Fast, Practical Algorithms for Computing All the Repeats in a String

Simon J. Puglisi, W. F. Smyth and Munina Yusufu

Abstract. Given a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet of size α, and a threshold $p_{min} \geq 1$, we describe four variants of an algorithm PSY1 that, using a suffix array, computes all the complete nonextendible repeats in $\boldsymbol{x}$ of length $p \geq p_{min}$. The basic algorithm PSY1–1 and its simple extension PSY1–2 are fast on strings that occur in biological, natural language and other applications (not highly periodic strings), while PSY1–3 guarantees $\Theta(n)$ worst-case execution time. The final variant, PSY1–4, also achieves $\Theta(n)$ processing time and, over the complete range of strings tested, is the fastest of the four. The space requirement of all four algorithms is about $5n$ bytes, but all make use of the "longest common prefix" (LCP) array, whose construction requires about $6n$ bytes. The four algorithms are faster in applications and use less space than a recently-proposed algorithm [17] that produces equivalent output. The suffix array is not explicitly used by algorithms PSY1, but may be required for postprocessing; in this case, storage requirements rise to $9n$ bytes. We also describe two variants of a fast $\Theta(n)$-time algorithm PSY2 for computing all complete supernonextendible repeats in $\boldsymbol{x}$.

Mathematics Subject Classification (2010). Nonnumerical Algorithms 68W05.

Keywords. Repeat, Repetition, Suffix Array, Suffix Tree.

1. Introduction

A **repeating substring** $\boldsymbol{u}$ in a string $\boldsymbol{x}$ is a substring of $\boldsymbol{x}$ that occurs more than once. A **repeat** in $\boldsymbol{x}$ is a set of locations in $\boldsymbol{x}$ at which a repeating substring $\boldsymbol{u}$ occurs; it can be specified by the length $p \geq 1$ of $\boldsymbol{u}$ (what we

Address for correspondence: Munina Yusufu, Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada L8S 4K1, yusufum@mcmaster.ca.

The work of the first author is supported by the Australian Research Council. The work of the second and third authors was supported in part by grants from the Natural Sciences & Engineering Research Council of Canada. A preliminary version of this paper appeared in [19]. The authors thank an anonymous referee whose comments have materially improved this paper.

call its **period**) and the locations. For example, in $\boldsymbol{x} = abaababa$, ab is a repeating substring which occurs three times at positions 1, 4, and 6 with length 2. Thus the tuple $(2;\ 1,4,6)$ describes the repeat of $\boldsymbol{u} = ab$ $(p = 2)$ at positions 1, 4, 6.

Following [22] we say that a repeat $(p;\ i_1, i_2, \ldots, i_k)$, $k \geq 2$, is **complete** iff it includes all occurrences of $\boldsymbol{u}$ in $\boldsymbol{x}$; **left-extendible** (LE) iff

$$\boldsymbol{x}[i_1{-}1] = \boldsymbol{x}[i_2{-}1] = \cdots = \boldsymbol{x}[i_k{-}1];$$

and **right-extendible** (RE) iff

$$\boldsymbol{x}[i_1{+}p] = \boldsymbol{x}[i_2{+}p] = \cdots = \boldsymbol{x}[i_k{+}p].$$

A repeat is NLE iff it is not LE; NRE iff it is not RE; **nonextendible** (NE) iff it is both NLE and NRE. A repeat is **supernonextendible** (SNE) iff it is NE and its repeating substring $\boldsymbol{u}$ is not a proper substring of any other repeating substring of $\boldsymbol{x}$.

Detecting repeats is important in applications such as bioinformatics and data compression, as well as in musicology (identification of repeating rhythms or passages) and software engineering (clone detection in large software systems).

In [8, p. 147] an algorithm is described that, given the suffix tree $\mathrm{ST}_{\boldsymbol{x}}$ of $\boldsymbol{x}$, computes all the NE (called "maximal") *pairs of repeats* in $\boldsymbol{x}$ in time $O(\alpha n{+}q)$, where q is the number of pairs output. [4] uses similar methods to compute all NE pairs $(p; i_1, i_2)$ such that $i_2 - i_1 \geq g_{min}$ (or $\leq g_{max}$) for user-defined **gaps** g_{min}, g_{max}. [1] shows how to use the suffix array $\mathrm{SA}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ to compute the NE pairs in time $O(\alpha n{+}q)$. Since it may be that $\alpha \in O(n)$, all of these algorithms require $O(n^2)$ time in the worst case, though in many applications $\alpha = 4$ (DNA alphabet). [7] uses the suffix arrays of both $\boldsymbol{x}$ and its reversed string $\overline{\boldsymbol{x}} = \boldsymbol{x}[n]\boldsymbol{x}[n{-}1]\cdots\boldsymbol{x}[1]$ to compute all the complete NE repeats in $\boldsymbol{x}$ in $\Theta(n)$ time. More recently, [17] describes suffix array-based $\Theta(n)$-time algorithms to compute all **substring equivalence classes** — including the complete NE repeats — in $\boldsymbol{x}$.

In this paper we present four variants of a new fast algorithm PSY1 that computes all the complete NE repeats in a given string $\boldsymbol{x}$ whose length (period) $p \geq p_{min}$, where $p_{min} \geq 1$ is a user-specified minimum. PSY1 uses $5n$ bytes of space directly, but requires the LCP array, whose construction needs $6n$ bytes. The variants PSY1–3 and PSY1–4 execute in $\Theta(n)$ time independent of alphabet size.

We also describe two versions of a new linear-time algorithm PSY2 to compute all the SNE repeats in $\boldsymbol{x}$. The second of these, PSY2–2, executes in time $\Theta(n{+}\alpha)$. This improves on the algorithm described in [8, p. 146] that does the same calculation (of "supermaximal" repeats) in time $O(n \log \alpha)$ using a suffix tree, as well as on the algorithm described in [1, p. 59] that uses a suffix array and requires $O(n{+}\alpha^2)$ time. For $\alpha \in O(n)$ these times become $O(n \log n)$ and $O(n^2)$, respectively, whereas PSY2–2 remains $\Theta(n)$. We remark that in many applications SNE repeats are those of greatest interest.

In Sections 2 and 3 we describe our algorithms for computing, respectively, nonextendible and supernonextendible repeats. Section 4 summarizes the results of experiments that compare the algorithms with each other and with existing algorithms. Section 5 discusses these results, including the strategy of computing complete (NE and SNE) repeats in the context of applications to bioinformatics.

2. Algorithms for Nonextendible Repeats

We suppose that a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is given, defined on an ordered alphabet A of size α (where if there is no explicit bound on alphabet size, we suppose $\alpha \leq n$). We refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as **suffix** i. Then the **suffix array** $\mathrm{SA}_{\boldsymbol{x}}$ is an array $[1..n]$ in which $\mathrm{SA}_{\boldsymbol{x}}[j] = i$ iff suffix i is the j^{th} in lexicographical order among all the nonempty suffixes of $\boldsymbol{x}$. Let $\mathrm{lcp}_{\boldsymbol{x}}(i_1, i_2)$ denote the **longest common prefix** of suffixes i_1 and i_2 of $\boldsymbol{x}$. Then $\mathrm{LCP}_{\boldsymbol{x}}$ is an array $[1..n{+}1]$ in which $\mathrm{LCP}_{\boldsymbol{x}}[1] = \mathrm{LCP}_{\boldsymbol{x}}[n{+}1] = -1$, while for $j \in 2..n$,

$$\mathrm{LCP}_{\boldsymbol{x}}[j] = \Big|\mathrm{lcp}_{\boldsymbol{x}}\big(\mathrm{SA}_{\boldsymbol{x}}[j{-}1], \mathrm{SA}_{\boldsymbol{x}}[j]\big)\Big|.$$

$\mathrm{SA}_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ worst-case time [9, 12], though various supralinear methods [16, 14] are certainly much faster, as well as more space-efficient, in practice [18], in some cases requiring space only for $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$ itself. Given $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$, $\mathrm{LCP}_{\boldsymbol{x}}$ can also be computed in $\Theta(n)$ time [11, 15, 20]: the first algorithm described in [15] requires $9n$ bytes of storage and is almost as fast in practice as that of [11], which requires $13n$ bytes. However the algorithm recently proposed in [20] is generally faster and requires about $6n$ bytes of storage for its execution, since it overwrites the suffix array — this is the LCP algorithm used in this paper. (For space calculations, we make throughout the usual assumption that an integer occupies four bytes, a letter one.) When the context is clear, we write SA for $\mathrm{SA}_{\boldsymbol{x}}$, LCP for $\mathrm{LCP}_{\boldsymbol{x}}$.

We also define the Burrows-Wheeler Transform $\mathrm{BWT}_{\boldsymbol{x}}$ or BWT [5]: for $\mathrm{SA}[j] > 1$, $\mathrm{BWT}[j] = \boldsymbol{x}\big[\mathrm{SA}[j]{-}1\big]$, while for j such that $\mathrm{SA}[j] = 1$, $\mathrm{BWT}[j] =$ \$, a sentinel letter not equal to any other in $\boldsymbol{x}$. We set $\mathrm{BWT}[n{+}1] =$ \$. BWT can clearly be computed in linear time from SA; since it occupies only n rather than $4n$ bytes, we use BWT rather than SA if there is a choice. Figure 1 gives the SA, LCP, and BWT arrays for the string $\boldsymbol{x} = abababababab$. We will use this example in the following sections as well.

	1	2	3	4	5	6	7	8	9	10	11
$\boldsymbol{x} =$	a	b	a	b	a	b	a	b	a	b	\$
$\mathrm{SA}_{\boldsymbol{x}} =$	9	7	5	3	1	10	8	6	4	2	
$\mathrm{LCP}_{\boldsymbol{x}} =$	-1	2	4	6	8	0	1	3	5	7	-1
$\mathrm{BWT}_{\boldsymbol{x}} =$	b	b	b	b	\$	a	a	a	a	a	\$

FIGURE 1. String $\boldsymbol{x} = abababababab$ with SA, LCP and BTW arrays

As explained below, there are altogether nine repeats in $\boldsymbol{x}$. As defined in the Introduction, each repeat can be presented in the form $(p;\ i_1, i_2, \ldots, i_k)$, where p is the period of the repeat and $i_1, i_2, \ldots, i_k$ are positions in which the repeating substring occurs. For example, the repeating substring $\boldsymbol{u}= bab$ of length 3 occurs in positions $2, 4, 6, 8$ of $\boldsymbol{x}$, so the repeat can be described as $(3; 2, 4, 6, 8)$. Note that our algorithms report this fact as a complete repeat in the form $(3; 7, 10)$ with period $p = 3$, where $7, 10$ is a range identifying $\mathrm{SA}[7] = 8, \mathrm{SA}[8] = 6, \mathrm{SA}[9] = 4, \mathrm{SA}[10] = 2$. For convenience, we may refer to repeat $(3; 7, 10)$ as *bab*, or as $(3; 7, 10) = bab$ depending on the context in the following sections. Note that, among the nine repeats in $\boldsymbol{x}$, $a, ab, abab$, *ababab*, and *abababab* are NLE; since *abababab* is also NRE, it is NE, and it is the only SNE repeat in $\boldsymbol{x}$. The other four repeats b, bab, *babab*, *bababab* are LE.

The PSY1 and PSY2 algorithms described below make direct use of LCP and BWT (but neither of SA nor of $\boldsymbol{x}$ itself), and therefore require only $5n$ bytes of storage (plus relatively small stack space in the case of PSY1). For these algorithms, then, the $6n$ bytes needed for LCP construction (see above) provide an upper bound on the overall space requirement. PSY1 and PSY2 output ranges $i..j$ of positions in SA that specify complete repeats (NE for PSY1, SNE for PSY2). See Section 5 for further discussion of the postprocessing of PSY1/PSY2 output.

PSY1–1

```
    — Preprocessing: compute SA, BWT & LCP
    — in $\Theta(n)$ time (LCP overwrites SA).
  $j \leftarrow 0;\ p \leftarrow -1;\ q \leftarrow 0;\ prevNE \leftarrow 0;$ push(LB; 0, 0)
  while $j < n$ do
      repeat
          $j \leftarrow j+1;\ p \leftarrow q;\ q \leftarrow \mathrm{LCP}[j+1]$
          if $q > p$ and $q \geq p_{min}$ then push(LB; $j, q$)
      until $p > q$
      repeat
          if top(LB).$lcp > 0$ then
              $(i, p) \leftarrow$ pop(LB)
              if $prevNE \geq i$ then output($p; i, j$)
              elsif NLE($i, j$) then $prevNE \leftarrow i$; output($p; i, j$)
      until top(LB).$lcp \leq q$
      if top(LB).$lcp < q$ and $q \geq p_{min}$ then
          push(LB; $i, q$)
```

FIGURE 2. Algorithm PSY1–1 — compute all NE repeats of period $p \geq p_{min}$ as ranges in SA using one stack

Algorithms PSY1 take as a point of departure the algorithm described in [1] that, based on $\mathrm{SA}_{\boldsymbol{x}}$, outputs repeating substrings of $\boldsymbol{x}$ in pairs. Given a threshold $p_{min} \geq 1$, PSY1 by contrast outputs complete NE repeats, each one a triple $(p; i, j)$ specifying a period $p \geq p_{min}$ and a range $i..j$ in $\mathrm{SA}_{\boldsymbol{x}}$. Thus the range implicitly reports a complete repeat as defined in Section 1.

These algorithms all perform a single left-to-right scan of LCP. Figures 2 and 3 show pseudocode for a brute force (but fast) version PSY1–1 that uses a single stack LB (Left Boundary) to store leftmost positions i at which there is an increase in the LCP value. More precisely, entries (j, q) are pushed onto LB at every position j at which the LCP value increases (to q) and popped whenever the LCP value decreases. Thus each pop, together with the current j value, identifies a complete repeat that must be NRE but that may or may not be NLE. Since the expected maximum length of a repeating substring in a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet of size α is $2\log_\alpha n$ [10], this quantity is also the expected maximum number of entries in LB; in the worst case ($\boldsymbol{x} = a^n$), the maximum could be $\Theta(n)$. Note that since there is at most one output for each pop of LB, the number of repeats, thus in particular the number of complete NE repeats, in $\boldsymbol{x}$ is $O(n)$ (at most the number of internal nodes in the suffix tree).

```
function NLE($i, j$)
  — Range is LE only if all preceding letters are identical.
$\lambda \leftarrow$ BWT[$i$]; $i' \leftarrow i+1$
while $i' \leq j$ and $\lambda =$ BWT[$i'$] do $i' \leftarrow i'+1$
return ($i' \leq j$)
```

FIGURE 3. Determine whether the repeat SA$[i..j]$ is NLE (one stack)

An important feature of PSY1–1 is the use of the variable *prevNE* — this is the lefthand position i in SA of the repeat $(p; i, j)$ most recently discovered, through an invocation of function NLE, to be in fact non-left-extendible. The importance of *prevNE* derives from the observation that if any repeat of a substring $\boldsymbol{u}$ is NLE, then so is the complete repeat of any prefix of $\boldsymbol{u}$ — thus storing *prevNE* ensures that the upward propagation of the NLE property in the suffix tree of $\boldsymbol{x}$ is recognized without invoking function NLE unnecessarily. For example, for the string $\boldsymbol{x}$ in Figure 1, after the repeat $(8; 4, 5) = abababab$ was popped, since it is NLE, *prevNE* will be set to 4; then when the following repeat $(6; 3, 5) = ababab$ (note that $ababab$ is a prefix of $abababab$) is popped, since $i = 3 < 4 = prevNE$, it will be directly output without being checked by function NLE; similarly for the following repeats: $(4; 2, 5) = abab$, $(2; 1, 5) = ab$, and $(1; 1, 5) = a$.

In practice, this heuristic greatly reduces the time requirement. It is interesting that, apart from highly periodic strings (that rarely occur in practice), PSY1–1 is the fastest of the four variants on the strings used for testing.

However, PSY1–1 is not linear in string length n in the worst case. For integer $k \geq 1$, $n = 8k+2$, $\boldsymbol{x} = (ab)^{n/2}$, every repeat of $b, bab, \ldots, b(ab)^{n/2-2}$ is LE, requiring $n/2, n/2-1, \ldots, n/2-(n/2-2)$ positions of SA, respectively, to be checked by function NLE, a total of

$$\sum_{i=0}^{n/2-2} (n/2-i) = (n/2+2)(n/2-1)/2 \in \Theta(n^2)$$

letter comparisons. More generally, every string $\boldsymbol{x} = \boldsymbol{u}^{n/k}$, where k is a constant that divides n, will require $\Theta(n^2)$ letter comparisons.

For example, for the string $\boldsymbol{x}$ in Figure 1, after the repeat $(7; 9, 10) =$ *bababab* was popped, it will be checked by function NLE to determine whether BWT[9] = BWT[10]; since it is LE, the following repeat $(5; 8, 10) =$ *babab* will also be checked by function NLE to see whether BWT[8], BWT[9], and BWT[10] are equal. We notice that redundant letter comparisons occur here. Repeats $(3; 7, 10) =$ *bab* and $(1; 6, 10) = b$ will be processed analogously.

PSY1–2

To avoid repetitive checking of LE repeats, we introduce two integer variables, *leftLE* and *rightLE*, that identify the left and right boundaries, respectively, of the repeat (range in SA) most recently found to be LE in the left-to-right scan of LCP. In the event that $leftLE..rightLE$ is a subrange of a range $i..j$ whose LE status needs to be checked, this change allows the LE subrange to be skipped.

Assume that a particular repeat $(p; i'..j')$ is LE. Then

$$\boldsymbol{x}[\mathrm{SA}[i'] - 1] = \boldsymbol{x}[\mathrm{SA}[i' + 1] - 1] = ... = \boldsymbol{x}[\mathrm{SA}[j'] - 1].$$

If we set $i' \leftarrow leftLE$, $j' \leftarrow rightLE$, then for the subsequent complete NRE repeat $(\widetilde{p}; i, j)$ such that $i \le i' < j' \le j$, we also need to check whether or not it is NLE; that is, to check whether or not there exists at least one pair in the letters

$$\boldsymbol{x}[\mathrm{SA}[i] - 1], \boldsymbol{x}[\mathrm{SA}[i + 1] - 1], ..., \boldsymbol{x}[\mathrm{SA}[j] - 1]$$

that are different from each other. But since $i \le i' < j' \le j$, we only need to check that

$$((\widetilde{p}; i..leftLE) \text{ is NLE}) \vee ((\widetilde{p}; rightLE..j) \text{ is NLE}).$$

From this analysis, we know that we can eliminate unnecessary letter comparisons. The revised algorithm PSY1–2 is shown in Figure 4.

Compared with PSY1–1, for the string $\boldsymbol{x}$ of Figure 1, by using PSY1–2, we can reduce letter comparisons for the repeats $(5; 8, 10) =$ *babab*, $(3; 7, 10) =$ *bab*, and $(1; 6, 10) = b$ followed by LE repeat $(7; 9, 10) =$ *bababab*, by setting *leftLE* and *rightLE* accordingly. As discussed below, there may however still be some duplicated letter comparisons using PSY1–2.

We shall find in Section 4 that for highly periodic strings, RPT1–2 provides significant speed-up over RPT1–1, mainly due to the use of the variables *leftLE* and *rightLE*; but it runs the same or slightly slower for other strings, indicating that not many cases exist in these strings such that the subranges of a range are LE; therefore the functionality of *leftLE* and *rightLE* actually is not very useful. Moreover, RPT1–2 is still not linear in the worst case, a result that is not unexpected, but that turns out to be

```
    — Preprocessing: compute SA, BWT & LCP
    — in $\Theta(n)$ time (LCP overwrites SA).
$j \leftarrow 0$; $p \leftarrow -1$; $q \leftarrow 0$; $prevNE \leftarrow 0$; push(LB; 0, 0)
while $j < n$ do
    repeat
        $j \leftarrow j+1$; $p \leftarrow q$; $q \leftarrow \mathrm{LCP}[j+1]$
        if $q > p$ and $q \geq p_{min}$ then push(LB; $j, q$)
    until $p > q$
    $leftLE \leftarrow j$; $rightLE \leftarrow j$
    repeat
        if top(LB).$lcp > 0$ then
            $(i, p) \leftarrow$ pop(LB)
            if $prevNE \geq i$ then output($p; i, j$)
            elsif $rightLE < j$ and NLE($rightLE, j$) then
                $prevNE \leftarrow rightLE$; output($p; i, j$)
            elsif $i < leftLE$ and NLE($i, leftLE$) then
                $prevNE \leftarrow i$; output($p; i, j$)
            else
                $leftLE \leftarrow i$; $rightLE \leftarrow j$
    until top(LB).$lcp \leq q$
    if top(LB).$lcp < q$ and $q \geq p_{min}$ then
        push(LB; $i, q$)
```

FIGURE 4. Algorithm PSY1–2 — compute all NE repeats of period $p \geq p_{min}$ as ranges in SA using one stack and LE range variables $leftLE, rightLE$

rather more difficult to establish than for RPT1–1. Consider a string $\boldsymbol{x}$ in which the following substrings occur:

$$\begin{array}{rcll} \boldsymbol{u}^{(k)} & = & \mu\lambda_1\cdots\lambda_k & (k \text{ times}) \\ \boldsymbol{u}^{(k-1)} & = & \mu\lambda_1\cdots\lambda_{k-1} & (2 \text{ times}) \\ & \vdots & & \\ \boldsymbol{u}^{(1)} & = & \mu\lambda_1 & (2 \text{ times}) \end{array}$$

where μ and λ_i, $1 \leq i \leq k$, are letters such that $\lambda_1 < \lambda_2 < \cdots < \lambda_k < \mu$. For example, let

$$\boldsymbol{x} = \boldsymbol{u}^{(k)}\boldsymbol{u}^{(1)}\boldsymbol{u}^{(1)}\boldsymbol{u}^{(k)}\boldsymbol{u}^{(2)}\boldsymbol{u}^{(2)}\cdots\boldsymbol{u}^{(k)}\boldsymbol{u}^{(k-1)}\boldsymbol{u}^{(k-1)}\boldsymbol{u}^{(k)}$$

of length $n = 2(k^2+k-1)$, and observe that for every λ_i, $1 \leq i \leq k$, there exist exactly $k-i+1$ distinct substrings

$$\lambda_i\lambda_{i+1}\cdots\lambda_k < \lambda_i\lambda_{i+1}\cdots\lambda_{k-1} < \cdots < \lambda_i,$$

all of which are NRE and LE repeats, with the lexicographically least occurring k times, the others twice. It follows that during the execution of PSY1–2, the function NLE will need to perform letter comparisons on the substring $\lambda_i\lambda_{i+1}\cdots\lambda_k$ of length $k-i+1$ a total of $k-i+1$ separate times. Then at least

$$\sum_{i=1}^{k}(k-i+1)^2 = k(k+1)(2k+1)/6 \in \Theta(k^3)$$

letter comparisons are required. Since $n \in \Theta(k^2)$, we see that the number of letter comparisons is $O(n\sqrt{n})$, as far as we know the worst case for PSY1–2.

PSY1–3

To guarantee worst-case linear time, we use another stack PREVRANGES, thus creating a third variant PSY1–3 (see Figures 5 and 6).

```
function NLE($i, j$, PREVRANGES)
  — Range is NLE if any subrange is NLE.
$\lambda \leftarrow$ BWT[$i$]; $I \leftarrow i$
while top(PREVRANGES).$j' > i$ do
    $(i', j', \lambda') \leftarrow$ pop(PREVRANGES)
    if $\lambda \neq \$$ then
        if $\lambda = \lambda'$ then $I \leftarrow j-1$
        else $\lambda \leftarrow \$$
if $\lambda = \$$ then return TRUE
else
    $\lambda \leftarrow$ CHECK($I+1, j, \lambda$)
    if $\lambda = \$$ then return TRUE
    else push(PREVRANGES; $i, j, \lambda$); return FALSE

function CHECK($min, max, \lambda$)
$j' \leftarrow min$
while $j' \leq max$ and BWT[$j'$] $= \lambda$ do $j' \leftarrow j'+1$
if $j' > max$ then
    return $\lambda$
else
    return $\$$
```

FIGURE 5. Determine whether the repeat SA[$i..j$] is NLE (two stacks)

If the current repeat processed by function NLE (Figure 5) is found to be left-extendible, its range limits i, j are pushed onto PREVRANGES together with the letter λ that precedes each suffix in the range $i..j$. Since each range must be either disjoint from, or a proper subrange of, subsequent ranges identified during the scan (reflecting the subtree structure of the suffix tree of $\boldsymbol{x}$), these ranges allow us to efficiently determine the left-extendibility of subsequent ranges without duplicating letter comparisons already made, based on the following simple observations:

∗ every subrange of an LE range must also be LE (that is, a single NLE subrange ensures that the enclosing range is also NLE);
∗ moreover, the letter $\lambda \neq \$$ that establishes left-extendibility must be identical over all the subranges found in PREVRANGES.

```
    — Preprocessing: compute SA, BWT & LCP
    — in $\Theta(n)$ time (LCP overwrites SA).
$j \leftarrow 0$; $p \leftarrow -1$; $q \leftarrow 0$; $prevNE \leftarrow 0$
push(LB; 0, 0); push(PREVRANGES; 0, 0, $)
while $j < n$ do
    repeat
        $j \leftarrow j+1$; $p \leftarrow q$; $q \leftarrow$ LCP[$j+1$]
        if $q > p$ and $q \geq p_{min}$ then push(LB; $j, q$)
    until $p > q$
    repeat
        if top(LB).$lcp > 0$ then
            $(i, p) \leftarrow$ pop(LB)
            if $prevNE \geq i$ then output($p; i, j$)
            elsif NLE($i, j$, PREVRANGES) then
                setempty(PREVRANGES)
                push(PREVRANGES; 0, 0, $)
                $prevNE \leftarrow i$; output($p; i, j$)
    until top(LB).$lcp \leq q$
    if top(LB).$lcp < q$ and $q \geq p_{min}$ then
        push(LB; $i, q$)
```

FIGURE 6. Algorithm PSY1–3 — compute all NE repeats of period $p \geq p_{min}$ as ranges in SA using two stacks

We mentioned above that by using PSY1–2, we can only reduce some letter comparisons for the repeats $(5; 8, 10)$ = *babab*, $(3; 7, 10)$ = *bab*, and $(1; 6, 10)$ = *b* followed by LE repeat $(7; 9, 10)$ = *bababab*, but it is turns out that by using PSY1–3, we will eliminate all unnecessary letter comparisons. For example, if NRE repeats $(7; 9, 10)$, $(5; 8, 10)$, and $(3; 7, 10)$ are already identified by function NLE as LE, when an NRE repeat $(1; 6, 10)$ is checked by function NLE, since ranges 9..10, 8..9 and 7..8 were already checked (only once) and $\lambda = a$ was pushed onto stack PREVRANGES (since BWT[7] = BWT[8] = a), therefore function NLE will yield this λ value and only check the range 6..7; that is, whether or not BWT[6] = BWT[7].

Now we have the conclusion that since every LE range is pushed onto PREVRANGES with a λ value that is not a sentinel $, therefore subsequent NRE repeats, which include the previous ranges as subranges, will only check the ranges that are unmarked with λ values; thus in the worst case, the number of letter comparisons is $O(n)$.

PSY1–4

To guarantee worst-case linear time, and still use one stack, we create the fourth variant PSY1–4 (see Figure 7). As shown in Figure 7, PSY1–4 performs a single left-to-right scan of LCP, inspecting each position j from 1 to n. During the scan, whenever a position lb (initially $lb = j$) is found for which the LCP value increases, an entry is pushed onto a stack LB. As before, LB specifies the Left Boundary lb and period p of a repeat that must be NRE, but that may or may not be NLE: lb marks the leftmost occurrence in SA of a repeating substring of length $p = \text{LCP}[lb+1] > \text{LCP}[lb]$, thus the left boundary of a repeat. In fact, a triple (p, lb, bwt) is pushed onto the stack, where bwt is a letter that determines the left-extendibility of the repeat: initially bwt equals the sentinel letter \$ if $\text{BWT}[lb] \neq \text{BWT}[lb{+}1]$, and otherwise equals $\text{BWT}[lb]$. This is the calculation performed repeatedly by the function `LEletter`. Thus $bwt = \$$ if the repeat is NLE (and so eventually should be output), but assumes a regular letter value if the repeat (so far at least) is LE.

```
  — Preprocessing: compute SA, BWT & LCP
  — in $\Theta(n)$ time and $6n$ bytes of space.
$lcp \leftarrow$ LCP[1]; $lb \leftarrow 1$; $bwt1 \leftarrow$ BWT[1]
push(LB; $lcp, lb, bwt1$)
for $j \leftarrow 1$ to $n$ do
    $lb \leftarrow j$; $lcp \leftarrow$ LCP[$j+1$]
      — Compute LEletter of BWT[$j$] and BWT[$j+1$].
    $bwt2 \leftarrow$ BWT[$j+1$]; $bwt \leftarrow$ LEletter($bwt1, bwt2$)
    $bwt1 \leftarrow bwt2$
    while top(LB).$lcp > lcp$ do
        pop(LB; $p, i, prevbwt$)
        if $prevbwt = \$$ and $p \geq p_{min}$ then
            output($p; i, j$)
        $lb \leftarrow i$
        top(LB).$bwt \leftarrow$ LEletter($prevbwt$, top(LB).$bwt$)
        $bwt \leftarrow$ LEletter($prevbwt, bwt$)
    if top(LB).$lcp = lcp$ then
        top(LB).$bwt \leftarrow$ LEletter(top(LB).$bwt, bwt$)
    else
        push(LB; $lcp, lb, bwt$)

function LEletter($\ell_1, \ell_2$)
if $\ell_1 = \$$ or $\ell_1 \neq \ell_2$ then return \$
else return $\ell_1$
```

FIGURE 7. Algorithm PSY1–4: compute all NE repeats of period $p \geq p_{min}$ as ranges in SA

Since the pushes to LB occur in increasing order of position lb, the pops occur in decreasing order of lb: the most recently pushed triple is popped when a position j is reached for which LCP$[j+1]$ < $\texttt{top}(LB).lcp$. Then j is the right boundary for the popped triple $(p, i, prevbwt)$ and a repeat $(p; i, j)$ is identified. Observe that this repeat is NRE: if the same letter followed each occurrence of the repeating substring of length p, then p could not be maximum, contradicting the definition of LCP.

It remains to determine whether or not the popped triple is NLE. For this the popped value $prevbwt$ needs to be inspected to determine whether it is \$ — that is, whether the repeat is NLE, whether it should be output. To ensure that $\texttt{top}(\text{LB}).bwt$ is maintained correctly, we use a simple property of ranges of repeats: two ranges are either disjoint (empty common prefix) or else one range contains the other (common prefix over the longer range). It follows that if $\texttt{top}(\text{LB}).bwt = \$$ for a contained range, then for every range that encloses it, we must also have $\texttt{top}(\text{LB}).bwt = \$$. Moreover, if for some letter $\lambda \in A$, a contained range is LE with $bwt = \lambda$, then the enclosing range will be LE only if every other contained range also has $bwt = \lambda$. In PSY1–4 the correct bwt value for the enclosing range is maintained by invoking `LEletter` to update $\texttt{top}(\text{LB}).bwt$ whenever LCP$[j+1] \le \texttt{top}(\text{LB}).lcp$. For LCP$[j+1] < \texttt{top}(\text{LB}).lcp$, `LEletter` is used again to update the current bwt based on the $prevbwt$ just popped.

In view of this discussion, we claim the correctness of PSY1–4. Execution time is $\Theta(n)$, since the number of executions of the **while** loop is at most the number of triples pushed onto LB, thus $O(n)$.

Space required directly for the execution of all PSY1 variants is $5n$ bytes plus maximum stack storage: 8-byte entries in LB and 9-byte entries in PREVRANGES. The largest number of entries in both of these stacks will be the maximum depth of the suffix tree — thus $O(n)$ in the worst case — but expected depth on an alphabet of size $\alpha > 1$ is $2\log_\alpha n$ [10]. Thus even for $\alpha = 2$, expected space for LB is $18\log_\alpha n$ bytes — if $n = 2^{20}$, 360 bytes. On strings arising in practice, LB requires negligible space (Section 5).

3. Algorithms for Supernonextendible Repeats

PSY2–1

The SNE ("supermaximal") repeats algorithm described in [1] does not deal explicitly with the problem of determining whether or not a complete super NRE (SNRE) repeat is also SNLE. This determination requires that the left extensions (BWT values) of the k positions in the repeat be pairwise distinct. The straightforward approach to this problem requires at most $\binom{k}{2}$ letter comparisons, where k can possibly be order n. However, we make two observations:

* The cardinality k of an SNE repeat cannot exceed the alphabet size α. Thus a single test suffices to eliminate candidate SNRE repeats of cardinality greater than α, and the straightforward algorithm can then

compute SNE repeats in time $O(n{+}z\alpha^2)$, where $z \in O(n)$ is the number of SNRE repeats in $\boldsymbol{x}$.

$*$ Use of a bit map $B = B[1..\alpha]$ can reduce to $\Theta(\alpha)$ the time required to determine whether or not each SNRE repeat is also SNLE, thus reducing worst-case time to $\Theta(\alpha n)$.

Figure 8 gives details of the algorithm PSY2–1 suggested by these remarks. Since PSY2–1 requires no stacks, storage is reduced to $5n$ bytes plus α bits (again with up to $6n$ bytes for LCP construction).

```
  — Preprocessing: compute SA, BWT & LCP
  — in $\Theta(n)$ time (LCP overwrites SA).
$j \leftarrow 0;\ p \leftarrow -1;\ q \leftarrow 0$
while $j < n$ do
    $high \leftarrow 0$
    repeat
        $j \leftarrow j{+}1;\ p \leftarrow q;\ q \leftarrow \mathrm{LCP}[j{+}1]$
        if $q > p$ then $high \leftarrow q;\ start \leftarrow j$
    until $p > q$
    if $high > 0$ and SNLE$(start, j)$ then
        output$(p; start, j)$

function SNLE$(start, end)$
$k \leftarrow end{-}start{+}1$
if $k > \alpha$ then return FALSE
else
    $B[1..\alpha] \leftarrow \mathtt{FALSE}^{\alpha}$
    for $h \leftarrow start$ to $end$ do
        $\lambda \leftarrow \mathrm{BWT}[h]$
        if $B[\lambda]$ then return FALSE
        else $B[\lambda] \leftarrow$ TRUE
    return TRUE
```

FIGURE 8. Algorithm PSY2–1 — compute all SNE repeats as ranges in SA using a bit array B

PSY2–2

A theoretical and perhaps also practical disadvantage of PSY2–1 is its need to perform $\Theta(\alpha)$-time processing in function SNLE in order to clear the bit array B, a task that may be repeated $O(n)$ times. We now describe a more sophisticated approach that reduces worst-case complexity to $\Theta(n{+}\alpha)$ at the cost of a slight increase in actual processing time.

Instead of $\mathrm{BWT}_{\boldsymbol{x}}$, we compute an array $\mathrm{LAST} = \mathrm{LAST}_{\boldsymbol{x}}[1..n]$ in which for every $j \in 1..n$, $\mathrm{LAST}[j]$ measures the offset between the BWT letter corresponding to the current position j in SA and the position $jprev$ of the rightmost previous occurrence in SA of the same BWT letter — if $jprev$ does

```
function SNLE($start, end,$ LAST)
$k \leftarrow end - start + 1$
if $k > \alpha$ then return FALSE
else
    for $h \leftarrow start+1$ to $end$ do
        if $h-$LAST$[h] > start$ then return FALSE
    return TRUE
```

FIGURE 9. Algorithm PSY2–2 — the simplified SNLE function using LAST

```
 — Initialize an array storing rightmost positions of each letter.
for $\ell \leftarrow 1$ to $\alpha$ do
    $lastpos[\ell] \leftarrow 0$
 — Compute LAST in a single left-to-right scan of SA.
$\alpha' \leftarrow \alpha - 1$
for $j \leftarrow 1$ to $n$ do
    $i \leftarrow$ SA$[j]-1$
    if $i \leftarrow 0$ then
        LAST$[j] \leftarrow \alpha'$
    else
        $letter \leftarrow \boldsymbol{x}[i]$; $jprev \leftarrow lastpos[letter]$
        if $jprev = 0$ or $j - jprev \geq \alpha$ then
            LAST$[j] \leftarrow \alpha'$
        else
            LAST$[j] \leftarrow j - jprev - 1$
        $lastpos[letter] \leftarrow j$
```

FIGURE 10. Preprocessing for Algorithm PSY2–2 — computing LAST

not exist or if $j - jprev \geq \alpha$, then LAST$[j] \leftarrow \alpha - 1$. However, if *jprev* exists and satisfies $j - jprev < \alpha$, we set

$$\text{LAST}[j] \leftarrow j - jprev - 1,$$

so that LAST$[j]$ takes values in the range $0..\alpha-2$. Then when function SNLE processes a possibly supernonextendible repeat consisting of $end - start + 1$ occurrences of a repeating substring of $\boldsymbol{x}$, for every position $h \in start + 1..end$, the value of BWT$[h]$ will be unique within the range if and only if $h-$LAST$[h] > start$. Given LAST, the function SNLE simplifies as shown in Figure 9.

In general it is possible that the offsets stored in LAST could be integers of size $O(n)$. But offsets of magnitude greater than $\alpha-1$ need not be stored, since as remarked above the interval *start..end* consists of at most α positions. Thus LAST requires the same amount of storage as BWT, which stores letters that are also restricted to be at most $\alpha-1$ in magnitude. The method can be implemented for any finite α, but with the usual convention that each letter in the alphabet is confined to a single byte ($\alpha \leq 256$), the array LAST becomes an array of bytes, just like BWT. The calculation of LAST is shown in Figure 10. (In fact, in order to take advantage of the CPU cache, our implementation of this algorithm actually computes BWT first, then makes a pass over BWT to convert it into LAST — an approach that turns out to be 2–3 times faster than a straightforward implementation of the preprocessing algorithm.)

4. Experimental Results

Experiments were conducted on a diverse selection of files (Table 1) chosen from the collection at `http://www.cas.mcmaster.ca/~bill/strings/`. These files are of five main types:

* highly periodic strings – strings that do not occur often in practice, containing many repetitions (Fibonacci strings, binary strings constructed in [6]);
* strings with very few runs (random strings on small and fairly large alphabets).
* DNA strings on alphabet $\{a, c, g, t\}$;
* protein sequences on an alphabet of 20 letters;
* strings on large alphabets (English-language, ASCII characters).

File Type	Name	No. Bytes	α	Description
highly periodic	fibo35	9,227,465	2	Fibonacci
	fibo36	14,930,352	2	Fibonacci
	fss9	2,851,443	2	run-rich [6]
	fss10	12,078,908	2	run-rich [6]
random	rand2	8,388,608	2	$\alpha = 2$
	rand21	8,388,608	21	$\alpha = 21$
DNA	ecoli	4,638,690	4	*escherichia coli* genome
	chr22	34,553,758	4	human chromosome 22
	chr19	63,811,651	4	human chromosome 19
Genbank protein database	prot-a	16,777,216	20	sample
	prot-b	33,554,432	20	doubled sample
English	bible	4,047,392	63	King James bible
	howto	39,422,105	197	Linux howto files
	mozilla	51,220,480	256	Mozilla source code

TABLE 1. Files used for testing.

Tests were conducted using a 2.6GHz Opteron 885 processor with 2GB main memory available, under Red Hat Linux 4.1.2–14. The compiler was `gcc` with the -O3 option. The run times used were the minima over four runs, not including disk I/O. We choose minima instead of average running times because increased running time is usually caused by some form of interference from the operating system due to competing activities in the computer: the minimum time will therefore be closest to actual.

Run times for the tests are shown in Tables 2, 3 and 4.

File	SA	LCP	BWT	LAST
fibo35	0.898	0.142	0.025	0.031
fibo36	0.886	0.601	0.027	0.033
fss9	0.826	0.561	0.026	0.031
fss10	0.958	0.576	0.025	0.032
periodic AVG	0.892	0.470	0.026	0.032
rand2	0.947	0.144	0.026	0.031
rand21	1.135	0.112	0.025	0.031
random AVG	1.041	0.128	0.025	0.031
ecoli	1.413	0.116	0.025	0.031
chr22	1.635	0.146	0.035	0.040
chr19	1.873	0.160	0.044	0.053
DNA AVG	1.754	0.141	0.035	0.041
prot-a	1.778	0.142	0.027	0.032
prot-b	1.971	0.159	0.034	0.039
protein AVG	1.874	0.151	0.030	0.036
bible	1.417	0.111	0.024	0.030
howto	1.912	0.178	0.035	0.039
mozilla	1.815	0.135	0.032	0.036
English AVG	1.417	0.141	0.030	0.035
AVERAGE	1.390	0.235	0.029	0.035

TABLE 2. Microseconds per letter used by each run for preprocessing: SA, LCP, BWT, and LAST arrays

We give in Table 2 the preprocessing times for the various data structures required by the PSY1, PSY2, and [17] algorithms; specifically, the SA, LCP, BWT, and LAST arrays. For SA construction the KS algorithm was used [9] — the fastest such algorithm is perhaps MP2 [14] that, based on experiments documented in [14, 18], would perform 5–10 times faster on average, using about $5.2n$ bytes of storage. For LCP construction the algorithm of [20] was used.

Test results for the algorithms are shown in Table 3 (PSY1 together with the algorithm of [17]) and Table 4 (PSY2).

Averages within file type are not weighted by file size, and the final AVERAGE is a simple average of the "microseconds per letter" ratios for each of the 14 test files. Note that for each program tested, the number of microseconds per letter is generally stable within each file type and not highly variable overall. Tests of PSY1 used $p_{min} = 1$; as expected, for larger p_{min} the run time decreased, usually to about half the starting value.

File	PSY1–1	PSY1–2	PSY1–3	PSY1–4	[17]
fibo35	0.054	0.052	0.019	**0.012**	0.061
fibo36	0.056	0.052	0.020	**0.012**	0.056
fss9	0.049	0.049	0.021	**0.014**	0.057
fss10	0.055	0.055	0.021	**0.013**	0.059
periodic AVG	0.054	0.052	0.020	**0.013**	0.058
rand2	**0.014**	0.016	0.017	0.017	0.052
rand21	**0.008**	0.009	0.010	0.012	0.013
random AVG	**0.011**	0.013	0.013	0.015	0.032
ecoli	**0.012**	0.014	0.015	0.015	0.031
chr22	**0.014**	0.016	0.016	0.016	0.047
chr19	**0.014**	0.020	0.022	0.016	0.051
DNA AVG	**0.013**	0.017	0.019	0.016	0.043
prot-a	**0.011**	0.013	0.013	0.013	0.028
prot-b	**0.012**	0.014	0.014	0.013	0.037
protein AVG	**0.012**	0.013	0.014	0.013	0.033
bible	0.016	0.017	0.017	**0.015**	0.049
howto	**0.015**	0.017	0.018	0.016	0.060
mozilla	**0.011**	0.013	0.013	0.013	0.032
English AVG	**0.014**	0.016	0.016	0.015	0.047
AVERAGE	0.024	0.025	0.017	**0.014**	0.045

TABLE 3. Microseconds per letter used by each run for PSY1 and the algorithm of [17]

In each section of both Table 3 and 4, we underline in bold the quantity that achieves the best result for the current test case.

To assess the effect of stack use on the space requirements of the PSY1 family, a record was kept of maximum stack usage for each of the files tested, as shown in Table 5. The files `prot-a` and `prot-b` consumed by far the most stack space, totalling in each case a little less than 125K bytes. (The maximum size of the LB stack was as expected identical for all four PSY1 algorithms.)

5. Discussion

We make the following observations:

* We see from Tables 3 and 4 that the six new algorithms tested are very fast, especially on strings that arise in practice: even if SA were to execute 10 times faster, as it might if MP2 were used, still each algorithm would require 5% or less of the total SA/LCP time shown in Table 2.
* From the 6th and last lines in Table 3, we observe that the PSY1–4 executes very quickly on highly periodic strings and is the fastest one of the four variants over the complete range of strings tested.

File	PSY2–1	PSY2–2
fibo35	0.005	**0.004**
fibo36	0.005	**0.004**
fss9	0.006	**0.004**
fss10	0.006	**0.005**
periodic AVG	0.006	**0.004**
rand2	0.010	**0.008**
rand21	0.010	**0.008**
random AVG	0.010	**0.008**
ecoli	0.010	**0.008**
chr22	0.010	**0.008**
chr19	0.010	**0.008**
DNA AVG	0.010	**0.008**
prot-a	0.010	**0.008**
prot-b	0.010	**0.008**
protein AVG	0.010	**0.008**
bible	0.010	**0.008**
howto	0.009	**0.008**
mozilla	0.008	**0.007**
English AVG	0.009	**0.008**
AVERAGE	0.009	**0.007**

TABLE 4. Microseconds per letter used by each run for PSY2 algorithms

* Contrary to the averages given in the last line of Table 3, PSY1–1 and PSY1–2 are actually slightly faster than PSY1–3 and PSY1–4 on strings that arise in practice (that is, strings that are not highly periodic): the average microseconds per letter for PSY1–1 on such files is 0.013, the best of all.
* As shown in Table 5, we have computed maximum stack size for each of the test files: only for `prot-a` and `prot-b` did the maximum size of the LB stack exceed three digits — for `prot-a` (the worst case) the total maximum storage for LB and PREVRANGES was 0.1% of the $5n$ bytes required for LCP and BWT storage.
* The algorithm of [17] was originally designed to provide more comprehensive output than that of our PSY1 algorithms, and so is not directly comparable. However, the authors have modified their code to reduce processing requirements. Using this version for Table 3, the modified version appears to execute around 2–3 times slower than PSY1 on real-world files.
* We conducted no experiments on the algorithm of [7] because it needs to compute SA/LCP twice and will therefore be very slow.
* Even though, in addition to its asymptotic advantage, PSY2–2 runs around 30% faster than PSY2–1, nevertheless it does not overcome the disadvantage of the additional preprocessing time required for LAST

compared to BWT (see both Tables 2 and 4): PSY2–1 together with BWT runs consistently slightly faster than PSY2–2 with LAST.

File	LB	PREVRANGES
fibo35	33	30
fibo36	34	32
fss9	33	36
fss10	37	40
random2	24	8
random21	8	5
ecoli	24	14
chr22	64	35
chr19	249	50
prot-a	6701	7448
prot-b	6701	7448
bible	23	112
howto	91	292
mozilla	2772	2750

Table 5. Maximum number of stack entries required by PSY1

In view of particularly the first of these observations, it seems clear that future progress in computing repeats will depend upon more efficient preprocessing – either much improved SA/LCP construction or the use of entirely new data structures.

The output of PSY1 and PSY2 can be used in various ways and for various purposes. For offline data compression the output can be used for phrase selection [2, 13, 23]. It could also be used in the algorithm of [3] for duplicate text/document detection. If the user requires positions in $\boldsymbol{x}$ to be output, this can trivially be achieved, since SA is available, by postprocessing that replaces $i..j$ by $\mathrm{SA}[i], \mathrm{SA}[i{+}1], \ldots, \mathrm{SA}[j]$. In applications to protein sequences, such as the detection of low-complexity regions, the use of PSY1 will provide significant algorithmic speed-up over currently-proposed methods [21] that are effective but slow. In the context of genome analysis the postprocessing of interest may be to compute NE pairs as in [8, 4, 1]. Assuming an integer alphabet $1..\alpha$, this can be accomplished as follows for each range $i..j$. Introduce a new array $\mathrm{BWT}' = \mathrm{BWT}'[1..n]$, where for $\mathrm{SA}[h] < n$, $\mathrm{BWT}'[h] = \boldsymbol{x}[\mathrm{SA}[h]{+}1]$, otherwise $\mathrm{BWT}'[h] = \$$.

(1) Perform a radix sort on the pairs

$$(\mathrm{BWT}[i], \mathrm{BWT}'[i]),\ (\mathrm{BWT}[i{+}1], \mathrm{BWT}'[i{+}1]),\ \ldots,\ (\mathrm{BWT}[j], \mathrm{BWT}'[j])$$

into bins that are accessed from an array $\mathrm{B} = \mathrm{B}[1..\alpha, 1..\alpha]$. As a byproduct of the sort, positions in a Boolean array $\mathrm{E} = \mathrm{E}[1..\alpha]$ are set: $\mathrm{E}[b] =$ `TRUE` if and only if row b of B is empty.

(2) For every nonempty row b_1 of B, and for every $b_2 \in 1..\alpha$, perform the following simple processing:

```
for $h_1 \leftarrow b_1+1$ to $\alpha$ do
  if not $\mathrm{E}[h_1]$ then
    for $h_2 \leftarrow (1$ to $b_2-1)$ and $(b_2+1$ to $\alpha)$ do
      output all pairs $\mathrm{B}(b_1, b_2)$ with $\mathrm{B}(h_1, h_2)$
```

This approach requires checking at most $\alpha^2(\alpha-1)^2/2$ positions in B for each range processed; in the DNA case with $\alpha = 4$, this amounts to at most 72 (that is, $\alpha^3+2\alpha$) positions, but will for most ranges be much less. Otherwise the time required is proportional to the number of pairs output. Due to CPU cache effects, we believe this will be an efficient algorithm for computing NE pairs: it depends only on $i, j, \mathrm{BWT}, \mathrm{BWT}'$.

References

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

[2] Alberto Apostolico and Stefano Lonardi, **Off-line compression by greedy textual substitution**, *Proc. IEEE 88–11* (2000) 1733–1744.

[3] Yaniv Bernstein and Justin Zobel, **Accurate discovery of co-derivative documents via duplicate text detection**, *Information Systems 31* (2006) 595–609.

[4] Gerth S. Brodal, Rune B. Lyngso, Christian N. S. Pederesen, and Jens Stoye, **Finding maximal pairs with bounded gap**, *J. Discrete Algs. 1* (2000) 77–103.

[5] Michael Burrows and David J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Technical Report 124, Digital Equipment Corporation (1994).

[6] Frantisek Franek, R. J. Simpson, and W. F. Smyth, **The maximum number of runs in a string**, *Proc. 14th Australasian Workshop on Combinatorial Algs.*, Mirka Miller & Kunsoo Park (eds.) (2003) 36–45.

[7] Frantisek Franek, W. F. Smyth, and Yudong Tang, **Computing all repeats using suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 579–591.

[8] Dan Gusfield, *Algorithms on Strings, Trees & Sequences*, Cambridge University Press (1997) 534 pp.

[9] Juha Kärkkäinen and Peter Sanders, **Simple linear work suffix array construction**, *Proc. 30th Internat. Colloq. Automata, Languages & Programming* (2003) 943–955.

[10] S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn, **New approaches for computer analysis of nucleic acid sequences**, *Proc. Natl. Acad. Sci. USA 80* (1983) 5660–5664.

[11] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181–192.

[12] Pang Ko and Srinivas Aluru, **Space efficient linear time construction of suffix arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez & M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003) 200–210.

[13] Jesper Larsson and Alistair Moffat, **Off-line dictionary-based compression**, *Proc. IEEE 88–11* (2000) 1722–1732.

[14] Michael Maniscalco and Simon J. Puglisi, **Faster lightweight suffix array construction**, *Proc. 17th Australasian Workshop on Combinatorial Algs.*, J. Ryan & Dafik (eds.) (2006) 16–29.

[15] Giovanni Manzini, **Two space-saving tricks for linear time LCP computation**, *Proc. 9th Scandinavian Workshop on Alg. Theory*, LNCS 3111, T. Hagerup & J. Katajainen (eds.), Springer-Verlag (2004) 372–383.

[16] Giovanni Manzini and Paolo Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica 40* (2004) 33–50.

[17] Kaziyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda, **Efficient computation of substring equivalence classes with suffix arrays**, *Proc. 18th Annual Symp. Combinatorial Pattern Matching* (2007) 340–351.

[18] Simon J. Puglisi, W. F. Smyth, and Andrew Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys 39–2* (2007) 1–31.

[19] S. J. Puglisi, W. F. Smyth, and M. Yusufu, **Fast optimal algorithms for computing all the repeats in a string**, *Proc. the Prague Stringology Conference* (2008) 161–169.

[20] Simon J. Puglisi and Andrew Turpin, **Space-time tradeoffs for longest-common-prefix array computation**, *Proc. 19th Internat. Symp. Algorithms & Computation*, LNCS 5369, S-H. Hong, H. Nagamochi & T. Fukunaga (eds.), Springer-Verlag (2008) 124–135.

[21] Sung W. Shin and Sam M. Kim, **A new algorithm for detecting low-complexity regions in protein sequences**, *Bioinformatics 21-2* (2005) 160–170.

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

[23] Andrew Turpin and W. F. Smyth, **An approach to phrase selection for offline data compression**, *Proc. 25th Australasian Computer Science Conference*, Michael Oudshoorn (ed.) (2002) 267–273.

Simon J. Puglisi
School of Computer Science & Information Technology,
RMIT University
GPO Box 2476V
Melbourne, Victoria 3001, Australia
e-mail: `simon.puglisi@rmit.edu.au`

W. F. Smyth
Department of Computing
Digital Ecosystems & Business Intelligence Institute
Curtin University of Technology
GPO Box U1987, Perth WA 6845, Australia
e-mail: `smyth@cs.curtin.edu.au`

Algorithms Research Group
Department of Computing & Software
McMaster University
Hamilton, Ontario, Canada L8S 4K1
e-mail: `smyth@mcmaster.ca`

Munina Yusufu
Algorithms Research Group
Department of Computing & Software
McMaster University
Hamilton, Ontario, Canada L8S 4K1
e-mail: `yusufum@mcmaster.ca`
Digital Ecosystems & Business Intelligence Institute
Curtin University of Technology
GPO Box U1987, Perth WA 6845, Australia