa[i] = j$ iff the suffix $s[j..n]$ has rank i in the sorted array of all suffixes of s. To ease computation, we normally assume that s ends with $\$ \notin \Sigma$, which is smaller than all values over Σ. The suffix $\$$ is therefore the smallest of all suffixes, and $sa[1] = n+1$.

Usually, the suffix array is used conjointly with an array called lcp, that gives the length of the longest common prefix between two suffixes whose starting positions are adjacent in sa. Formally,

$$\begin{aligned} lcp[1] = & \quad 0 \\ \forall\ i \in [2, n+1] : lcp[i] = & \quad j \textit{ iff } j \textit{ is the largest value such that} \\ & \quad s[sa[i-1]..sa[i-1]+j] = s[sa[i]..sa[i]+j] \end{aligned}$$

Suffix and lcp arrays can be constructed in linear time [22], and we will refer to both arrays together as *enhanced suffix arrays*.

3. Context-diverse repeats

Indexing the set $\mathcal{R}(s)$ permits to analyze potential constituents of the string or to perform other indexing or counting operations. However, the total set of repeats is highly redundant and can grow quadratically with the size of the string ($|\mathcal{R}(s)| \in \mathcal{O}(|s|^2)$). These "maps bigger than the empire" (as Apostolico referred to this [2], borrowing an expression from Borges), lead to the popular use of family of equivalent classes of repeats. The representatives of these classes should ideally correspond to the "interesting" repeats. We detail below three of these families:

Maximal repeat: a repeat is said to be maximal, if it cannot be extended without losing support. Formally: ω is a maximal repeat iff there is no other repeat ω' such that ω occurs in ω' and $|occ(\omega)| = |occ(\omega')|$. The number of maximal repeats grows linearly with the size of the string, although their total number of occurrences can still be quadratic [11].

Largest-maximal repeats (or near-supermaximal repeat [14, 19]): a repeat is said to be largest-maximal if it has at least one context that is unique. Equivalently, this means that at least one occurrence is not covered by another repeat. Formally, ω is largest-maximal iff there do not exist other repeats $\omega_1, \ldots \omega_k$ such that ω occurs in all of them and $pos(\omega) \subseteq \bigcup_{i=1}^{k} pos(\omega_i)$, where $pos(\omega) = \bigcup_{i \in occ(\omega)} \{i, \ldots, i + |\omega| - 1\}$, the set of positions that ω covers over the string. Largest-maximal repeats are a subset of maximal repeats and therefore linear. The total number of occurrences grows in the worst case at least as $\Omega(n^{\frac{3}{2}})$, although a tight bound is unknown [11].

Super-maximal repeat: A repeat is said to be super-maximal if it does not occur in any other repeat. The total number of occurrences of super-maximal repeats is linear [14].

As an example, consider the string *dabWabXacYacZdab*. The only super-maximal repeats here are *ac* and *dab*. In addition to these *ab* is also a largest-maximal repeat because it appears once without any other repeat covering it (its second occurrence). Finally, *a* is also a maximal repeat as there is no other repeat that contains it and has the same number of occurrences.

The following two lemmas provide the fundamental motivation of this paper, as they show how to characterize maximal and super-maximal repeats by the number of their left and right context.

Lemma 1. *A repeat ω is maximal in s iff $lc(\omega) \geq 2$ and $rc(\omega) \geq 2$.*

Proof. In order for ω to be maximal, there can not be a single other repeat that contains it with the same frequency. Assume $lc(\omega) = 1$ and that unique context is a. In that case $a\omega$ would occur $occ(\omega)$ times, and ω would not be maximal. The same of course applies for rc.

On the other side, suppose $lc(\omega) \geq 2$. This implies that there is no other single repeat of the form $a\omega$ that occurs the same number of times. ω is therefore left-maximal, and the same argument can be used for right-maximality. □

Lemma 2. *A repeat ω is super-maximal in s iff $lc(\omega) = rc(\omega) = |occ(\omega)|$.*

Proof. Of course $lc(\omega) \leq |occ(\omega)|$. Assume that it is not equal. In that case there exists a left-context a such that $a\omega$ occurs at least twice and is therefore a repeat containing ω, contradicting the super-maximality of ω.

On the other side, if $lc(\omega) = |occ(\omega)|$, then all occurrences have a unique (left) context. Any substring $a\omega$ is therefore not a repeat, making ω super-maximal. □

Context-diverse (*xkcd*) repeats fill the whole range of these two extremes, by permitting to vary the values of these contexts:

Definition 2. *$\omega \in \mathcal{R}(s)$ is said to be x–left-context-diverse (xlcd) in s if $lc_s(\omega) \geq x$. It is said to be k–right-context-diverse (krcd) in s if $rc_s(\omega) \geq k$.*

Finally, ω is a $\langle x, k\rangle$–context-diverse (xkcd) in s if it is xlcd and krcd.

The following proposition summarizes both Lemma 1 and 2:

Proposition 1.

1. *ω is a maximal repeat in s iff it is $\langle 2, 2\rangle$–context-diverse.*
2. *ω is a super-maximal repeat in s iff it is $\langle |occ(\omega)|, |occ(\omega)|\rangle$–context-diverse.*

The following Corollary follows from here, and from the linearity of right- and left-maximal repeats [14]:

Corollary 1. *The number of xkcd-repeats is $\mathcal{O}(n)$ iff $\max(x, k) \geq 2$. It is $\Theta(n^2)$ otherwise.*

If $\max(x, k) < 2$ (this is, $x = k = 1$), then we are talking about all normal repeats, and their number is bounded by $\Theta(n^2)$. For the remainder of this paper, we will always consider $x, k \geq 2$. The case were one of the values is 1 would require special treatment, and – in many of the algorithms analyzed in the following – would change the time complexity. The repeats that we miss by this restrictions are not maximal, and therefore their interest is minor in most of the applications.

A notable exception of a class of repeats that cannot be captured by this notion of *xkcd*-repeats are *largest-maximal repeats*. For a repeat to be largest-maximal, it has to have at least one occurrence with a right-(left-)context different from all right-(left-) contexts of remaining occurrences. Such *context-uniqueness* cannot be captured with the rc and lc functions.

3.1. Motivation

We tested the capacity of these repeats to capture semantic blocks in real texts. For this we use the Penntree-bank collection[1], a collection of English sentences annotated with parentheses, denoting the phrase-structure of the sentence; that is, how the underlying constituent grammar generated it. In the following experiments, we used the Part-of-Speech (POS)–tagged sentence (there were 36 different POS-tags), which reduce the vocabulary by clustering them (nouns, verbs, etc). We filtered out parentheses spanning single words and whole sentences. There remained 697 080 constituents, corresponding to 325 069 different strings. Of these, only 17% are repeated substrings but they make up 61% of the total constituents. This pre-processing is standard practice in the community, as it is to evaluate the algorithms with the unlabeled F_1 measure [15].

Because we do not concentrate here on the task of deciding at which positions a substring becomes constituent, but only if a substring is used or not as constituent in some moment, we use a modified version of the F_1 measure that takes into account the fact that some substrings appear very often as constituents. For precision we use the standard notion: the percentage of retrieved substrings that are used at least once as constituents. For recall however, we use a weighted version: each constituent gets multiplied by the number of time it appears as constituent. This therefore corresponds to an optimistic version, supposing the best case that all constituent-occurrences of the given substring will be correctly identified. By focusing on repeated substrings, this means that the maximal recall that can be obtained on the Penntree bank is 61%. We then define F_1-measure in the usual way, as the harmonic mean between precision and recall.

We plot this measure against different values of x and k in Fig. 1. Note that the upper left corner corresponds to the case of maximal repeats. For super-maximal repeats – not depicted as the value of x and k depends on the particular word – the value is 0.047 and for largest-maximal repeats 0.25. Note that part of the increase when the context increases could be explained simply by the fact that more frequent substrings are more probable to be constituents (see Fig. 2). However, this does not explain the dissymmetry between the left and right context. Note that while a higher diversity in context increases the score in general, there seems to be a higher dependency on the right context than the left one. *xkcd*-repeats offer a direct way of capturing this dissymmetry.

Finally, due to the roof imposed by the maximal value of recall, the F_1 value is actually indicating a high precision score (0.84 at the best F_1 value, and increasing up to 0.97 at the right border).

4. Computation

We will present and compare three algorithms to compute *xkcd*-repeats, all using the suffix array. We will analyze their running time and compare them

[1] `http://www.cis.upenn.edu/~treebank/home.html`

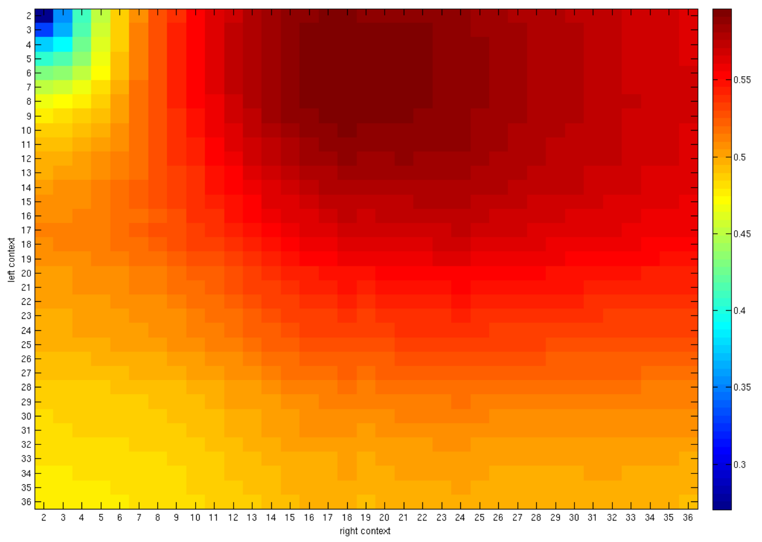


Figure 1: weighted F_1 (see text for details) for different values of $\langle x, k\rangle$

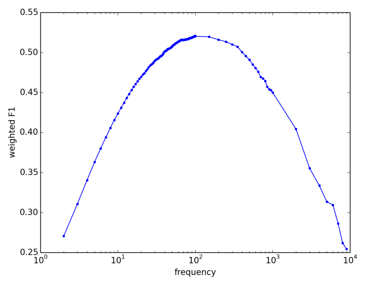


Figure 2: weighted F_1 (y-axis) by looking at maximal repeats that appear at least k times (x-axis)

empirically.

First note that a straightforward way of computing all *xkcd*-repeats would be the following two-stage approach: first, compute all repeats. Then, for each repeat ω inspect all occurrences and store two sets of symbols: those occurring to the left and to the right (this is, $\{s[i-1]\}$ and $\{s[i+|\omega|\}, \forall i \in occ_s(\omega))$. *xkcd*-repeats are then those where the size of these sets are greater than x and k, respectively. Unfortunately, such an approach is $\mathcal{O}(n^2)$, as there may be this number of simple repeats in s. Even if we are only interested in *xkcd*-repeats such that $\max(x,k) \geq 2$ and we precompute only the linear number of left (or right) maximal repeats, the total number of *occurrences* of such repeats is still $\mathcal{O}(n^2)$ in general.

4.1. Simple algorithm

Repeats can easily be computed through the enhanced suffix array because all occurrences of a given repeat are adjacent in the suffix array. Moreover, information about the right context can also be easily obtained through the same way, by analyzing how the *lcp* values evolve.

We keep a stack of the current analyzed repeat and traverse the suffix array in order. If the value of the lcp value remains equal it indicates just another occurrence of the current repeat (which is at the top of the stack), but with a different right context. An increase in the *lcp* value indicates not only the presence of another repeat, but also that the current one is not adding an additional right-context until the newly found repeat is popped out of the stack. Finally, a decrease in the *lcp* value indicates the last occurrence of a repeat, and triggers the eventual output of the current analysed repeat.

Unfortunately, information of the left-context is much harder to get, as it is spread out over the suffix array. An easy way of collecting this (which we will improve later on) is to store all the symbols appearing as left context. When a repeat is popped out, these are then inherited by the topmost repeat in the stack. Recording all the left-contexts adds an extra $|\Sigma|$ factor to the space and time complexity. In our implementation we actually used a set implementation which adds an additional $\log(|\Sigma|)$, but which allows a better trade-off with the memory requirements than using a bit array for example.

The exact algorithm is depicted in Alg. 1, which presupposes the existence of the *lcp* array and *sa* array. Each repeat ω is represented by a tuple $\langle p, \ell \rangle$, where p is the leftmost occurrence of repeat ω in s and $\ell = |\omega|$. While having the leftmost occurrence is not necessary at this stage, we will use this later on. In line 23 a new repeat is added to the stack. Keeping track of where this repeat started (variable *st*) is an important detail [23], which we extend with tracking also the set of left context seen so far (variable *stlc*). Note that, thanks to the way the suffix array is built, the repeat added at this stage has exactly two different right contexts.

4.2. Using Dynamic Range Computation

What we need in order to determine if a repeat has enough left contexts is to compute the number of different elements in the virtual array $[lc[i], \ldots, lc[p]]$,

Algorithm 1 Computation of *xkcd*-repeats in $\mathcal{O}(|\Sigma|n)$ (with $x, k \geq 2$)

xkcd (s,sa,lcp,x,k)

Input: string s, suffix array sa, lcp-array, minimal value of right and left context diversity $x, k \geq 2$

Output: $xkcd$-repeats in the form $\langle p, \ell \rangle$

```
1:  $T$ = **empty stack** // leftmost occurrence, length, size of right ctxt, left ctxt
2:  $\langle p, \ell, r, lc \rangle := \langle 0, 0, 1, \{\$\} \rangle$
3:  $T.push(\langle p, \ell, r, lc \rangle)$ // ensures that the stack never becomes empty
4:  **for all** $i \in [2..n+1]$ **do**
5:      $st := sa[i-1]$
6:      $stlc := \{s[st-1]\}$
7:      **while** $T.top().\ell > lcp[i]$ **do** // last occurrence of a repeat
8:          $\langle p, \ell, r, lc \rangle := T.pop()$ // repeat of length $\ell$ with leftmost occurrence $p$ and $r$,$|lc|$ different right/left context
9:          $st := p$
10:         $T.top().p := \min(T.top().p, p)$
11:         $stlc := lc$
12:         **if** $r \geq k \wedge |lc| \geq x$ **then**
13:             **output** $\langle p, \ell \rangle$ // has $i - p$ occurrences and exactly 2 right ctxt
14:         **end if**
15:         $T.top().lc := T.top().lc \cup lc$
16:     **end while**
17:     **if** $T.top().\ell = lcp[i]$ **then** // new occurrence of same repeat
18:         $T.top().r := T.top().r + 1$
19:         $T.top().lc := T.top().lc \cup \{s[sa[i]-1]\}$
20:         $T.top().p := \min(T.top().p, sa[i])$
21:     **else** // new repeat, which already has $i - st$ occurrences
22:         $stlc := stlc \cup \{s[sa[i]-1]\}$
23:         $T.push(\langle \min(sa[i-1], sa[i], st), lcp[i], 2, stlc \rangle)$
24:     **end if**
25: **end for**
```

where $lc = [s[sa[1]-1], \ldots, s[sa[n]-1]]$ (the symbols occurring to the left of $sa[i]$). The way Alg. 1 achieves this is by storing explicitly these different elements in a set, but this problem (known as *color counting*, see [18, Problem 11]) has already been studied in the literature. The best know solution [5] requires $\mathcal{O}(\log n / \log \log n)$ time for each query. In our implementation we used a simpler and very easy to implement solution based upon Fenwick trees (also called BIT trees) [10]. Given an array x of integers, a Fenwick tree permits to compute prefix sums $psum_x(k) = \sum_{i=1}^{k} x[i]$ in time $\log(|x|)$, and to modify $x[i]$ also in time $\log(|x|)$.

We traverse the suffix array as in Alg. 1, updating a Fenwick tree over a binary array *islast* which contains an 1 at position i if the last occurrence of *lc[i] – so far –* is i, and 0 otherwise. This update is done at the beginning of the outermost loop in Alg. 1. Therefore, when a repeat is popped out the number of its different left contexts is given by the value $psum_{islast}(i) - psum_{islast}(p-1)$[2]. To perform the update over *islast* we need an additional array *last* of size $|\Sigma|$ that keeps for each symbol σ its right-most position so far in *lc*. The update then becomes simply to set $islast[last[lc[i]] = 0$ (except if this is the first occurrence of $lc[i]$) and $islast[i] = 1$; and of course $last[lc[i]] = i$. Remember that all updates of *islast* cost $\log(n)$ as its Fenwick tree has also to be updated.

4.3. A truly linear algorithm

Supposing a fixed alphabet is acceptable for many applications, including genetic string analysis on the original alphabet (of size 4 or 20). However, as soon as new symbols are added [7] or the symbols of the analyzed strings correspond to word types in a natural language document, having the time complexity depending on the size of Σ can become a problem. We present an algorithm that computes all *xkcd* repeats in optimal linear time, even for an integer alphabet.

Like in many algorithms based on suffix data structures, reasoning with the right context in Alg. 1 is straightforward. It is the left context which adds complexity. The algorithm traverses the *lcp-interval tree* [1] bottom-up. Any information over the left context that can be computed in constant time, based on the information of its children does therefore not present problems. This is the case for maximal repeats (where we are interested in context diversity), super-maximal repeats (context uniqueness of all occurrences) [23] and largest-maximal repeats (context uniqueness of at least one occurrence). However, for *xkcd*-repeats we are interested in the number of *different* contexts and there does not seem to be an easy way of computing this based on the different contexts of its children.

We therefore diverge from other algorithms that compute some classes of repeats with the suffix array. Our algorithm is divided in three stages: computation of (i) *krcd*-repeats, (ii) *xlcd*-repeats and (iii) their merging. In order to

[2] $psum_{islast}(i)$ is the number of different symbols encountered so far as left context, and can therefore be kept as global variable

keep linearity, we consider only the case of $x, k \geq 2$. That is, it will not compute *xkcd*-repeats that are not maximal. Besides the reasons stated before, this also ensures that both step (i) and (ii) run in linear time.

4.3.1. Computation of krcd-repeats

In its most basic form, this is just a simplified version of Alg. 1, where we ignore everything related to the left context. We are, however, interested in having a fixed representation for each repeat that permits to compare it afterwards (in step (iii)). Now, it becomes important to have a canonical definition of a repeat, which we take here to be its leftmost occurrences, together with its length. As shown in Alg. 1, the left-most position for a node in the lcp-interval tree is one of those statistics that can easily be computed from the left-most occurrences of its children (variable p).

The final output of this phase is an array of lists denoted by q_{krcd}. For each position $1 \leq p \leq n$, the list $q_{krcd}[p]$ contains the length of all *krcd*-repeats whose first occurrence in s is at position p. An important fact is that each list should be sorted in strict decreasing order.

The following lemma shows that this order is obtained for free.

Lemma 3. *If Alg. 1 outputs $\langle p_1, \ell_1 \rangle$ before $\langle p_2, \ell_2 \rangle$ and $\ell_1 < \ell_2$ then $p_1 \neq p_2$*

Proof. Two cases are possible:

1. $\langle p_2, \ell_2 \rangle$ is already in the stack when $\langle p_1, \ell_1 \rangle$ is popped out. This however is absurd, because at any moment the elements in the stack are in ascending ℓ order (due to the condition to enter the branch of line 23).
2. $\langle p_2, \ell_2 \rangle$ appears in the stack after popping $\langle p_1, \ell_1 \rangle$.
   In that case, $s[p_1 + \ell_1] < s[p_2 + \ell_1]$ which implies $s[p_1 \ldots p_1 + \ell_1] < s[p_2 \ldots p_2 + \ell_1]$. Therefore $p_1 \neq p_2$, as they would be the same string otherwise.

□

4.3.2. Computation of xlcd-repeats

Computing *xlcd* on a string s is equivalent to compute *krcd* repeats on the reversed string ($\overleftarrow{s}$). However, in order to compare these repeats to the *krcd*-pairs, we need to compute the maximal (right-most) occurrence (replace min by max in Alg. 1). An $\langle p, \ell \rangle$ *krcd*-repeat on $\overleftarrow{s}$ corresponds therefore to the *xlcd*-repeat $\langle n - p + \ell, \ell \rangle$ of s. We define $inv(p, \ell) = n - p + \ell$, which gives the index over s corresponding to position p over $\overleftarrow{s}$. Moreover, if p is the rightmost occurrence of a repeat ω in $\overleftarrow{s}$, then $inv(p, |\omega|)$ is the left-most occurrence of $\overleftarrow{\omega}$ in s.

Equivalently to step (i), the output here is an array of lists $\overleftarrow{q}_{krcd}$, where $\overleftarrow{q}_{krcd}[p]$ contains the length of all *krcd*-repeats whose last occurrence in $\overleftarrow{s}$ is at position p.

4.3.3. Merging

In order to do the merging in step (iii) in linear time we need to transform $\overleftarrow{q}_{krcd}$ to q_{xlcd} in linear time, such that the lists at each position are sorted. Lemma 3 ensures this for q_{krcd} already. The *xlcd*-repeats however are computed as *krcd*-repeats on the reversed string, and it is not trivial which should be the right order to traverse them to keep order. The following lemma resolves this:

Lemma 4. *Let $\langle p, \ell_1 \rangle$ and $\langle p, \ell_2 \rangle$ be two xlcd-repeats (with p being the left-most occurrence), such that $\ell_1 < \ell_2$, and let p'_1, p'_2 be such that $p = inv(p'_1, \ell_1) = inv(p'_2, \ell_2)$. Then, $p'_1 > p'_2$*

Proof. It is direct from the definition of *inv*:

If $n - p'_1 + \ell_1 = inv(p'_1, \ell_1) = p = inv(p'_2, \ell_2) = n - p'_2 + \ell_2$, and $\ell_1 < \ell_2$, then $p'_1 > p'_2$. □

So, to keep the order in the lists of q_{xlcd} of the original string, it is enough to traverse the $\langle p, \ell \rangle$ *krcd*-repeats of $\overleftarrow{s}$ in decreasing order of p and to add them in this order to $q_{xlcd}[inv(p, \ell)]$.

Lemmas 3 and 4 ensures that the lists $q_{xrcd}[i]$ and $q_{klcd}[i]$ are sorted in decreasing and increasing order, respectively. Finding the intersection between them can then be done in linear time

Theorem 1. *The xkcd-repeats of string s can be computed in time $\mathcal{O}(|s|)$ (for $x, k \neq 1$), independent of the size of the alphabet.*

For completeness, we present in Alg. 2 the necessary steps to compute all *xkcd* repeats, starting from the q_{krcd} and $\overleftarrow{q}_{krcd}$ arrays.

4.4. Comparison

We compared the execution time of all three algorithms for strings of different alphabet size, over two kind of strings: randomly generated (uniform and independent distribution of symbols) over different alphabet size, and an English wikipedia dump (of Sept 2012), where each tokenized word[3] was assigned an integer identifier. There were 9 million different symbols, distributed as expected by a power law. Herdan's law says that the number of word type in a string of size n is expected to be $k * n^{\beta}$, with typical values of $30 \leq k \leq 100$ and $\beta \approx 0.5$ [16]. However, in our case we did not perform any cleaning and included all the XML meta-data of the wikimedia dump. For the random strings, we report the average over 5 runs for each point, and for the wikipedia string we took increasing prefixes.

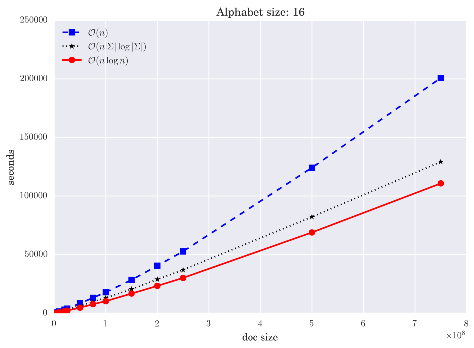
Because the linear algorithm requires to construct two suffix array, the choice of which algorithm to use is crucial. We used Yuta Moris' implementation[4] of the SAIS algorithm [29], the fastest of the variants we tried, and which also runs in linear time. While we measured only user time, we did not have major

[3]we used the NLTK library of python for tokenization

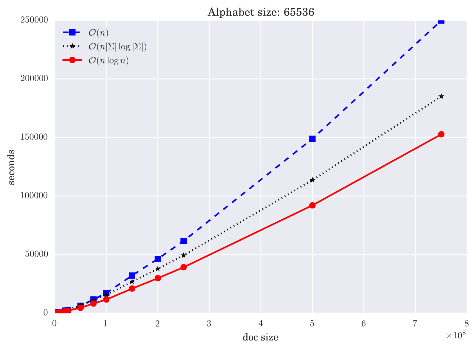
[4]`https://sites.google.com/site/yuta256/sais`

Algorithm 2 Computation of *xkcd*-repeats in linear time

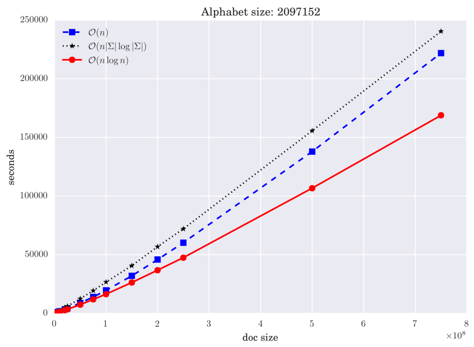
```
$merge(q_{krcd}, \overleftarrow{q}_{krcd})$
Input: $q_{krcd}$ array of lists, of size $n$. Output of modified Alg. 1 (see Sect. 4.3.1).
    $q_{krcd}[p]$ contains the list of lengths of $krcd$ repeats starting at position $p$ of
    $s$. It is sorted – by construction – in descending order.
    $\overleftarrow{q}_{krcd}$ array of lists, of size $n$. Output of modified Alg. 1 (see Sect. 4.3.1),
    with max instead of min. $q_{krcd}[p]$ contains the list of lengths of $krcd$ repeats
    starting at position $p$ of $\overleftarrow{s}$. It is sorted – by construction – in descending
    order
$q_{xlcd}$ := array of lists, of size $n$
// transform $\overleftarrow{q}_{krcd}$ into $q_{xlcd}$
for all $p$ from $n$ to 1 do
    for all $j$ from 1 to $|\overleftarrow{q}_{krcd}[p]|$ do
        $\ell = \overleftarrow{q}_{krcd}[p][j]$
        $q_{xlcd}[n-p+\ell].pushBack(\ell)$
    end for
end for
for all $p$ from 1 to $n$ do
    // $q_{xlcd}[p]$ now contains the length of $xlcd$ repeats at $p$
    $i := 1$
    $j := |q_{xkrd}[p]|$
    while $i \leq |q_{xlcd}[p]|$ and $j \geq 1$ do
        if $q_{xlcd}[p][i] = q_{krcd}[p][j]$ then
            output$(p, q_{xlcd}[p][i])$
            $i := i + 1$
            $j := j - 1$
        else if $q_{xlcd}[p][i] > q_{krcd}[p][j]$ then
            $i := i + 1$
        else
            $j := j - 1$
        end if
    end while
end for
```



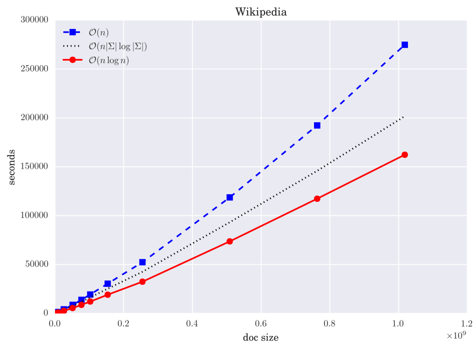
(a) Uniform IID tokens, with $|\Sigma| = 2^4$



(b) Uniform IID tokens, with $|\Sigma| = 2^{16}$



(c) Uniform IID tokens, with $|\Sigma| = 2^{21}$



(d) Samples over a dump from wikipedia

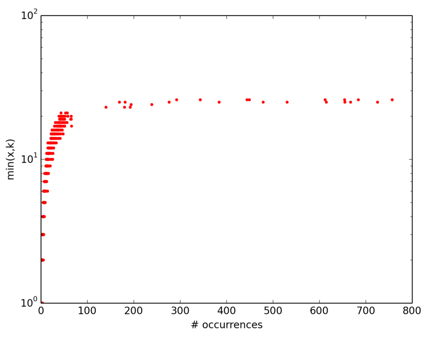
Figure 3: (User) Time usage of all three algorithms, over different document lengths. All datapoints are the average over 5 runs.

problems with swapping in the size of strings considered here (the machine had 32GB of RAM).

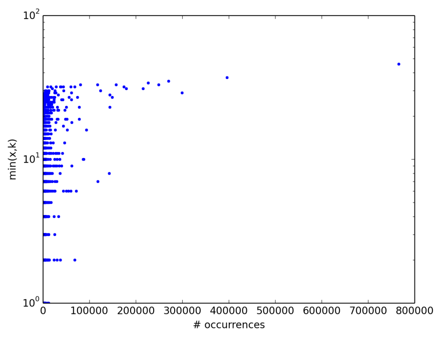
As can been seen in Fig. 3, only with very large alphabets the linear algorithm effectively outperforms the $\mathcal{O}(|\Sigma|n\log|\Sigma|)$ one. In all cases, the BIT implementation outperforms all other. However, it should be noted that most of the time in the linear algorithm is used in the construction of the two suffix arrays and *lcp* (85% of the total time, compared with 68% for the BIT version). Any improvement in this, or creating both suffix arrays at the same time [20] would directly impact these plots.

5. Unbounded context-diverse repeats

Fixing the context of an occurrence to be the single symbols before and after an occurrence enables the design of efficient algorithms, but to the cost of a constraint which has no direct semantic interpretation. Consider for instance a travel corpus with several occurrences of *Alice travels to [CITY]*. If most



(a) Uniform IID generated string of length 10^4 over an alphabet of 26.



(b) King James Version of the Bible (using characters as symbols).

Figure 4: A dot at (x, y) corresponds to a maximal repeat with x occurrences and y different right and left contexts.

city-names in these documents start the same (with *New* for instance), then *travels to* would have a very low right-context, ignoring that *New* itself is not a constituent but only part of one. We argue that *New* should not be considered as a context, but that the whole city-name should be. Constraining the context to be of size 1 makes for an asymmetric treatment of constituents and context: while the former is allowed to be of unbounded length, the latter is of fixed size. It also contradicts our conclusion in [12] that using a bag-of-n-grams – with fixed n — is less expressive than a bag-of-∞-grams, where all repeats are considered. Having to fix n adds yet another parameter to models which in general already have several parameters to optimize. Moreover, any fixed n will not take into account different contexts that are longer than n and ignore the common factor of contexts shorter than that.

Using a context of size 1 adds another limitation: the total size of the context is bounded by the alphabet size. This becomes obvious in Fig. 4, where we plot all maximal repeats, showing the number of occurrences versus the maximal value m such that the repeat is $\langle m, m \rangle$–context-diverse. The size of the context is upper bounded by the size of the alphabet (26 in the case of the synthetic data in Fig. 4a and 63 for the natural language one in Fig. 4b). Two words of completely different occurrence may have the same amount of different contexts, without any possibility of discriminating them about their potential interest as semantic units.

We propose therefore to apply a similar ideas as in [12] to the notion of context: instead of only considering a single symbol, we propose to consider all possible substrings that start immediately to the right (respectively, end immediately to the left).

However, if we just use the counts of different substrings starting at the right, this would clearly favor words occurring at the beginning of the string.

Coming back to the underlying assumption that repetition is a good indicator of importance, a first filtering would be to count only repeated substrings. But here again the notion of maximality is important: a straightforward approach of just counting repeats starting to the right of the occurrences would results in words that end at the start of a long, repeated block of text, without considering the importance of this block of text.

Instead, we use the same notion of maximal repeat as defined before and consider only those repeats that are right-maximal with respect to all its occurrences to the right of an occurrence of ω. Consider for instance, word ω with exactly 5 occurrences in string s:

$$s = \ldots \omega ab \ldots \omega ac \ldots \omega de \ldots \omega dfg \ldots \omega dfh$$

If only a unit-length context is taken into account, ω would have a right context of 2 ($\{a, d\}$). With our definition, its right context is 3, composed of $\{a, d, df\}$, as these are the repeated substrings to the right of ω which are maximal in that set of occurrences. Note how d and df are taken as different contexts.

Formally, we will define that notion of right and left-maximality with respect to a set of occurrences:

Definition 3. *ν is right-maximal over I iff $I \subseteq occ_s(\nu)$ and $|\{s[i + |\nu|] : i \in I\}| \geq 2$.*

and

Definition 4. *ν is left-maximal over I iff $I \subseteq occ_s(\nu)$ and $|\{s[i-1] : i \in I\}| \geq 2$.*

Note that if a word ν is right-maximal over a subset I, then it is right-maximal over any superset $I' \supseteq I$ (including $I' = occ(\nu)$).

We now define our notion of ∞–context-diversity:

Definition 5. *The ∞–right-context of a maximal repeat ω is the set of right-maximal substrings ν over $I = \{i + |\omega| : i \in occ_s(\omega)\}$.*

Equivalently, the ∞–left-context of a maximal repeat ω is the set of left-maximal substrings ν over $I = \{i - |\nu| : i \in occ_s(\omega)\}$.

Note that contexts of length 1 that are unique are not counting towards the total count. Therefore, ω can be a k–right-context repeat, but not a k–∞-right-context one (and vice-versa: they are disjoint in general).

5.1. Algorithm

At a first glance, computing ∞-cd seems not an easy task as one has to keep track simultaneously of the word currently analyzed, as well as of all repeats occurring as contexts. However, the following characterization simplifies this task:

Proposition 2. *Let S be the suffix tree of string s. The ∞–right-context of a word ω corresponds to all internal nodes of the subtree of S rooted by ω.*

Proof. We first note that there is a node ω in S: This comes from the fact that ω is by definition a maximal repeat, and the internal nodes of S are all right-maximal repeats of s [14]. That a right-context ν is right-maximal over the occurrences starting after an occurrence of ω is equivalent to say that $\omega\nu$ is right-maximal. But there exists a 1-to-1 relationship between right-maximal repeats of the form $\omega\nu$ and the internal nodes under ω. $\square$

In Alg. 3 we show how to compute ∞–right-cd repeat using a suffix array, a more memory-efficient data structure than suffix trees. We traverse this data structure as before, which corresponds to a depth-first search on the the suffix tree (the so-called *lcp-interval tree* [1]). The size of its ∞–right-context can then be easily obtained by keeping a global counter of repeats that passed over the stack (variable *counter* in Alg. 3). As before, an increase of the values in the *lcp* array denotes a new repeat, a decrease the end of a repeat and equality just another occurrence of the current topmost repeat.

Of course, ∞–left-maximal repeat can be computed equivalently using the suffix array of the reversed string. We use the same idea of Sect. 4.3 to merge a linear list of substrings on s with another linear list of substrings on $\overleftarrow{s}$ in linear time. Applying this technique here yields the following:

Theorem 2. *Given a string s, all $\langle x, k\rangle$–∞–context-diverse repeats over s (with $x, k \neq 1$) can be computed in $\mathcal{O}(|s|)$ time.*

Because there might be a linear number of these repeats, this algorithm is optimal.

5.2. Results

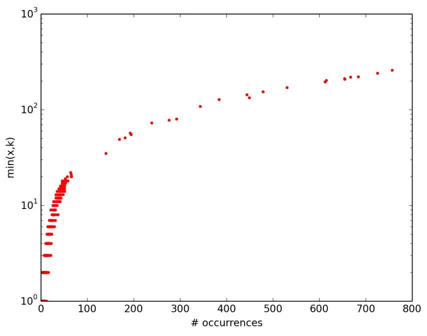
We implemented Alg. 3, and computed ∞–context-diverse repeats over the same two strings as in Fig. 4. The corresponding plots can be appreciated in Fig. 5. As it can be seen, the linearity with respect to the number of occurrences (seen as a logarithmic curve in the figures due to the scale of the y-axis) is not interrupted, making it easier to detect outliers on this curve which may be of potential interest as constituents.

6. Incremental Computation

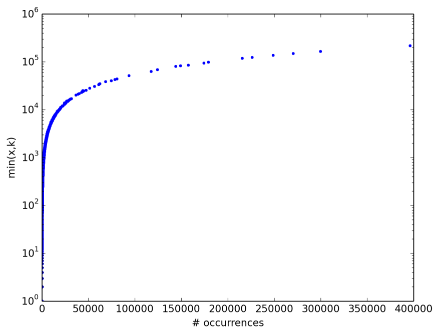
Existing algorithms to compute repeats, including those presented so far, work in an off-line mode, where the whole string is supposed to be available. This assumption may be true in some applications, but many use-cases around natural language work in a streaming setting (ex: online news analysis, real-time document classification, etc). An approach that models documents with repeats (like the one we proposed in [12]) can not be adapted directly to such a streaming setting without paying an extremely high efficiency toll. We present here a general method to compute – iteratively – a set of characteristics on context-diverse repeats. For this we will rely on the incremental suffix tree creation algorithm of Ukkonen [27].

Algorithm 3 Computation of ∞–right-context-diverse repeats in $\mathcal{O}(|s|)$

```
∞–rcd(lcp,k)
Input: lcp-array, minimal value of right context diversity x
Output: ∞–right-context-diverse repeats in the form ⟨ position over suffix array, length ⟩
 1: $T$ = empty stack
 2: $\langle p, \ell, c\rangle := \langle 0, 0, 0\rangle$
 3: $T.push(\langle p, \ell, c\rangle)$ // ensures that the stack never becomes empty
 4: $counter = 0$
 5: for all $i \in [2..n+1]$ do
 6:     $st := i - 1$
 7:     while $T.top().\ell > lcp[i]$ do // last occurrence of a repeat
 8:         $\langle p, \ell, c\rangle := T.pop()$
 9:         $st := p$
10:         if $counter - c \geq k$ then
11:             output $\langle p, \ell\rangle$ // has $i - p$ occurrences
12:         end if
13:     end while
14:     if $T.top().\ell \neq lcp[i]$ then // new repeat, which already has $i - st$ occurrences
15:         $T.push(\langle st, lcp[i], counter\rangle)$
16:         $counter := counter + 1$
17:     end if
18: end for
```



(a) Uniform IID generated string of length 10^4 over an alphabet of 26.



(b) King James Version of the Bible (using characters as symbols).

Figure 5: A dot at (x, y) corresponds to a maximal repeat with x occurrences and y different ∞-right and ∞-left contexts.

We will analyze two variants of the problem, and consider various different equivalence classes of repeats. In the first variant we are only interested in updating the set of repeats of the particular class we are considering, while in the second one we also want to update the occurrence number of these repeats.

All classes we will consider are super-sets of maximal repeats. In particular, they are all a sub-set of the nodes of the suffix tree (which are all the right-maximal repeats). The problems therefore becomes to update any marker (or other additional information) over these nodes.

For this section, we will define the concatenation of s_1 and s_2 as $s_1.s_2 = s_1\$s_2$, with \$ a fresh symbol.

Problem 1. *For a class of repeats* $\mathbf{X}$ *(with* $\mathbf{X}$ *one of* $\mathbf{MR}, \mathbf{LMR}, \mathbf{SMR}$ *or* $\mathbf{XKCD}$*) and given documents* $D = \{d_1 \dots d_k\}$*, document* d_{k+1} *and a suffix tree on* $S = d_1.d_2.\dots.d_k$ *where all nodes corresponding to* $\mathbf{X}(S)$ *are marked, return a suffix tree on* $S.d_{k+1}$ *such that all nodes corresponding to* $\mathbf{X}(S.d_{k+1})$ *are marked.*

In principle one would like an algorithm solving this problem to run in time proportional to $n = |d_{k+1}|$. This may be impossible however, in the case that the number of nodes to be (un-)marked is larger than n. We therefore say that an algorithm resolving Problem 1 is *optimal* if it runs in time $\mathcal{O}(\max(n, |\mathbf{X}_{\mathit{diff}}(S,d)|))$, where $\mathbf{X}_{\mathit{diff}}(S,d)$ are the changes of marker that have to be performed: $\mathbf{X}_{\mathit{diff}}(S,d) = (\mathbf{X}(S.d) \setminus \mathbf{X}(S)) \cup (\mathbf{X}(S) \setminus \mathbf{X}(S.d))$.

We will present a general solution that works for four classes (*xkcd*, maximal, largest-maximal, super-maximal) in non-optimal time and give an optimal solution for the case of maximal repeats.

Note that Problem 1 only marks the corresponding repeats, but does not retrieve any additional information. To do this, potentially the whole suffix tree has to be traversed, incurring in an additional penalty of $|S|$ after each increment. We therefore also consider a second problem, related to getting the most basic information associated to each repeat: its number of occurrences.

Problem 2. *For a class of repeats* $\mathbf{X}$*, and given documents* $D = \{d_1 \dots d_k\}$*, document* d_{k+1} *and a suffix tree on* $S = d_1.d_2.\dots.d_k$ *where all nodes corresponding to* $\mathbf{X}(S)$ *are marked and augmented with their number of occurrences; return a suffix tree on* $S.d_{k+1}$ *where all nodes corresponding to* $\mathbf{X}(S.d_{k+1})$ *are marked and augmented with their number of occurrences.*

An algorithm resolving Problem 2 is said to be optimal if it runs in time $\mathcal{O}(\max(n, |\mathbf{X}_{update}(S,d)|)$, where $\mathbf{X}_{update}(S,d)$ are the updates that have to performed, consisting of all repeats of the class $\mathbf{X}$ that occur in d, plus any repeat that does not occur anymore in $\mathbf{X}$ due to the addition of d: $\mathbf{X}_{update}(S,d) = \{w \in d : w \in \mathbf{X}(S.d)\} \cup (\mathbf{X}(S) \setminus \mathbf{X}(S.d))$. Note that the size of this set can be much larger than the length of the newly added document:

Proposition 3. $|\mathbf{MR}_{update}(S,d)| \in \Theta(|d|^2)$

Proof. It is enough to give a document d such that any substring of d is different and a maximal repeat over $S.d$. As we do not bound the number of documents that appeared previously, this can always be achieved.

□

6.1. Cover of a new document

When a new document d_{k+1} is added to S, the new suffix tree will have $|d_{k+1}|+1$ new leaves corresponding to each of the suffixes of d_{k+1}. Any ancestor of these leaves can potentially change its maximal class, either because it is repeated for the first time or because of a change in its context sets. It is therefore this set of nodes we will be traversing after each update. Formally:

Definition 6. *We define $cover(S, d_{k+1})$ as the set of substrings of d_{k+1} which are right-maximal repeats in $S.d_{k+1}$.*

This corresponds therefore to those internal nodes of the suffix tree which are ancestor of leaves added for d_{k+1}. However, the cover set should not be traversed in any arbitrary order: each node should of course be visited only once, and only after having visited all its children. Any information on the occurrence of an internal node v can be obtained by aggregating correctly this same information of the children of v. The way we achieve this is by ordering the nodes with respect to the lengths of their represented substring. This defines therefore a partial order on the nodes of the suffix tree, where $v < w$ iff v is a prefix of w and $|v| < |w|$. The nodes can then be correctly traversed by using this order with a priority queue, which is initialized with all new leaves. The order in which non-ordered pairs are selected is not important: the priority property ensures that when a node v is visited, all his descendants in the cover were visited before.

6.2. Problem 1: Updating a class

With the definition of cover given as before, any update algorithm runs then in time $\mathcal{O}((|cover| + n)\log(n))$, assuming than that the aggregation of information from the descendants can be done in constant time. The additional n factor is due to the leaves of d_{k+1} and the logarithmic factor is due to the complexity of insertion and deletion in the queue. The queue could be replaced by an array p of lists of nodes of maximal size n, where $p[i]$ is the list of nodes whose depth is equal to i and an boolean array *added*, *added*$[v]$ being True iff if the node v was added to p. These two arrays can be updated in constant time and the additional cost in memory trades off with speed, making the cover traversal $\mathcal{O}(|cover| + n)$.

Updating those nodes belonging to a given class is then straightforward, assuming that each node is enriched with additional information: we keep for each node v two sets of symbols: the first set lc_{unique} will contain those symbols c such that c is a left context of a leaf-child of v and there is not another leaf of v with the same left-context. The second set, lc will be disjoint to lc_{unique} and contain all other characters which are left-contexts of a leaf of v but are not in lc_{unique}. With these definitions, each node can easily be updated with the following rules:

- v is a maximal repeat iff $|v.lc| + |v.lc_{unique}| > 1 \wedge |children(v)| > 1$
- v is $\langle x, k\rangle$-cd iff $|v.lc| + |v.lc_{unique}| \geq x \wedge |children(v)| \geq k$
- v is a supermaximal repeat iff $|v.lc_{unique}| = |children(v)|$
- v is a largest maximal repeat iff $|v.lc_{unique}| > 0$

Of course, both sets have also to be updated. This can be done in a straightforward manner by updating both sets of the parent of each visited node, adding a additional $|\Sigma|$ factor to the final complexity.

6.2.1. A optimal algorithm for maximal repeats

The case of maximal repeats is special, because once a repeat becomes maximal, it can not become non-maximal in future iterations. Formally:

Definition 7. *A class of substrings* $\mathbf{X}$ *is said to be invariant iff:*
$w \in \mathbf{X}(s) \implies w \in \mathbf{X}(ss')\ \forall s, s' \in \Sigma^*$

Maximal repeats are invariant[5] and therefore $\mathbf{X}_{diff}(S, d)$ reduces to the set of newly added maximal repeats through the addition of d. Furthermore, maximal repeats have an additional property: if a node v is maximal, than all its ancestors are too. That is because the context diversity property of maximal repeats is inherited upwards. Therefore, when traversing the cover we can stop as soon as we reach a node that is already maximal. By induction, any of its ancestors will already be marked as such and none of them can cease to be maximal.

For this, it is not even necessary to keep the sets of left-context. A single variable ℓ suffices: it holds a negative value if the current node is maximal. If this is note the case then all its occurrences have a single left-context, and this symbol can then be stored in ℓ.

Using Ukkonen's incremental algorithm to construct the suffix tree we can therefore check any newly created internal node at creation time (Step 2 in the explanation of Gusfield [14]), and check its ancestors until no change is made. Because the only possible change is marking a non-maximal node as maximal, and because the traversal stops as soon as a node is not changed, the algorithm is optimal for fixed-size alphabets.

6.3. Problem 2: Counting repeats

Using the concept of cover, counting the number of occurrences of any internal node can be done similarly to the update of the membership to the classes of repeats. The number of occurrences of a repeat is the number of leaves in

[5]As are *xkcd* repeats, if x and k are constants; but neither are largest, super-maximal or *xkcd* if x or k are functions dependent on the number of occurrences of the repeat.

the subtree rooted at the node representing it, a value that can be kept in an additional variable per node. In addition, each node should keep an auxiliary variable with the number of new leaves it has due to the incorporation of d_{k+1} to the suffix tree. In a first traversal of the cover each node updates this auxiliary variable of his parent, and in a second traversal each node updates its count of subtrees adding the number of new leaves.

6.4. Measuring the Gap

Using the concept of *cover* allows a general way of computing different information of repeats in an incremental way. The framework is expressive enough to compute a large range of information as exemplified by the incremental computation of number of occurrences. For some of the cases however it is not optimal, in the sense that the cover may be larger than the absolute minimum number of nodes that could be inspected. We will not address here the issue if such optimal algorithms may exist. Instead, we will analyze the gap between the size of the cover and the minimal optimal set of nodes to inspect. We restrict this analysis to the case of maximal repeats only and consider its worst case behavior, over two simulated scenarios (documents originated from an i.i.d source and real-life natural language text). In all cases we are interested in how $|cover(S, d)|$ behaves with respect to the size of $\mathbf{MR}_{update}(S, d)$ which reduces to $\{w \in d : w \in \mathbf{MR}(S.d)\}$ due to the invariant property of maximal repeats. Because the cover only contains nodes of the suffix tree, which are right-maximal, this reduces to finding substrings of d that are right but not left-maximal in $S.d$.

6.4.1. Asymptotic Analysis

Consider the document $d^{(k)} = a_1 a_2 \dots a_k$ of length k over an alphabet of size k, with all a_i different. The document set consisting of $\{d^{(k)}\}$ alone does not contain any repeat, but we are interested in the update if an exact copy occurs: $D = \{d^{(k)}, d^{(k)}\}$. This document set has only one maximal repeat ($d^{(k)}$ itself), but contains a linear number of left-maximal repeats $(a_1, a_1 a_2, \dots, a_1 \dots a_k)$:

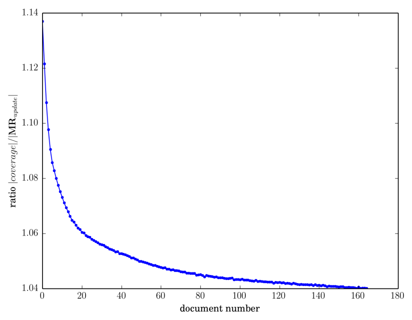
Proposition 4. *The ratio* $\dfrac{|cover(S, d)|}{|\mathbf{MR}_{update}(S, d)|} \in \Omega(|d|)$

We will now see if this scenario reflects what happens in real strings.

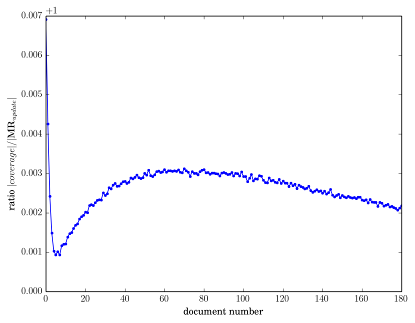
6.4.2. Simulations

We created documents generated from a source emitting symbols independent and identical distributed (with uniform probability). To reduce variations due to different parameters, we generated equally long documents, of one million bytes each.

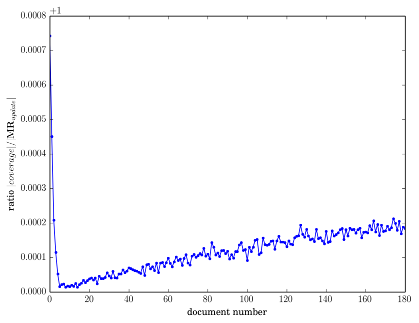
In Fig. 6 we plot the ratio $\frac{|cover(S,d)|}{|\mathbf{MR}_{update}(S,d)|}$ for each document, using different alphabet size. At each iteration, the new document d gets concatenated to S. While there are great relative variations, note that in absolute values the changes are in the order of 10^{-3}. We believe that the behavior using an alphabet size of



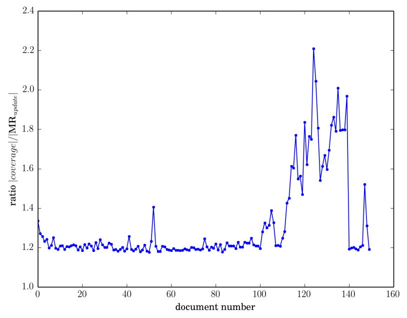
(a) $|\Sigma| = 4$



(b) $|\Sigma| = 100$



(c) $|\Sigma| = 1000$



(d) Wikipedia

Figure 6: Evolution of ratio of cover / maximal-repeats, for random documents (generated from an equiprobable i.i.d. source, varying alphabet size) and natural-language ones.

100 is the most typical, as it is reminescent of a periodic behavior which is known in the expected number of nodes of a suffix tree and maximal repeats [4, 24].

We repeated the previous examples, but instead of generating the document ourselves we used a dump of the complete English wikipedia, and split it into documents of equal size (5MB here). The results can be seen in Fig. 6d: the ratio now is higher than in the artificial strings but still below 2 in general. We verified that varying the length of the documents did not impact this. The sudden peaks seem to belong to cases where long stretches of documents are repeated, coming closer to the worst-case scenario.

7. Conclusions

We investigated classes of repeats which are defined explicitly by the context their occurrences appear. These definitions were motivated by linguistic clues that show the importance of the context to determine which substrings in a text are *constituents* (semantic units). Our definition encompasses well-known classes (maximal and supermaximal) as special cases, while at the same time allowing an algorithm to compute them in asymptotically optimal time. This optimal algorithm however requires the construction of two suffix arrays, and in our implementation another variant – though worse in its worst-case analysis – behaves faster in practice.

We furthermore present a still more general family of these context-diverse repeats, where the context now is not restricted to the immediate symbol preceding or following a substring, but to any maximal substring ending/starting at the left/right of a potential constituent. Surprisingly, these repeats can be computed in linear time too, re-using a trick to merge linearly repeats on the strings and its reverse.

A third contribution of this paper is the presentation of a general framework – relying on Ukkonen's suffix tree creation algorithm – to compute repeats incrementally, when documents arrive one after the other or in mini-batches. For this we introduce the notion of *cover*, which is the set of nodes of the suffix tree that contains all substrings whose condition may change due to the addition of the new document to the collection. It is therefore sufficient to traverse this cover, compared to the full suffix tree of the whole updated collection. Finally, for the specific case of maximal repeats (which has some nice properties regarding the invariability with respect to addition of new documents), we show an optimal algorithm for this same task.

One data-structure we did not explore further here is the suffix automaton (or the Directed Cyclic Word Graph) [3]. Some of the basic notions we rely on are directly expressed in that data structure and although algorithms may not run faster than using suffix arrays they may provide some elegant solution.

We hope that the contribution from this paper will increase the use of concepts from stringology to model natural language documents, an approach which has not sufficiently been studied in our opinion.

References

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

[2] Alberto Apostolico. Of maps bigger than the empire (invited paper). In *SPIRE*, pages 2–9, 2001.

[3] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, Mu-Tian Chen, and Joel Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[4] Anselm Blumer, Andrzej Ehrenfeucht, and David Haussler. Average sizes of suffix trees and DAWGs. *Discrete Applied Mathematics*, 24(13):37 – 45, 1989.

[5] Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. Succinct Orthogonal Range Search Structures on a Grid with Applications to Text Indexing. In *WADS*, pages 98–109, 2009.

[6] Rafael Carrascosa, François Coste, Matthias Gallé, and Gabriel Infante-Lopez. The smallest grammar problem as constituents choice and minimal grammar parsing. *MDPI Algorithms*, 4(4):262–284, 2011.

[7] Rafael Carrascosa, François Coste, Matthias Gallé, and Gabriel Infante-Lopez. Searching for smallest grammars on large sequences and application to DNA. *Journal of Discrete Algorithms*, 11(0):62 – 72, 2012. Special issue on Stringology, Bioinformatics and Algorithms.

[8] Alexander Clark. Learning deterministic context free grammars: The omphalos competition. *Machine Learning*, pages 93–110, Jan 2007.

[9] Alexander Clark, Rémi Eyraud, and Amaury Habrard. A polynomial algorithm for the inference of context free languages. In *ICGI*, Jul 2008.

[10] Peter M Fenwick. A New Data Structure for Cumulative Frequency Tables. *Softw. Pract. Exper.*, 24(June 1993):327–336, 1994.

[11] Matthias Gallé. *Searching for Compact Hierarchical Structures in DNA by means of the Smallest Grammar Problem.* Université de Rennes 1, February 2011.

[12] Matthias Gallé. The bag-of-repeats representation of documents. In *SIGIR*, 2013.

[13] Matthias Gallé and Matías Tealdi. On context-diverse repeats and their incremental computation. In *Language and Automata Theory and Applications*, pages 384–395. Springer International Publishing, 2014.

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

[15] Dan Klein. *The Unsupervised Learning of Natural Language Structure.* PhD thesis, University of Stanford, 2005.

[16] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Inf Retrieval.* Cambridge UP, 2009.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[18] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys (CSUR)*, 46(4):52, 2014.

[19] Jacques Nicolas, Christine Rousseau, Anne Siegel, Pierre Siegel, François Coste, Patrick Durand, Sébastien Tempel, Anne-Sophie Valin, and Frédéric Mahé. Modeling local repeats on genomic sequences. Technical report, INRIA, 2008.

[20] Enno Ohlebusch, Timo Beller, and Mohamed I. Abouelhoda. Computing the BurrowsWheeler transform of a string and its reverse in parallel. *Journal of Discrete Algorithms*, 1:1–13, June 2013.

[21] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[22] Simon Puglisi, William F Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), Jul 2007.

[23] Simon J Puglisi, William F Smyth, and Munina Yusufu. Fast optimal algorithms for computing all the repeats in a string. In *Prague Stringology Conference*, pages 161–169, 2008.

[24] Mathieu Raffinot. On maximal repeats in strings. *Information Processing Letters*, 80(3):165 – 169, 2001.

[25] Heinrich Schütze. Automatic Word Sense Discrimination. *Computational Linguistics*, 24(1), 1998.

[26] Zach Solan, David Horn, Eytan Ruppin, and Shimon Edelman. Unsupervised learning of natural languages. *Proceedings of the National Academy of Sciences*, Jan 2005.

[27] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[28] Menno van Zaanen. ABL: Alignment-based learning. In *International Conference on Computational Linguistics*, 2000.

[29] Sen Zhang, Ge Nong, and Wai Hong Chan. Fast and space efficient linear suffix array construction. In *Data Compression Conference*, Washington, DC, USA, 2008. IEEE Computer Society.