Fast Optimal Algorithms for Computing All the Repeats in a String[⋆]

Simon J. Puglisi[1], William F. Smyth[2,3], and Munina Yusufu[2]

[1] School of Computer Science & Information Technology,
RMIT University, GPO Box 2476V, Melbourne, Victoria 3001, Australia
`sjp@cs.rmit.edu.au`

[2] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
`{smyth,yusufum}@mcmaster.ca`
`http://www.cas.mcmaster.ca/cas/research/algorithms.htm`

[3] Digital Ecosystems & Business Intelligence Institute
Curtin University, GPO Box U1987, Perth WA 6845, Australia
`W.Smyth@curtin.edu.au`

Abstract. Given a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet of size α, and a threshold $p_{min} \geq 1$, we first describe a new algorithm PSY1 that, based on suffix array construction, computes all the complete ***nonextendible*** repeats in $\boldsymbol{x}$ of length $p \geq p_{min}$. PSY1 executes in $\Theta(n)$ time independent of alphabet size and is an order of magnitude faster than the two other algorithms previously proposed for this problem. Second, we describe a new fast algorithm PSY2 for computing all complete ***supernonextendible*** repeats in $\boldsymbol{x}$ that also executes in $\Theta(n)$ time independent of alphabet size, thus asymptotically faster than methods previously proposed. Both algorithms require $9n$ bytes of storage, including preprocessing (with a minor *caveat* for PSY1). We conclude with a brief discussion of applications to bioinformatics and data compression.

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 repeating substrings $\boldsymbol{u}$ of $\boldsymbol{x}$; it can be specified by the length $p \geq 1$ of $\boldsymbol{u}$ (what we call its ***period***) and the locations at which $\boldsymbol{u}$ occurs. Thus in $\boldsymbol{x} = abaababa$, the tuple $(3;\ 1, 4, 6)$ describes the repeat of $\boldsymbol{u} = aba$ $(p = 3)$ at positions 1, 4, 6.

Following [20] 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 $\boldsymbol{x}$.

[⋆] The work of the first author was supported by the Australian Research Council, that of the second and third authors by the Natural Sciences & Engineering Research Council of Canada.

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 applications usually $\alpha = 4$ (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*** — essentially the complete NE repeats — in $\boldsymbol{x}$.

In this paper we first describe an 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 executes in $\Theta(n)$ time independent of alphabet size and requires $5n$ bytes of storage, plus a stack, but its preprocessing includes suffix array construction that raises the storage requirement to $9n$ bytes. PSY1 is an order of magnitude faster than the complete repeats algorithms described in [7,17].

We also describe a new fast algorithm PSY2 that computes all the complete SNE repeats in $\boldsymbol{x}$ 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 remains $\Theta(n)$.

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

2 Description of the Algorithms

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 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]: 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. (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. Examples of these standard data structures follow:

$$\begin{array}{rcccccccccc}
 & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\boldsymbol{x} & = & a & b & a & a & b & a & b & a & \$ \\
\mathrm{SA}_{\boldsymbol{x}} & = & 8 & 3 & 6 & 1 & 4 & 7 & 2 & 5 & \\
\mathrm{LCP}_{\boldsymbol{x}} & = & \text{-}1 & 1 & 1 & 3 & 3 & 0 & 2 & 2 & \text{-}1 \\
\mathrm{BWT}_{\boldsymbol{x}} & = & b & b & b & \$ & a & a & a & a & \$
\end{array}$$

Here as in the Introduction the repeating substring $\boldsymbol{u} = aba$ of length 3 occurs in positions $6, 1, 4$ of $\boldsymbol{x}$; our algorithms report this fact as a complete repeat (it is both NE and SNE) in the form $(3; 3, 5)$ with period $p = 3$, where $3, 5$ is a range identifying $\mathrm{SA}[3] = 6, \mathrm{SA}[4] = 1, \mathrm{SA}[5] = 4$. Note that $p = \mathrm{LCP}[4] = \mathrm{LCP}[5]$.

All of the algorithms described in this paper make direct use of LCP and BWT (or equivalent), but not of SA, and therefore require only $5n$ bytes of storage (plus relatively small stack space in the case of PSY1). However, the calculation [15] of LCP requires SA, a further $4n$ bytes, and so, as noted above, the total space requirement is $9n$. The output of both algorithms is a range $i..j$ of positions in SA that specifies a complete repeat (NE for PSY1, SNE for PSY2).

PSY1

Given a threshold $p_{min} \geq 1$, PSY1 outputs all the complete NE repeats in a given string $\boldsymbol{x}$, each one a triple $(p; i, j)$ specifying a period $p \geq p_{min}$ and a range $i..j$ in SA such that the suffixes $\mathrm{SA}[i], \mathrm{SA}[i{+}1], \ldots, \mathrm{SA}[j]$ form a maximal set with the same longest common prefix of length

$$p\ (lcp) = \mathrm{LCP}[i{+}1] = \mathrm{LCP}[i{+}2] = \cdots = \mathrm{LCP}[j].$$

As shown in Figure 1, PSY1 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. 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 $p = \mathrm{LCP}[lb{+}1] > \mathrm{LCP}[lb]$, 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 $\mathrm{BWT}[lb] \neq \mathrm{BWT}[lb{+}1]$, and otherwise equals $\mathrm{BWT}[lb]$. This is the calculation performed repeatedly by the function `LEletter`. Thus $bwt = \$$ if the repeat is NLE (and so eventually should be printed), but assumes a regular letter value if the repeat (so far at least) is LE.

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 $\mathrm{LCP}[j{+}1] < \texttt{top}(LB).lcp$. 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.

```
  — Preprocessing: compute SA, BWT & LCP
  — in $\Theta(n)$ time and $9n$ 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 1. Algorithm PSY1: compute all NE repeats of period $p \geq p_{min}$ as ranges in SA

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 top(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 top(LB).$bwt = \$$ for a contained range, then for every range that encloses it, we must also have top(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 the correct *bwt* value for the enclosing range is maintained by invoking LEletter to update top(LB).*bwt* whenever LCP[j+1] $\leq$ top(LB).*lcp*. For LCP[j+1] $<$ top(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. 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 is $5n$ bytes plus maximum stack size at 9 bytes per entry (four bytes each for *lb* and *lcp*, plus a byte for *bwt*). The largest number of entries in LB is exactly the maximum depth of the suffix tree — in fact n for $\boldsymbol{x} = a^n$ — 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 4).

PSY2

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

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

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

Figure 2. Algorithm PSY2 with a simplified SNLE function using LAST

the k positions in the repeat be pairwise distinct. The approach apparently proposed by the authors requires at most $\binom{k}{2}$ letter comparisons, where k can be order n, thus leading to $O(n^2)$ time in the worst case. A perhaps more efficient approach would be to use a bit map B$[1..\alpha]$ to determine if any letter in the alphabet has occurred more than once as a left extension over the range of the repeat. However, this would require initializing the α positions in B for each of $O(n)$ candidate repeats, and since possibly $\alpha \in O(n)$, the time required could again be $O(n^2)$. Our proposed algorithm PSY2 (Figure 2) incorporates two improvements, one to decrease execution time in practice, the other to reduce asymptotic complexity to $O(n+\alpha)$.

We observe first that the cardinality k of an SNE repeat cannot exceed the alphabet size α. Thus as shown in function SNLE of Figure 2, a single test suffices to eliminate candidate SNRE repeats of cardinality greater than α, thus substantially reducing processing time in many cases. We now describe a more sophisticated approach that reduces worst-case complexity to $\Theta(n+\alpha)$ with a negligible effect on actual processing time.

Instead of BWT$_{\boldsymbol{x}}$, we compute an array LAST = LAST$[1..n]$ in which for every $j \in 1..n$, LAST$[j]$ is 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 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 LAST$[j] \leftarrow j$–$jprev$–1, so that LAST$[j]$ takes values in the range $0..\alpha-2$. See Figure 3. Then when function SNLE processes a possibly supernonextendible repeat consisting of $end - start + 1$ substrings 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 $h - \text{LAST}[h] > start$. See Figure 2.

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 if the interval $start..end$ actually is an SNE repeat, it can contain no more than 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

```
— 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
    $i \leftarrow \mathrm{SA}[j]-1$
    if $i \leftarrow 0$ then
        $\mathrm{LAST}[j] \leftarrow \alpha'$
    else
        $letter \leftarrow \boldsymbol{x}[i]$; $jprev \leftarrow lastpos[letter]$
        if $jprev = 0$ or $j - jprev \geq \alpha$ then
            $\mathrm{LAST}[j] \leftarrow \alpha'$
        else
            $\mathrm{LAST}[j] \leftarrow j - jprev - 1$
        $lastpos[letter] \leftarrow j$
```

Figure 3. Preprocessing for Algorithm PSY2 — computing LAST

alphabet is confined to a single byte ($\alpha \leq 256$), the array LAST becomes an array of bytes, just like BWT. (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.)

3 Experimental Results

Experiments were conducted on a diverse selection of files (see Table 1) chosen from `http://www.cas.mcmaster.ca/~bill/strings/`. Tests were conducted using a 2.6 GHz Opteron 885 processor with 2 GB 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 input/output.

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

Table 1. Files used for testing.

Test results are shown in Table 2, where the vertical line separates preprocessing from processing. 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 Kasai *et al.* [11] was used, the fastest one known — according to experiments documented in [15], the first Manzini variant runs almost as fast. Table 2 compares PSY1 with the algorithm of [17]. The algorithm of [7] was not tested because it computes SA twice, and so could not be competitive. Not shown in the table are tests against three variants of PSY1, two of them using heuristics designed to speed up processing, another using a different approach that also achieves $\Theta(n)$ worst case time: on each of the test files listed in Table 1, PSY1 is at least as fast as any of the three. Note that for each program tested, the number of microseconds per letter is generally stable within each file type and not highly variable overall. Averages are not weighted by file size. Tests shown for PSY1 used $p_{min} = 1$; as expected, for larger p_{min} run time was unchanged.

File	SA	LCP	BWT	LAST	PSY1	[17]	PSY2
fibo35	0.898	0.169	0.025	0.031	0.012	0.448	0.009
fibo36	0.886	0.170	0.027	0.033	0.012	0.475	0.007
fss9	0.826	0.154	0.026	0.031	0.014	0.330	0.007
fss10	0.958	0.177	0.025	0.032	0.013	0.469	0.008
periodic AVG	0.892	0.168	0.026	0.032	0.013	0.430	0.008
rand2	0.947	0.188	0.026	0.031	0.017	0.215	0.012
rand21	1.135	0.199	0.025	0.031	0.012	0.122	0.012
random AVG	1.041	0.193	0.025	0.031	0.015	0.169	0.012
ecoli	1.413	0.175	0.025	0.031	0.015	0.155	0.011
chr22	1.635	0.285	0.035	0.040	0.016	0.278	0.012
chr19	1.873	0.333	0.044	0.053	0.016	0.242	0.012
DNA AVG	1.754	0.309	0.035	0.041	0.016	0.225	0.012
prot-a	1.778	0.222	0.027	0.032	0.013	0.211	0.012
prot-b	1.971	0.277	0.034	0.039	0.013	0.247	0.012
protein AVG	1.874	0.249	0.030	0.036	0.013	0.229	0.012
bible	1.417	0.151	0.024	0.030	0.015	0.168	0.012
howto	1.912	0.214	0.035	0.039	0.016	0.219	0.012
mozilla	1.815	0.187	0.032	0.036	0.013	0.139	0.011
English AVG	1.417	0.151	0.024	0.035	0.014	0.175	0.012
AVERAGE	1.390	0.207	0.029	0.035	0.014	0.266	0.011

Table 2. Microseconds per letter used by each run.

4 Discussion

We make the following observations:

* Both new algorithms are very fast, especially on strings that arise in practice: even if SA were to execute 10 times faster, still each algorithm would require less than 5 % of total SA/LCP time.
* Computing LAST for PSY2 requires about 20 % more time than computing BWT for PSY1. Both requirements are small compared to SA/LCP computation time.
* For PSY1 we have computed maximum stack size for each of the test files: for `prot-a` (the worst case) the maximum storage for LB was less than 0.1 % of the $5n$ bytes required for LCP and BWT.

∗ The algorithm of [17] appears to execute 10–15 times slower than PSY1 on real-world files, while requiring $12n$ bytes of storage (SA, inverse of SA, and LCP). (The timing facilities for this algorithm were included in the code kindly provided by the authors.)

∗ Assuming the use of a fast space-efficient SA construction algorithm, LCP construction turns out to be the main obstacle to further improvement, due to both its time and its space requirements.

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 [2,13,21]. It is also useful for duplicate text/document detection [3]. 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 either PSY1 or PSY2 will provide significant algorithmic speed-up over currently-proposed methods [19] 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 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. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* Journal of Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. Apostolico and S. Lonardi: *Off-line compression by greedy textual substitution.* Proceedings of the IEEE, 88(11) 2000, pp. 1733–1744.
3. Y. Berstein and J. Zobel: *Accurate discovery of co-derivative documents via duplicate text detection.* Information Systems, 31 2006, pp. 595–609.
4. G. S. Brodal, R. B. Lyngso, C. N. S. Pederesen, and J. Stoye: *Finding maximal pairs with bounded gap.* Journal of Discrete Algorithms, 1 2000, pp. 77–103.

5. M. Burrows and D. J. Wheeler: *A block sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California, 1994.
6. F. Franek, J. Simpson, and W. F. Smyth: *The maximum number of runs in a string*, in Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms, M. Miller and K. Park, eds., Seoul, Korea, 2003, pp. 36–45.
7. F. Franek, W. F. Smyth, and Y. Tang: *Computing all repeats using suffix arrays*. Journal of Automata, Languages and Combinatorics, 8(4) 2003, pp. 579–591.
8. D. Gusfield: *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*, Cambridge University Press, Cambridge, United Kingdom, 1997.
9. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proceedings of the 30th International Colloquium Automata, Languages and Programming, vol. 2971 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 943–955.
10. S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn: *New approaches for computer analysis of nucleic acid sequences*. Proceedings of the National Academy of Science, 80(18) September 1983, pp. 5660–5664.
11. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, A. Amir and G. M. Landau, eds., vol. 2089 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001, pp. 181–192.
12. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, R. Baeza-Yates, E. Chávez, and M. Crochemore, eds., vol. 2676 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 200–210.
13. J. Larsson and A. Moffat: *Off-line dictionary-based compression*. Proceedings of the IEEE, 88(11) 2000, pp. 1722–1732.
14. M. A. Maniscalco and S. J. Puglisi: *Faster lightweight suffix array construction*, in Proceedings of 17th Australasian Workshop on Combinatorial Algorithms, J. Ryan and Dafik, eds., 2006, pp. 16–29.
15. G. Manzini: *Two space saving tricks for linear time LCP computation*, in Proceedings of 9th Scandinavian Workshop on Algorithm Theory (SWAT '04), T. Hagerup and J. Katajainen, eds., vol. 3111 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2004, pp. 372–383.
16. G. Manzini and P. Ferragina: *Engineering a lightweight suffix array construction algorithm*. Algorithmica, 40 2004, pp. 33–50.
17. K. Narisawa, S. Inenaga, H. Bannai, and M. Takeda: *Efficient computation of substring equivalence classes with suffix arrays*, in Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, B. Ma and K. Zhang, eds., vol. 4580 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 340–351.
18. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms*. ACM Computing Surveys, 39(2) 2007, pp. 1–31.
19. S. W. Shin and S. M. Kim: *A new algorithm for detecting low-complexity regions in protein sequences*. Bioinformatics, 21(2) 2005, pp. 160–170.
20. B. Smyth: *Computing Patterns in Strings*, Pearson Addison-Wesley, Essex, England, 2003.
21. A. Turpin and W. F. Smyth: *An approach to phrase selection for offline data compression*, in Proceedings of the 25th Australasian Computer Science Conference, M. Oudshoorn, ed., 2000, pp. 267–273.