For submission to the Journal of Automata, Languages and Combinatorics
Created on Friday 21st October, 2016

ENHANCED COVERS OF REGULAR & INDETERMINATE STRINGS USING PREFIX TABLES

Ali Alatabbi[(A)] A. S. M. Sohidull Islam[(B)] M. Sohel Rahman[(C,F)]
Jamie Simpson[(D)] W. F. Smyth[(B,E,G)]

[(A)] *Department of Informatics, King's College London*
ali.alatabbi@kcl.ac.uk

[(B)] *Algorithms Research Group, Department of Computing & Software, McMaster University*
sohansayed@gmail.com, smyth@mcmaster.ca

[(C)] *AℓEDA Group, Department of Computer Science & Engineering, Bangladesh University of Engineering & Technology*
msrahman@cse.buet.ac.bd

[(D)] *Department of Mathematics and Statistics, Curtin University of Technology*
Jamie.Simpson@curtin.edu.au

[(E)] *School of Engineering & Information Technology, Murdoch University*

ABSTRACT

A ***cover*** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that $\boldsymbol{x}$ can be constructed from possibly overlapping instances of $\boldsymbol{u}$. A recent paper [12] relaxes this definition — an ***enhanced cover*** $\boldsymbol{u}$ of $\boldsymbol{x}$ is a border of $\boldsymbol{x}$ (that is, a proper prefix that is also a suffix) that covers a *maximum* number of positions in $\boldsymbol{x}$ (not necessarily all) — and proposes efficient algorithms for the computation of enhanced covers. These algorithms depend on the prior computation of the ***border array*** $\beta[1..n]$, where $\beta[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$, $1 \leq i \leq n$. In this paper, we first show how to compute enhanced covers using instead the ***prefix table***: an array $\boldsymbol{\pi}[1..n]$ such that $\boldsymbol{\pi}[i]$ is the length of the longest substring of $\boldsymbol{x}$ beginning at position i that matches a prefix of $\boldsymbol{x}$. Unlike the border array, the prefix table is robust: its properties hold also for ***indeterminate strings*** — that is, strings defined on *subsets* of the alphabet Σ rather than individual elements of Σ. Thus, our algorithms, in addition to being more space-efficient than those of [12], allow us to easily extend the computation of enhanced covers to indeterminate strings. Both for regular and indeterminate strings, our algorithms execute in expected linear time and our experimental results suggest that they are faster than those of [12] in practice. Along the way we establish an important theoretical result: that the expected maximum length of any border of any prefix of a regular string $\boldsymbol{x}$ is approximately 1.64 for binary alphabets, less for larger ones.

[(F)] Currently on a sabbatical leave from BUET. Part of this research was carried out when Rahman was visiting McMaster University under an ACU Titular Fellowship.

[(G)] Supported in part by a grant from the Natural Sciences & Engineering Research Council (NSERC) of Canada.

1. Introduction

The concept of *periodicity* is fundamental to combinatorics on words and related algorithms: it is difficult to imagine a research contribution that does not somehow involve periods of strings. But periodicity alone may not be the best descriptor of a string; for example, $\boldsymbol{x} = abaababab$, a string of length $n = 9$, has period 7 and corresponding ***generator***[1] $abaabab$, but it could well be more interesting that every position but one in $\boldsymbol{x}$ lies within an occurrence of ab. In 1990 Apostolico & Ehrenfeucht [3] introduced the idea of quasiperiodicity: a ***quasiperiod*** or ***cover*** of a string $\boldsymbol{x}$ is a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that any position in $\boldsymbol{x}$ is contained in an occurrence of $\boldsymbol{u}$; $\boldsymbol{u}$ is then said to ***cover*** $\boldsymbol{x}$, which is said to be ***quasiperiodic***. Thus, for example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaaba$. Several linear-time algorithms were proposed for the computation of covers [4, 8, 19, 20], culminating in an algorithm [18] to compute the ***cover array*** $\boldsymbol{\gamma}$, where $\boldsymbol{\gamma}[i]$ gives the length j of the longest cover of $\boldsymbol{x}[1..i]$. Since the longest cover of $\boldsymbol{x}[1..j]$ is also a cover of $\boldsymbol{x}[1..i]$, $\boldsymbol{\gamma}$ implicitly specifies all the covers of every prefix of $\boldsymbol{x}$. A recent paper [2] extends the computation of $\boldsymbol{\gamma}$ to "indeterminate strings" (see below for definition).

Even though the cover of a string can provide useful information, quasiperiodic strings are on the other hand infrequent among strings in general. Another approach to string covering was therefore proposed in [15]: a set $U_k = U_k(\boldsymbol{x})$ of strings, each of length k, is said to be a ***minimum k-cover*** of $\boldsymbol{x}$ if every position in $\boldsymbol{x}$ lies within some occurrence of an element of U_k, and no smaller set of k-strings has this property. Thus $U_2(abaababab) = U_2(ababaaba) = \{ab, ba\}$. In [10] the computation of U_k was shown to be NP-complete, though an approximate polynomial-time algorithm was presented in [14].

Recall that a ***border*** of $\boldsymbol{x}$ is a possibly empty proper prefix of $\boldsymbol{x}$ that is also a suffix: every nonempty string has a border of length zero. Recently the promising idea of an ***enhanced cover*** was introduced [12]; that is, a border $\boldsymbol{u}$ of $\boldsymbol{x} = \boldsymbol{x}[1..n]$ that covers a maximum number $m \leq n$ of positions in $\boldsymbol{x}$. Then the ***minimum enhanced cover*** $\texttt{mec}(\boldsymbol{x})$ is the *shortest* border $\boldsymbol{u}$ that covers m positions, and [12] presented an algorithm to compute $\texttt{mec}(\boldsymbol{x})$ in $\Theta(n)$ time. Thus for $\boldsymbol{x} = abaababab$, $\texttt{mec}(\boldsymbol{x}) = \texttt{ab}$. Further, on the analogy of the cover array defined above, the authors proposed the ***minimum enhanced cover array*** $\texttt{MEC}_{\boldsymbol{x}}$ — for every $i \in 1..n$, $\texttt{MEC}_{\boldsymbol{x}}[i] = |\texttt{mec}(\boldsymbol{x}[1..i])|$, the length of the minimum enhanced cover of $\boldsymbol{x}[1..i]$ — and showed how to compute it in $\mathcal{O}(n \log n)$ time. In this paper we introduce in addition the $\texttt{CMEC}$ array, where $\texttt{CMEC}[i]$ specifies the number of positions in $\boldsymbol{x}[1..i]$ covered by the border of length $\texttt{MEC}[i]$. Thus, for example, $\texttt{MEC}_{\texttt{abaababab}} = \texttt{001123232}$ and $\texttt{CMEC}_{abaababab} = 002346688$. Note that, extending the idea of enhanced cover further, the notions of partial covers and seeds have recently been introduced and investigated in the literature [17, 16].

In order to compute $\texttt{MEC}_{\boldsymbol{x}}$, the authors of [12] made use of a variant of the ***border array*** — that is, an integer array $\boldsymbol{\beta}[1..n]$ in which for every $i \in 1..n$, $\boldsymbol{\beta}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. In this paper we adopt a different approach to the

[1]Notation and terminology generally follow [21].

computation of $\texttt{MEC}_{\boldsymbol{x}}$, using, instead of a border array, the ***prefix table*** $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$, where for every $i \in 1..n$, $\boldsymbol{\pi}[i]$ is the length of the longest substring at position i of $\boldsymbol{x}$ that equals a prefix of $\boldsymbol{x}$. It has long been folklore that $\boldsymbol{\beta}$ and $\boldsymbol{\pi}$ are "equivalent", but it has only recently been made explicit [6] that each can be computed from the other in linear time. However, this equivalence holds only for ***regular*** strings $\boldsymbol{x}$ in which each entry $\boldsymbol{x}[i]$ is constrained to be a single element of the underlying alphabet Σ.

We say that a letter λ is ***indeterminate*** if it is any nonempty subset of Σ, and thus a string $\boldsymbol{x}$ is said to be ***indeterminate*** if some constituent letter $\boldsymbol{x}[i]$ is indeterminate. The idea of an indeterminate string was first introduced in [11] — with letters constrained to be either regular (single elements of Σ) or Σ itself — and the properties of these strings have been much studied by Blanchet-Sadri [7] and her collaborators as "partial words" or "strings with holes". Indeterminate strings can model DNA sequences on $\Sigma = \{A, C, G, T\}$ when ambiguities arise in determining individual nucleotides (letters).

Two indeterminate letters λ and μ are said to ***match*** (written $\lambda \approx \mu$) whenever $\lambda \cap \mu \neq \emptyset$, a relation that is in general nontransitive [13, 24]: $a \approx \{a, b\}$ and $\{a, b\} \approx b$, but $a \not\approx b$. An important consequence of this nontransitivity is that the border array no longer correctly describes *all* of the borders of $\boldsymbol{x}$: it is no longer necessarily true, as for regular strings, that if $\boldsymbol{u}$ is the longest border of $\boldsymbol{v}$, in turn the longest border of $\boldsymbol{x}$, then $\boldsymbol{u}$ is a border of $\boldsymbol{x}$. On the other hand, the prefix array retains all its properties for indeterminate strings $\boldsymbol{x}$ and, in particular, correctly identifies all the borders of every prefix of $\boldsymbol{x}$ [6]. [22] describes algorithms to compute the prefix table of an indeterminate string; conversely, [9] proves that there exists an indeterminate string corresponding to every feasible prefix table, while [1] describes an algorithm to compute the lexicographically least indeterminate string determined by any given feasible prefix table. There is thus a many-many relationship between the set of all indeterminate strings and the set of all prefix tables. Consequently, computing $\texttt{MEC}_{\boldsymbol{x}}$ (or simply `MEC` when there is no ambiguity) from the prefix table $\boldsymbol{\pi} = \boldsymbol{\pi}_{\boldsymbol{x}}$ rather than from a variant of the border array allows us to extend the computation to indeterminate strings.

In Section 2 we outline the basic methodology and data structures used to compute the minimum enhanced cover array from the prefix table, while illustrating the ideas with an example. Then Section 3 provides a proof of the algorithm's correctness, as well as an analysis of its complexity, both worst and average case. Section 4 describes extensions of the basic `MEC` algorithm to enhanced left covers and enhanced left seeds. In Section 5 we discuss the practical application of our algorithms, in terms of time and space requirements, and compare our prefix-based implementation with the border-based implementation of [12]. Section 6 shows how to extend the various enhanced cover array algorithms to indeteterminate strings, while Section 7 summarizes our results and suggests future research directions.

2. Methodology

In this section we describe the computation of $\texttt{MEC}_{\boldsymbol{x}}$, the enhanced cover array of $\boldsymbol{x}$, based on the prefix array $\boldsymbol{\pi}$. Since every minimum enhanced cover of $\boldsymbol{x}$ is also a border of $\boldsymbol{x}$, we are initially interested in the covers of prefixes of $\boldsymbol{x}$. For this purpose we need arrays whose size is B, the maximum length of any border of any prefix of $\boldsymbol{x}$. Noting that B must be the maximum entry in the prefix array $\boldsymbol{\pi}$, we write $\mathtt{B} = \max_{2 \le \mathtt{i} \le \mathtt{n}} \boldsymbol{\pi}[\mathtt{i}]$.

Definition 1. *In the **maximum no cover** array* $\mathtt{MNC} = \mathtt{MNC}[1..\mathtt{B}]$*, for every* $q \in 1..\mathtt{B}$, $\mathtt{MNC}[q] = q'$*, where* q' *is the maximum integer in* $1..q$ *such that the prefix* $\boldsymbol{x}[1..q']$ *has no cover — that is, such that* $\boldsymbol{\gamma}[q'] = 0$*.*

As shown in Figure 1, once B is computed in $\Theta(n)$ time from the prefix array $\boldsymbol{\pi}$, MNC can be easily computed in $\Theta(\mathtt{B})$ time using the cover array $\boldsymbol{\gamma}[1..\mathtt{B}]$ of $\boldsymbol{x}[1..\mathtt{B}]$. Note that the entries in MNC are monotone nondecreasing with $1 \le \mathtt{MNC}[q] \le q$ for every $q \in 1..\mathtt{B}$. The following is fundamental to the execution of our main algorithm:

Observation 2. *If a prefix* $\boldsymbol{v} = \boldsymbol{x}[1..q]$ *of* $\boldsymbol{x}$ *has a cover* $\boldsymbol{u}$*, then* $\boldsymbol{v} \neq \mathtt{mec}(\boldsymbol{x})$ *(since* $|\boldsymbol{u}| < q$ *and* $\boldsymbol{u}$ *covers every position covered by* $\boldsymbol{v}$*).*

```
procedure Compute_MNC($n, \boldsymbol{\pi}$; $\mathtt{B}, \boldsymbol{\gamma}, \mathtt{MNC}$)
B $\leftarrow \boldsymbol{\pi}[2]$
for $i \leftarrow 3$ to $n$ do
    B $\leftarrow \max(\mathtt{B}, \boldsymbol{\pi}[i])$
▷ Compute $\boldsymbol{\gamma}[1..\mathtt{B}]$ of $\boldsymbol{x}[1..\mathtt{B}]$ using
▷ the algorithm Compute_PCR of [2].
Compute_PCR($\mathtt{B}, \boldsymbol{\pi}$; $\boldsymbol{\gamma}$)
▷ Note that MNC can overwrite $\boldsymbol{\gamma}$.
for $q \leftarrow 1$ to B do
    if $\boldsymbol{\gamma}[q] = 0$ then MNC[$q$] $\leftarrow q$
    else MNC[$q$] $\leftarrow$ MNC[$q-1$]
```

Figure 1: Computing MNC from the prefix array $\boldsymbol{\pi}[1..n]$ and the cover array $\boldsymbol{\gamma}[1..\mathtt{B}]$.

Thus $\mathtt{MNC}[q]$ specifies an upper bound $q' \in 1..q$ on the length of a minimum enhanced is contained in a sequencecover of $\boldsymbol{x}$. Two other arrays are required for the computation, both of length B:

Definition 3. *For every* $q \in 1..\mathtt{B}$*:*

- $\mathtt{PR}[q]$ *is the rightmost position in* $\boldsymbol{x}$ *at which the prefix* $\boldsymbol{x}[1..q]$ *occurs;*
- $\mathtt{CPR}[q]$ *is the number of positions in* $\boldsymbol{x}$ *covered by occurrences of* $\boldsymbol{x}[1..q]$*.*

An example of the arrays introduced thus far is given in Figure 2. Note that for $\boldsymbol{x}[1..9]$ and $\boldsymbol{x}[1..10]$, there are actually *two* borders that cover a maximum number of positions; in each case the border of minimum length is identified in MEC.

The algorithm Compute_MEC is shown in Figure 3. In the first stage, B and MNC are computed and the arrays CMEC, PR and CPR are initialized. Then every position $i > 1$ such that $q = \boldsymbol{\pi}[i] > 0$ is considered. Using MNC, the longest prefix $\mathtt{Q'} = \boldsymbol{x}[1..q']$ of $\boldsymbol{x}[1..q]$ that does *not* have a cover is identified; for prefixes of $\boldsymbol{x}[1..q]$ that do have a

		1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x}$	=	a	b	a	b	a	a	b	a	b	a
$\boldsymbol{\pi}$	=	10	0	3	0	1	5	0	3	0	1
γ	=	0	0	0	2	3					
MNC	=	1	2	3	3	3					
PR	=	10	8	8	6	6					
CPR	=	6	8	10	8	10					
MEC	=	0	0	1	2	3	1	2	3	2	3
CMEC	=	0	0	2	4	5	4	6	8	8	10

Figure 2: All the arrays required to compute MEC and CMEC arrays

```
procedure Compute_MEC($\boldsymbol{\pi}$; MEC, CMEC)
$n \leftarrow |\boldsymbol{\pi}|$
Compute_MNC($n, \boldsymbol{\pi}$; B, $\gamma$, MNC)
MEC $\leftarrow 0^n$; CMEC $\leftarrow 0^n$; PR $\leftarrow 1^{B}$
for $q \leftarrow 1$ to B do CPR[$q$] $\leftarrow q$
for $i \leftarrow 2$ to $n$ do
    $q \leftarrow \boldsymbol{\pi}[i]$
▷ $\boldsymbol{x}[i..i+q-1] = \boldsymbol{x}[1..q]$.
    while $q > 0$ do
▷ $\boldsymbol{x}[1..q']$ is the longest prefix of $\boldsymbol{x}[1..q]$ without a cover.
        $q' \leftarrow$ MNC[$q$]
▷ $\boldsymbol{x}[1..q']$ also occurs at $i$: update CPR[$q'$] & PR[$q'$].
        if $i-$PR[$q'$] $< q'$ then
            CPR[$q'$] $\leftarrow$ CPR[$q'$]$+i-$PR[$q'$]
        else
            CPR[$q'$] $\leftarrow$ CPR[$q'$]$+q'$
        PR[$q'$] $\leftarrow i$
▷ Update MEC & CMEC accordingly for interval $i..i+q'-1$.
        if CPR[$q'$] $\geq$ CMEC[$i+q'-1$] then
            MEC[$i+q'-1$] $\leftarrow q'$
            if CPR[$q'$] $>$ CMEC[$i+q'-1$] then
                CMEC[$i+q'-1$] $\leftarrow$ CPR[$q'$]
        $q \leftarrow q'-1$
```

Figure 3: Computing MEC and CMEC from the prefix array $\boldsymbol{\pi}$.

cover, there is no need to compute the PR and CPR values. There are two main steps in the processing of Q′:

- Since i has now become the rightmost occurrence of Q′ in $\boldsymbol{x}[1..i]$, we must set PR[q'] $\leftarrow i$ and increment the corresponding number CPR[q'] of positions covered.
- If the number CPR[q'] of positions covered by occurrences of Q′ exceeds CMEC[$i+q'-1$], then CMEC and MEC must be updated accordingly.

These steps are repeated recursively for the longest proper prefix of $\mathtt{Q}'$ that does not have a cover.

3. Correctness & Complexity of Compute_MEC

We begin by proving the correctness of Compute_MEC, which depends on the prior computation of $\boldsymbol{\pi} = \boldsymbol{\pi}_{\boldsymbol{x}}$ [6]. Consider first procedure Compute_MNC, where $\mathtt{B}$ is computed, followed by the cover array $\boldsymbol{\gamma}[1..\mathtt{B}]$. Then for every $q \in 1..\mathtt{B}$, $\mathtt{MNC}[q] \leftarrow q$ whenever there is no cover of $\boldsymbol{x}[1..q]$, with $\mathtt{MNC}[q] \leftarrow \mathtt{MNC}[q{-}1]$ otherwise, an easy and straightforward calculation.

Compute_MEC then independently considers positions $i = 2, 3, \ldots, n$ for which $\boldsymbol{\pi}[\boldsymbol{i}] > 0$; that is, such that a border of $\boldsymbol{x}[1..i+q-1]$ of length $q = \boldsymbol{\pi}[i]$ begins at i. The internal **while** loop then processes in decreasing order of length the prefixes $\mathtt{Q}' = \boldsymbol{x}[1..q']$ of $\boldsymbol{x}[1..q]$ that have no cover — and that therefore, by Observation 2, can possibly be minimum enhanced covers of $\boldsymbol{x}[1..i{+}q'{-}1]$. Thus, for every $i \in 2..n$, all such borders $\boldsymbol{x}[1..q] = \boldsymbol{x}[i..i{+}q{-}1]$ are considered and, for each one, all such prefixes $\mathtt{Q}'$. For each q':

- the number $\mathtt{CPR}[q']$ of positions covered by $\mathtt{Q}'$ is updated, as well as the position $\mathtt{PR}[q'] = \mathtt{i}$ of rightmost occurrence of $\mathtt{Q}'$;
- $\mathtt{MEC}[i{+}q'{-}1]$ and $\mathtt{CMEC}[i{+}q'{-}1]$ are updated accordingly for sufficiently large $\mathtt{CPR}[q']$.

We claim therefore that

Theorem 4. *For a given string $\boldsymbol{x}$, Compute_MEC correctly computes the minimum enhanced cover array $\mathtt{MEC}_{\boldsymbol{x}}$ and the number $\mathtt{CMEC}_{\boldsymbol{x}}$ of positions covered by it, based solely on the prefix array $\boldsymbol{\pi}_{\boldsymbol{x}}$.*

We have seen that in aggregate Compute_MEC processes a subset of the nonempty borders of every prefix $\boldsymbol{x}[1..i]$, devoting $\mathcal{O}(1)$ time to each one. As we have seen, each border $\mathtt{Q}'$ in each such subset is constrained to have no cover. We say that a string $\boldsymbol{v}$ is ***strongly periodic*** if it has a border $\boldsymbol{u}$ such that $|\boldsymbol{u}| \geq |\boldsymbol{v}|/2$; otherwise $\boldsymbol{v}$ is said to be ***weakly periodic***. Observe that the borders $\mathtt{Q}'$ must all be weakly periodic; if not, then they would have a cover $\boldsymbol{u}$ with $|\boldsymbol{u}| \geq |\boldsymbol{v}|/2$. In [12] the following result is proved:

Lemma 5. *[12] There are at most $\log_2 n$ weakly periodic borders of a string of length n.*

It follows then that for each $i \in 2..n$, there are at most $\log_2 i$ borders considered, thus overall requiring $\mathcal{O}(n \log n)$ time.

The space requirement of Compute_MEC, apart from the $\boldsymbol{\pi}$, $\mathtt{MEC}$ and $\mathtt{CMEC}$ arrays, each of length n, consists of three integer arrays ($\mathtt{MNC}$ (overwriting $\boldsymbol{\gamma}$), $\mathtt{PR}$, $\mathtt{CPR}$), each of length $\mathtt{B}< n$. Thus

Theorem 6. *In the worst case, Compute_MEC computes* $\mathtt{MEC}$ *and* $\mathtt{CMEC}$ *from* $\boldsymbol{\pi}$ *using* (*a*) $\mathcal{O}(n \log n)$ *time;*

(*b*) *three additional arrays* $1..\mathtt{B}$ *of integers* $1..n$, *thus* $\Theta(\mathtt{B}\log n)$ *bits of space.*

Now we would like to analyze the expected (average) case behaviour of Compute_MEC. To do this, we prove and make use of a combinatorial result of independent interest. We first discuss this new interesting result in Section 3.1 and then we use it to complete the average case analysis in Section 3.2.

3.1. Combinatorics on the border length

Here we show that the expected length of the longest border of a string $\boldsymbol{x}$ approaches a limit as $|\boldsymbol{x}|$ tends to infinity, the limit depending on the alphabet size. For a binary alphabet it is approximately 1.64. We use the following notation: $\sigma = |\Sigma|$ is the alphabet size, $B(w)$ is the length of the longest border of string w, and $B_k(w)$ is the length of the longest border of string w which has length at most k (so ignoring any borders longer than k). Thus if $\boldsymbol{x} = babaabababbabaabab$, then $B(\boldsymbol{x}) = 8$, since $\boldsymbol{x}$ has longest border *babaabab*, and $B_4(\boldsymbol{x}) = 3$, since the longest border of $\boldsymbol{x}$ which has length at most 4 is *aba*. Let W_n be the set of all strings of length n on an alphabet of size σ. Since W_0 contains only the empty string, we have $|W_0| = 1$.

Lemma 7. *The number of strings of length n on an alphabet of size σ which have a border of length k (not necessarily the longest border) is σ^{n-k}.*
Proof. A string with border of length k is periodic with period $n-k$ and so is determined by its length $n-k$ prefix. This prefix can be chosen in σ^{n-k} ways. □

We also need the following formula (obtainable using a computer algebra system):

Lemma 8. $\sum_{m=k+1}^{n} m\sigma^{n-m} = \frac{\sigma^{n-k+1}k - \sigma^{n-k}k + \sigma^{n-k+1} - \sigma n + n - \sigma}{(\sigma-1)^2}$.

Clearly $|W_n| = \sigma^n$. The expected size of the longest border of a string of length n on an alphabet of size σ is therefore

$$\overline{B}(n) = \frac{1}{\sigma^n}\sum_{w\in W_n} B(w). \tag{1}$$

Similarly, the expected size of the longest border not exceeding k is

$$\overline{B}_k(n) = \frac{1}{\sigma^n}\sum_{w\in W_n} B_k(w). \tag{2}$$

Clearly $B(w) \geq B_k(w)$ so

$$\overline{B}(n) \geq \overline{B}_k(n). \tag{3}$$

Note that if $n \geq 2k$ then $W_n = \{uxv : u \in W_k, x \in W_{n-2k}, v \in W_k\}$ and so

$$\overline{B}_k(n) = \frac{1}{\sigma^n} \sum_{u \in W_k} \sum_{x \in W_{n-2k}} \sum_{v \in W_k} B_k(uxv). \tag{4}$$

Now $B_k(uxv) = B_k(uv)$ so if $n \geq 2k$,

$$\begin{aligned} \overline{B}_k(n) &= \frac{1}{\sigma^n} \sum_{u \in W_k} \sum_{v \in W_k} B_k(uv) \sum_{x \in W_{n-2k}} 1 \\ &= \frac{\sigma^{n-2k}}{\sigma^n} \sum_{u \in W_k} \sum_{v \in W_k} B_k(uv) \\ &= \frac{1}{\sigma^{2k}} \sum_{w \in W_{2k}} B_k(w) \\ &= \overline{B}_k(2k). \end{aligned} \tag{5}$$

With (3) we then have, for $n \geq 2k$,

$$\overline{B}(n) \geq \overline{B}_k(2k). \tag{6}$$

Now any border that is counted in the right hand side of (1) but not counted on the right hand side of (2) has length at least $k+1$. The sum of the lengths of such borders is, by Lemma 7,

$$\sum_{m=k+1}^{n} m\sigma^{n-m}.$$

So, by Lemma 8 and (5),

$$\begin{aligned} \overline{B}(n) &\leq \frac{1}{\sigma^n}\Big(\sum_{w \in W_n} B_k(w) + \sum_{m=k+1}^{n} m\sigma^{n-m}\Big) \\ &= \overline{B}_k(n) + \frac{1}{\sigma^n}\Big(\frac{\sigma^{n-k+1}k + \sigma^{n-k+1} - \sigma^{n-k}k - \sigma\, n - \sigma + n}{(\sigma-1)^2}\Big) \\ &< \bar{B}_k(2k) + \frac{\sigma^{-k+1}k + \sigma^{-k+1} - \sigma^{-k}k}{(\sigma-1)^2} \\ &= \overline{B}_k(2k) + O(k\sigma^{-k}). \end{aligned} \tag{7}$$

Thus for all $n \geq 2k$,

$$\overline{B}_k(2k) \leq \overline{B}(n) \leq \overline{B}_k(2k) + O(k\sigma^{-k}),$$

so $\overline{B}(n)$ is contained in a sequence of nested intervals whose lengths have limit 0. By the Nested Intervals Theorem this means the limit of $\overline{B}(n)$ exists.

Using (3) and (7) with $k = 11$ we find that $lim_{n\to\infty}\overline{B}(n)$ lies in the interval $(1.6356, 1.6420)$ for binary alphabets. For ternary alphabets using $k = 6$ the limit lies in $(0.6811, 0.6864)$.

3.2. Average case analysis

With the combinatorics of Section 3.1 at our disposal, we can easily complete the average case analysis of Compute_MEC. Clearly, this depends critically on the expected length of the maximum border of $\boldsymbol{x}[1..n]$; that is, the expected value of `B`. Now, from Section 3.1 we know that for a given alphabet size, `B` approaches a limit as n goes to infinity. The limit is approximately 1.64 for binary alphabets, 0.69 for ternary alphabets, and monotone decreasing in alphabet size. Thus

Theorem 9. *In the average case, Compute_MEC requires $\mathcal{O}(n)$ time and $\Theta(\log n)$ additional bits of space.*

4. Enhanced Left Covers and Left Seeds

In [12] the authors extended the concept of the enhanced cover to the notion of enhanced left covers and enhanced left seeds as follows. A proper prefix $\boldsymbol{u}$ of $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is an ***enhanced left cover*** (respectively, ***enhanced left seed***) of $\boldsymbol{x}$ if $\boldsymbol{u}$ has at least two occurrences in $\boldsymbol{x}$ and the number of positions in $\boldsymbol{x}$ that lie within occurrences of $\boldsymbol{u}$ in $\boldsymbol{x}$ (respectively, a right extension of $\boldsymbol{x}$) is the maximum over all such prefixes. For example, $\boldsymbol{x} = abaabab$ has enhanced left covers $\boldsymbol{u} = ab$ and aba, both covering six positions in $\boldsymbol{x}$, with both occurring twice in $\boldsymbol{x}$; but its only enhanced left seed is $\boldsymbol{u} = aba$, which lies *three* times in the right extension $\boldsymbol{x}a$ and so covers all seven positions of $\boldsymbol{x}$.

Thus, like the minimum enhanced cover array, we can analogously consider the minimum enhanced left cover (MELC) array and the minimum enhanced left seed (MELS) array. In fact, [12] provides an $O(n \log n)$ algorithm for computing the MELC array and an $O(n^2)$ algorithm for computing the MELS array. Both of these algorithms are extended from the border-based algorithm of [12] that computes `MEC`. In this section, we discuss how we can extend our (prefix-based) Compute_MEC algorithm to compute the MELC and MELS arrays.

4.1. Minimum Enhanced Left Cover Array (`MELC`)

We first consider the computation of the `MELC` array. As before, we also compute an associated array `CMELC`, analogous to the `CMEC` array; that is, `CMELC`$[i]$ counts the number of positions in $\boldsymbol{x}$ covered by the prefix of length `MELC`$[i]$. Here, for clarity of presentation, we describe the algorithm assuming that the `MEC` and `CMEC` arrays are already computed. However, in practice we will compute `MELC` and `CMELC` on the fly as part of the computation of `MEC` and `CMEC` in Compute_MEC. The essential thing to note is that for `MELC` we only need to consider a prefix rather than a border of $\boldsymbol{x}$. The central argument for the computation of the `MELC` array from the `MEC` and `CMEC` arrays is presented in Proposition 10 below. For this purpose, we need to define two functions, `MaxCount` and `CorLen`, that work in tandem on a prefix of `CMEC`, with `MaxCount` returning the maximum value in the prefix and `CorLen` returning the corresponding value in the `MEC` array. Importantly, in case of a tie for the maximum value, `CorLen` will return the value from the `MEC` array that is lower. For example, consider Figure 2 and suppose that we are considering `CMEC`[1..6]. Then `MaxCount` will return `CMEC`[3] = 5 and `CorLen` will return `MEC`[3] = 3. Notice that we have `CMEC`[8] = `CMEC`[9] = 8. So, if we consider `CMEC`[1..9], then `MaxCount` will return 8 and `CorLen` will return 2, because `MEC`[9] = 2 is lower than `MEC`[8] = 3.

Proposition 10. *Suppose* `MEC` *and* `CMEC` *have been computed for* $\boldsymbol{x}$. *Then* `MELC` *and* `CMELC` *are computed according to the following equations:*

$$\texttt{CMELC}[i] = \texttt{MaxCount}(\texttt{CMEC}[1..i]), \tag{8}$$

$$\texttt{MELC}[i] = \texttt{CorLen}(\texttt{CMEC}[1..i]). \tag{9}$$

To see that Proposition 10 is correct, we only need to recall that now the cover need not be a border and hence its last occurrence may end before the (prefix of the) string under consideration. Clearly this extra work does not change the asymptotic behaviour of the algorithm: Theorems 6 and 9 hold also for the computation of `MELC`.

4.2. Minimum Enhanced Left Seed Array (`MELS`*)*

The computation of the `MELS` array is more complicated. The complication arises because now not only are we looking at the prefix (as opposed to a border) but also considering a superstring to cover rather than the original string. To comprehend the new setting let us recall how Compute_MEC actually works: the heart of this algorithm is the **while** loop where we fix on a prefix q, then consider prefixes q' of q that might possibly cover it. Note that during this **while** loop we remain on a particular index i and we only update index $i + q' - 1$ of the `MEC` and `CMEC` arrays, where q' is the length of the prefix we are considering. This works fine when the prefix under consideration also has to be a border, because then we know that it must occur at the end aligning with the end of the prefix of the string under consideration. But for a left seed, more work is required. Now we are interested in the interval $i..i+q'-1$. Clearly, for the index $i+q'-1$ — that is, for the string $\boldsymbol{x}[1..i+q'-1]$ — the occurrence of the prefix under consideration — that is, the prefix of length q' — is also a border. But for $\boldsymbol{x}[1..i+\ell-1]$ with $q' < \ell \leq B$, $\boldsymbol{x}[1..i+\ell-1]$ is a superstring and the prefix of length ℓ is an enhanced left seed. Therefore, we need to update `MELS` and `CMELS` for $i + q' - 1$ based on which prefix covers most, whereas in Compute_MEC we only need to update the index $i + q' - 1$ with only one prefix.

The correctness of Compute_MELS (Figure 4) readily follows from the above discussion. However, its running time is increased due to the newly introduced inner **while** loop within the outer **while** loop. Recalling the time complexity analysis of Compute_MEC, we realize that the only change between the two algorithms is that in Compute_MEC for each prefix considered we need to do the update on only one index (that is, index $i + q' - 1$) with one prefix q'; whereas in Compute_MELS, we need to update one index while considering all the prefixes larger than q' (**while** $mpf > q'$). Notice that each prefix of length ℓ for a particular substring $\boldsymbol{x}[1..i+\ell-1]$ is considered ℓ times, since the substring can be a superstring of $\ell - 1$ substrings of $\boldsymbol{x}$. There are $\mathcal{O}(\log n)$ of weakly periodic prefixes of a string each of which can at most be equal to `B`. Therefore, a straightforward analysis gives us a running time of

$\mathcal{O}(n\mathtt{B}\ \log n)$ for Compute_MELS. This ensures that the running time remains linear in the average case:

Theorem 11. *In the average case, Compute_MELS requires* $\mathcal{O}(n)$ *time and* $\Theta(\log n)$ *additional bits of space.*

However, a more careful analysis of the worst case running time of Compute_MELS can be performed as follows. We have already used the fact from [12] that there are at most $\log_2 n$ weakly periodic borders of a string of length n (Lemma 5). However, this is a consequence of the fact that a weakly periodic border of a string of length n can be of size at most $n/2$. Now recall that we are only handling weakly periodic prefixes in Compute_MELS in the inner **while** loop. And the number of superstrings involved with each weekly periodic prefix is the length of the current prefix under consideration. Summing the lengths of these prefixes yields a geometric series up to $\log n$ that adds up to $O(n)$. This implies that the total work done by the **for** loop within the **while** loop during the complete execution of the algorithm remains $O(n)$. Hence:

Theorem 12. *In the worst case, Compute_MELS computes* `MELS` *and* `CMELS` *from* $\boldsymbol{\pi}$ *using* $\mathcal{O}(n^2)$ *time and* $\Theta(\mathtt{B}\ \log n)$ *bits of space.*

5. Comparing Border-Based and Prefix-Based Algorithms

As mentioned above, in order to compute $\mathtt{MEC}_{\boldsymbol{x}}$, the authors of [12] made use of the ***border array***. On the other hand Compute_MEC is based on the ***prefix table***. As we have seen, Compute_MEC requires only three additional arrays $1..B$ of integers, compared to $4n$ for the algorithm of [12]. In the next section we will see how use of the prefix table enables Compute_MEC to be extended to indeterminate strings, not a possibility for a border-based algorithm. Here we compare the time requirements of the two algorithms, referring to our algorithm as ECP and to the border-based algorithm as ECB.

We implemented ECP in C# using Visual Studio 2010. We got the implementation of ECB from the authors of [12]. However, ECB was implemented in C. To ensure a level playing ground, we re-implemented ECB in C# following their implementation. Then we ran both the algorithms on all binary strings of lengths 2 to 30. The experiments were carried out on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. The results are illustrated in Figures 5 and 6.

Figure 5 shows the maximum number of operations (assignment, comparison, etc.) carried out by each algorithm. Figure 6 shows the ratio of the total number of operations performed by ECB and ECP to the length n of the string, over all strings on the binary alphabet. As is evident from the figures, ECP outperforms ECB and in fact it shows linear behaviour, verifying the claim in Theorem 9 above.

6. Indeterminate Strings

In Sections 2 and 3 we describe an algorithm to compute the minimum enhanced cover array $\mathtt{MEC}_{\boldsymbol{x}}$ of a given string $\boldsymbol{x}$, based only on the prefix array $\boldsymbol{\pi}_{\boldsymbol{x}}$. Then we have extended this algorithm for minimum enhanced left cover array and minimum enhanced left seed array in Section 4. As noted in the Introduction, since the prefix array can be computed also for indeterminate strings [22], this immediately raises the possibility of extending the MEC calculation to indeterminate strings.

In [2] two definitions of "cover" for an indeterminate string are proposed: a ***sliding cover*** where adjacent or overlapping covering substrings of $\boldsymbol{x}$ must match, and a ***rooted cover*** where each covering substring is constrained only to match a prefix of $\boldsymbol{x}$. The nontransitivity of matching (see Section 1) inhibits implementation of a sliding cover, but [2] shows how to compute all the rooted covers of indeterminate $\boldsymbol{x}$ from its prefix array in $\mathcal{O}(n^2)$ worst case time, $\Theta(n)$ in the average case. Thus it becomes possible to execute Compute_MNC for rooted covers, simply by replacing the function call to Compute_PCR by a function call to PCInd of [2]; that is, to compute the rooted cover array $\boldsymbol{\gamma}_{\boldsymbol{R}}[1..\mathtt{B}]$, hence $\mathtt{MNC}[1..\mathtt{B}]$ and thus $\mathtt{MEC}_{\boldsymbol{x}}$, all for indeterminate strings. Let us call this new algorithm Compute_MEC_Ind. We recall now a lemma from [5] stating that the expected number of borders in an indeterminate string is bounded above by a constant, approximately 29. Therefore, also for indeterminate strings, B can be treated as a constant, and we have the following remarkable result:

Theorem 13. *In the average case, Compute_MEC_Ind requires $\mathcal{O}(n)$ time and $\Theta(\log n)$ additional bits of space.*

Clearly, these results can be similarly extended for the MELC and MELS arrays to indeterminate strings.

7. Conclusion

In this paper we have described prefix array based algorithms to compute minimum enhanced cover arrays, minimum enhanced left cover arrays and minimum enhanced left seed arrays. The advantages of our prefix array based algorithms are threefold. Firstly, our prefix-array based algorithms exhibit the same worst case running time as the border-based algorithms of [12] but our experimental results suggest that the former are faster in practice. Secondly, our algorithms exhibit superior space efficiency. And finally, because of the robustness of the prefix array, our algorithms, in addition to being potentially faster in practice and more space-efficient than those of [12], allow us to easily extend the computation of enhanced covers to indeterminate strings. Additionally, both for regular and indeterminate strings, our algorithms execute in expected linear time. We have also established an important theoretical result which we believe is of independent interest: that the expected maximum length of any border of any prefix of a regular string $\boldsymbol{x}$ is approximately 1.64 for binary alphabets, less for larger ones.

We note further that the prefix array can be efficiently computed in a compressed form [22], taking advantage of the fact that for $i \in 1..n$, $\boldsymbol{\pi}[i] \neq 0$ if and only if $\boldsymbol{x}[i] = \boldsymbol{x}[1]$. Thus we can use two arrays `POS` and `LEN` to store nonzero positions in $\boldsymbol{\pi}$ and the values at those positions, respectively, thus saving much space in cases that arise in practice. We plan to design a `POS`/`LEN` version of Compute_MEC as an immediate future work. Another natural question of course is to investigate whether the `MEC` array can be computed in worst-case linear time.

Acknowledgement

We thank two referees for helpful comments that have materially improved the paper.

References

[1] Ali Alatabbi, M. Sohel Rahman & W. F. Smyth, **Inferring an indeterminate string from a prefix graph**, *J. Discrete Algorithms* (2014) to appear.

[2] Ali Alatabbi, M. Sohel Rahman & W. F. Smyth, **Computing covers using prefix tables**, `http://arxiv.org/abs/1412.3016`, *Discrete Appl. Maths.* (2015) to appear.

[3] Alberto Apostolico & Andrzej Ehrenfeucht, **Efficient Detection of Quasiperiodicities in Strings**, Tech. Report No. 90.5, The Leonardo Fibonacci Institute, Trento, Italy (1990).

[4] Alberto Apostolico, Martin Farach & C. S. Iliopoulos, **Optimal superprimitivity testing for strings**, *Inform. Process. Lett. 39-1* (1991) 17-20.

[5] Md. Faizul Bari, Mohammad Sohel Rahman, Rifat Shahriyar, **Finding All Covers of an Indeterminate String in $O(n)$ Time on Average**. *Proc. Prague Stringology Conference* (2009) 263–271

[6] Widmer Bland, Gregory Kucherov & W. F. Smyth, **Prefix table construction & conversion**, *Proc. 24th IWOCA*, Springer Lecture Notes in Computer Science LNCS 8288 (2013) 41–53.

[7] Francine Blanchet-Sadri, **Algorithmic Combinatorics on Partial Words**, Chapman & Hall/CRC (2008) 385 pp.

[8] D. Breslauer, **An on-line string superprimitivity test**, *Inform. Process. Lett. 44-6* (1992) 345-347.

[9] Manolis Christodoulakis, P, J. Ryan, W. F. Smyth & Shu Wang, **Indeterminate strings, prefix arrays & undirected graphs**, *Theoret. Comput. Sci.* (2015) to appear.

[10] Richard Cole, C. S. Iliopoulos, Manal Mohamed, W. F. Smyth & Lu Yang, **The complexity of the minimum k-cover problem**, *J. Automata, Languages & Combinatorics 10–5/6* (2005) 641–653.

[11] Michael J. Fischer & Michael S. Paterson, **String-matching and other products**, *Complexity of Computation, Proc. SIAM-AMS 7* (1974) 113-125.

[12] Tomáš Flouri, C. S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, W. F. Smyth & Wojciech Tyczyński, **Enhanced string covering**, *Theoret. Comput. Sci. 506* (2013) 102–114.

[13] Jan Holub & W. F. Smyth, **Algorithms on indeterminate strings**, *Proc. 14th Australasian Workshop on Combinatorial Algs.* (2003) 36–45.

[14] C. S. Iliopoulos, Manal Mohamed & W. F. Smyth, **New complexity results for the k-covers problem**, *Inform. Sciences 181* (2011) 2571–2575.

[15] C. S. Iliopoulos & W. F. Smyth, **On-line algorithms for k-covering**, *Proc. Ninth Australasian Workshop on Combinatorial Algs.* (1998) 64–73.

[16] T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter & T. Walen, **Efficient Algorithms for Shortest Partial Seeds in Words**, *Proc. Annual Symp. Combinatorial Pattern Matching* (2014) 192–201.

[17] T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter & T. Walen, **Fast Algorithm for Partial Covers in Words**, *Algorithmica 73–1* (2015) 217–233.

[18] Yin Li & W. F. Smyth, **An optimal on-line algorithm to compute all the covers of a string**, *Algorithmica 32–1* (2002) 95–106.

[19] Dennis Moore & W. F. Smyth, **An optimal algorithm to compute all the covers of a string**, *Inform. Process. Lett. 50* (1994) 239-246.

[20] Dennis Moore & W. F. Smyth, **Correction to: An optimal algorithm to compute all the covers of a string**, *Inform. Process. Lett. 54* (1995) 101-103.

[21] Bill Smyth, **Computing Patterns in Strings**, Pearson Addison-Wesley (2003) 423 pp.

[22] W. F. Smyth & Shu Wang, **New perspectives on the prefix array**, *Proc. 15th String Processing & Inform. Retrieval Symp.*, Springer Lecture Notes in Computer Science LNCS 5280 (2008) 133–143.

[23] W. F. Smyth & Shu Wang, **A new approach to the periodicity lemma on strings with holes**, *Theoret. Comput. Sci. 410–43* (2009) 4295–4302.

[24] W. F. Smyth & Shu Wang, **An adaptive hybrid pattern-matching algorithm on indeterminate strings**, *Internat. J. Foundations of Computer Science 20–6* (2009) 985–1004.

```
procedure Compute_MELS($\boldsymbol{\pi}$; MELS, CMELS)
$n \leftarrow |\boldsymbol{\pi}|$
Compute_MNC($n, \boldsymbol{\pi}$; B, $\boldsymbol{\gamma}$, MNC)
MEC $\leftarrow 0^n$; CMEC $\leftarrow 0^n$; PR $\leftarrow 1^{\mathtt{B}}$
for $q \leftarrow 1$ to B do CPR[$q$] $\leftarrow q$
for $i \leftarrow 2$ to $n$ do
    $q \leftarrow \boldsymbol{\pi}[i]$
▷ $\boldsymbol{x}[i..i+q-1] = \boldsymbol{x}[1..q]$.
    while $q > 0$ do
▷ $\boldsymbol{x}[1..q']$ is the longest prefix of $\boldsymbol{x}[1..q]$ without a cover.
        $q' \leftarrow$ MNC[$q$]
▷ $\boldsymbol{x}[1..q']$ also occurs at $i$: update CPR[$q'$] & PR[$q'$].
        if $q' = q$ then
            if $i -$ PR[$q'$] $< q'$ then
                CPR[$q'$] $\leftarrow$ CPR[$q'$] $+ i -$ PR[$q'$]
            else
                CPR[$q'$] $\leftarrow$ CPR[$q'$] $+ q'$
        else
            $q' \leftarrow q$
        $mpf \leftarrow B$
        while $mpf > q'$ do
            $mp \leftarrow MNC[mpf]$
            if $mp = q'$ then
                $break$
            if PRS[$mp$] $> 1$ && $i + q' >$ PRS[$mp$] $+ mp$ then
                if $i -$ PR[$mp$] $< mp$ then
                    $S_1 \leftarrow$ CPR[$mp$] $- mp + i -$ PR[$mp$] $+ q'$
                else
                    $S_1 \leftarrow$ CPR[$mp$]$+q'$
            if $S_1 \geq S_p$ then
                $maxp \leftarrow mp$
                $S_p \leftarrow S_1$
            $mpf \leftarrow mp - 1$
▷ Update $S_2$ and $q''$ depending on Maximum of $S_p$ and CPR[$q'$]
        $S_2 \leftarrow S_p$ or CPR[$q'$]
        $q'' \leftarrow maxp$ or $q'$
▷ Update CMELS & MELS accordingly.
        if $S_2 \geq$ CMELS[$i+q'-1$] then
            MELS[$i+q'-1$] $\leftarrow q''$
            if $S_2 >$ CMELS[$i+q'-1$] then
                CMELS[$i+q'-1$] $\leftarrow S_2$
        if PR[$q'$] $= 1$ then
            PRS[$q'$] $\leftarrow i$
        PR[$q'$] $\leftarrow i$
        $q \leftarrow q-1$
```

Figure 4: Computing MELS and CMELS from the prefix array $\boldsymbol{\pi}$.

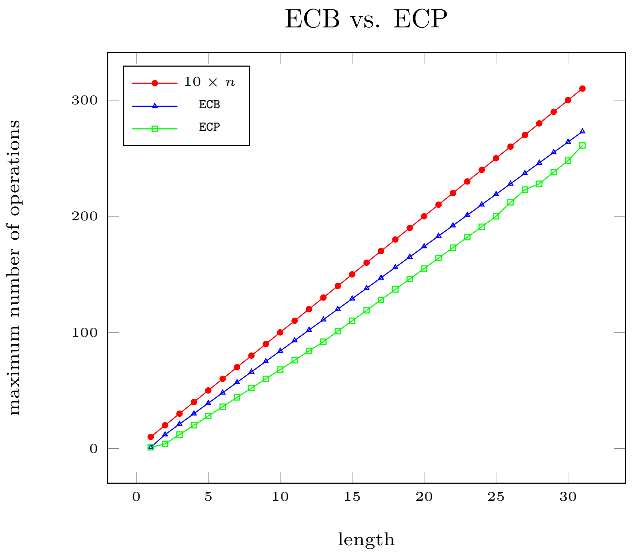

Figure 5: The maximum number of operations performed by the Border-Based (ECB) [12] and Prefix-Based (ECP) algorithm (i.e., Compute_MEC) to compute the Minimum Enhanced Cover array, for all strings on the binary alphabet.

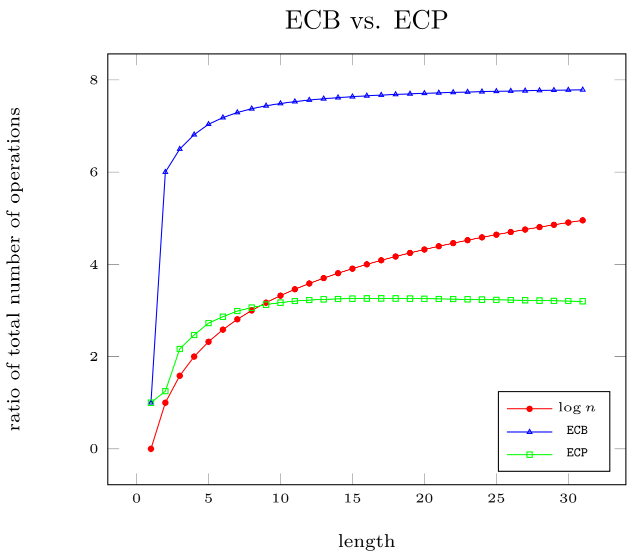

Figure 6: Ratio of the ***total*** number of operations performed by the Border-Based (ECB) [12] and Prefix-Based (ECP) algorithms to the length n of the string, for all strings on the binary alphabet. Note the linear behaviour of ECP compared to the supralinear behaviour of ECB.