Lempel-Ziv Factorization Using Less Time & Space

Gang Chen, Simon J. Puglisi and W. F. Smyth

Abstract. For 30 years the Lempel-Ziv factorization $\mathrm{LZ}_{\boldsymbol{x}}$ of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ has been a fundamental data structure of string processing, especially valuable for string compression and for computing all the repetitions (runs) in $\boldsymbol{x}$. Traditionally the standard method for computing $\mathrm{LZ}_{\boldsymbol{x}}$ was based on $\Theta(n)$-time (or, depending on the measure used, $O(n \log n)$-time) processing of the suffix tree $\mathrm{ST}_{\boldsymbol{x}}$ of $\boldsymbol{x}$. Recently Abouelhoda *et al.* proposed an efficient Lempel-Ziv factorization algorithm based on an "enhanced" suffix array — that is, a suffix array $\mathrm{SA}_{\boldsymbol{x}}$ together with supporting data structures, principally an "interval tree". In this paper we introduce a collection of fast space-efficient algorithms for LZ factorization, also based on suffix arrays, that in theory as well as in many practical circumstances are superior to those previously proposed; one family out of this collection achieves true $\Theta(n)$-time alphabet-independent processing in the worst case by avoiding tree structures altogether.

Mathematics Subject Classification (2000). Nonnumerical Algorithms 68W05.

Keywords. Lempel-Ziv Factorization, Suffix Array, Suffix Tree, LZ Factorization.

1. Introduction

Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string of length n on an alphabet A of size α. The **LZ factorization** $\mathrm{LZ}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ [22, 44] is a factorization $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ such that each $\boldsymbol{w_j}$, $j \in 1..k$, is

(a) a letter that does *not* occur in $\boldsymbol{w_1 w_2 \cdots w_{j-1}}$; or otherwise
(b) the longest substring that occurs at least twice in $\boldsymbol{w_1 w_2 \cdots w_j}$.

The work of the first and third authors was supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

For the string

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a, \end{array}$$

$\boldsymbol{w_1} = a$, $\boldsymbol{w_2} = b$, $\boldsymbol{w_3} = a$, $\boldsymbol{w_4} = aba$, $\boldsymbol{w_5} = ba$. Typically, integer pairs (POS, LEN) specify the factorization, where POS gives a position in $\boldsymbol{x}$ and LEN the corresponding length at that position (by convention zero if the position contains a "new" letter). The example thus yields

$$(\mathrm{POS}, \mathrm{LEN}) = (1,0), (2,0), (3,1), (4,3), (7,2).$$

$\mathrm{LZ}_{\boldsymbol{x}}$ can be computed from the suffix tree $\mathrm{ST}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ [44], also from an "enhanced" suffix array $\mathrm{SA}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ [1], both in time linear in string length n. [1]

LZ factorization has for many years been of great importance in data compression [33], in particular for the `gzip` (Unix), `winzip` and `pkzip` compression algorithms. However, our immediate motivation for developing new LZ construction algorithms is LZ's central role in computing repetitions in strings, as we now explain.

Periodicity (repetition) in infinite strings was the first topic of stringology [40]; counting and computing the maximum-length adjacent repeating substrings (repetitions) in a finite string was, along with pattern-matching, one of the earliest computational problems on strings to be studied [23, 25]. Given a nonempty string $\boldsymbol{u}$ and an integer $e \geq 2$, we call $\boldsymbol{u}^e$ a **repetition**; if $\boldsymbol{u}$ itself is not a repetition, then $\boldsymbol{u}^e$ is a **proper repetition**. Given a string $\boldsymbol{x}$, a **repetition in** $\boldsymbol{x}$ is a substring

$$\boldsymbol{x}[i..i+e|\boldsymbol{u}|-1] = \boldsymbol{u}^e,$$

where $\boldsymbol{u}^e$ is a proper repetition and neither $\boldsymbol{x}\big[i+e|\boldsymbol{u}|..i+(e+1)|\boldsymbol{u}|-1\big)$ nor $\boldsymbol{x}[i-|\boldsymbol{u}|..i-1]$ equals $\boldsymbol{u}$. Following [39], we say the repetition has **period** $|\boldsymbol{u}|$ and **exponent** e; it can be specified by the integer triple $(i, |\boldsymbol{u}|, e)$. It is well known [23, 4] that the maximum number of repetitions in a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is $\Theta(n \log n)$, and that the number of repetitions in $\boldsymbol{x}$ can be computed in $\Theta(n \log n)$ time [4, 2, 26].

A string $\boldsymbol{u}$ is a **run** iff it is periodic of (minimum) period $p \leq |\boldsymbol{u}|/2$. Thus $\boldsymbol{x} = abaabaabaabaab = (aba)^4 ab$ is a run of period $|aba| = 3$. A substring $\boldsymbol{u} = \boldsymbol{x}[i..j]$ of $\boldsymbol{x}$ is a **run in** $\boldsymbol{x}$ iff it is a run of period p and neither $\boldsymbol{x}[i-1..j]$ nor $\boldsymbol{x}[i..j+1]$ is a run of period p (**nonextendible**). The run $\boldsymbol{u}$ has **exponent** $e = \lfloor |\boldsymbol{u}|/p \rfloor$ and possibly empty **tail** $\boldsymbol{t} = \boldsymbol{x}[i+ep..j]$ (proper prefix of $\boldsymbol{x}[i..i+p-1]$). Thus

$$\begin{array}{rcccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \boldsymbol{x} = & b & a & a & a & b & a & a & b & a & a & b & a & b & a \end{array}$$

contains a run $\boldsymbol{x}[3..12]$ of period $p = 3$ and exponent $e = 3$ with tail $\boldsymbol{t} = a$ of length $t = |\boldsymbol{t}| = 1$. It can also be specified by a triple $(i, j, p) = (3, 12, 3)$, and it includes the repetitions $(aab)^3$, $(aba)^3$ and $(baa)^2$ of period $p = 3$. In general, for

[1] Strictly speaking, for [44] the bound is really $O(n \log n)$, since the method requires a top-down tree traversal that at each of $O(n)$ nodes may consume $O(\log \alpha)$ time — while, in the worst case, α may be of order n. On the other hand, even though the method of [1] uses a tree structure, the traversal is bottom-up and so avoids the $\log \alpha$ factor.

$e = 2$ a run **encodes** $t+1$ repetitions; for $e > 2$, p repetitions. Clearly, computing all the runs in $\boldsymbol{x}$ specifies all the repetitions in $\boldsymbol{x}$.

Runs were introduced by Main [24], who showed how to compute the leftmost occurrence of every run in $\boldsymbol{x} = \boldsymbol{x}[1..n]$ by

(1) computing $\mathrm{ST}_{\boldsymbol{x}}$, the suffix tree of $\boldsymbol{x}$ [43];
(2) using $\mathrm{ST}_{\boldsymbol{x}}$ to compute $\mathrm{LZ}_{\boldsymbol{x}}$ [22, 44];
(3) using $\mathrm{LZ}_{\boldsymbol{x}}$ to compute leftmost runs.

Main's algorithm for step (3) was linear in string length n, and use of Farach's linear-time suffix tree construction algorithm (STCA) [7] enables step (1) also to be performed in linear time. In [17] Kolpakov & Kucherov proved that the maximum number of runs in any string of length n is $O(n)$, then showed (Algorithm KK) how to compute all the runs in $\boldsymbol{x}$ from the leftmost ones in linear time. Thus a $\Theta(n)$-time all-runs algorithm waited only on a truly linear approach to step (2). [2]

In [1] Abouelhoda, Kurtz & Ohlebusch show how to compute $\mathrm{LZ}_{\boldsymbol{x}}$ from a suffix array $\mathrm{SA}_{\boldsymbol{x}}$, augmented with a linear-space "interval tree", rather than from $\mathrm{ST}_{\boldsymbol{x}}$. Even though a tree is used, the time required for Algorithm AKO is truly linear, as remarked in the footnote above. Since there now exist linear-time suffix array construction algorithms (SACAs) [13, 16] that work for large values of n, the goal of a practical and truly linear all-runs algorithm was thus achieved.

In this paper we describe three new algorithms for constructing $\mathrm{LZ}_{\boldsymbol{x}}$, the first two (CPS1 & CPS2) based on suffix array construction, the third (CPS3) based on a kind of inverted file construction (see for example [35]). In particular, CPS1 computes $\mathrm{LZ}_{\boldsymbol{x}}$ in true $\Theta(n)$ time, making use of no tree structures whatever, and thus enabling linear-time all-runs computation.

We describe three variants of CPS1 that execute generally faster and with lower space requirements than either of the algorithms AKO [1] or KK-LZ (a suffix tree-based implementation of Ukkonen's algorithm [42] by Kolpakov & Kucherov specifically designed for alphabet size $\alpha \leq 4$ [19]). Ukkonen's algorithm constructs ST on-line and so permits LZ to be built from subtrees of ST; this gives it an advantage, at least in terms of space, over the fast and compact version of McCreight's STCA [32] due to Kurtz [21]. Note also [34] that the linear-time algorithms [13, 16] for computing $\mathrm{SA}_{\boldsymbol{x}}$ are not, in practice, as fast as other algorithms [31, 29] that have only supralinear worst-case time bounds. Thus in testing AKO and CPS1 we make use of the supralinear SACA [29] that is probably at present overall the fastest in practice.

The second algorithm, CPS2, takes advantage of the fact, illustrated in the above example, that not all of the (POS, LEN) pairs need to be computed. This approach makes use of recent advances [9] in the implementation of range-minimum queries to compute specified (POS, LEN) pairs in constant time. CPS2 requires $O(n \log n)$ worst-case time, but is fast and space-efficient in practice.

[2] Though, since Farach's algorithm is not implementable for large n, this linearity was more theoretical than practical.

Finally we describe CPS3, an algorithm that makes use of inverted files together with partially-constructed "quasi" suffix arrays to compute $\mathrm{LZ}_{\boldsymbol{x}}$.

In Section 2 we describe the new algorithms. Section 3 summarizes the results of experiments that compare the algorithms with each other and with existing algorithms. Section 4 outlines future work.

2. Description of the Algorithms

Given a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet A of size α, we refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as **suffix** i. Then $\mathrm{SA}_{\boldsymbol{x}}$ 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 $j \in 2..n$,

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

Given $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$, $\mathrm{LCP}_{\boldsymbol{x}}$ can be quickly computed in $\Theta(n)$ time using either $13n$ [15] or $9n$ [30] bytes of storage. When the context is clear, we write SA for $\mathrm{SA}_{\boldsymbol{x}}$, LCP for $\mathrm{LCP}_{\boldsymbol{x}}$. For example:

	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	
$\mathrm{LCP}_{\boldsymbol{x}} =$	-1	1	1	3	3	0	2	2	-1

2.1. CPS1

In the course of a left-to-right scan of the SA/LCP arrays, Algorithm CPS1 takes advantage of two simple observations:

(O1) Every increase in consecutive LCP values marks the beginning of a collection of two or more repeating substrings in $\boldsymbol{x}$ with the same LCP. More precisely, if $\mathrm{LCP}[i_1] < \mathrm{LCP}[i_1{+}1]$, then the suffixes $j = \mathrm{SA}[i_1]$ and $j' = \mathrm{SA}[i_1{+}1]$ have a longest common prefix of length $\mathrm{LCP}[i_1{+}1]$ that is not shared by $\mathrm{SA}[i_1{-}1]$ nor by any preceding position in SA.

(O2) Every decrease in consecutive LCP values marks the end of a collection of two or more repeating substrings with the same LCP. More precisely, if $\mathrm{LCP}[i_2] > \mathrm{LCP}[i_2{+}1]$, then the suffixes $j = \mathrm{SA}[i_2{-}1]$ and $j' = \mathrm{SA}[i_2]$ have a longest common prefix of length $\mathrm{LCP}[i_2]$ that is not shared by $\mathrm{SA}[i_2{+}1]$ nor by any subsequent position in SA.

In view of observation (O1), CPS1 pushes (onto a stack S) each position i_1 that marks the beginning in SA of a sequence of suffixes all with lcp of length at least $\mathrm{LCP}[i_1{+}1]$. When, sooner or later in the left-to-right traversal of SA, the corresponding LCP value is found to fall, the position i_2 of the fall marks, by observation (O2), the end of at least one sequence of two or more suffixes sharing the same lcp — in particular, the sequence beginning at the position most recently pushed onto S. Depending on the extent of the drop in LCP, longer sequences of

suffixes (represented by positions previously pushed onto S) may also terminate at i_2; thus the stack is popped until a stacked position i is found for which $\mathrm{LCP}[i] \leq \mathrm{LCP}[i_2{+}1]$.

```
  — Using SA_x and LCP_x, compute POS[1..n] and LEN[1..n].
i1 ← 1; i2 ← 2; i3 ← 3
while i3 ≤ n+1 do
  — Identify the next position i2 < i3 with LCP[i2] > LCP[i3].
    while LCP[i2] ≤ LCP[i3] do
        push(S, i1); i1 ← i2; i2 ← i3; i3 ← i3+1
  — Backtrack using the stack S to locate the first i1 < i2 such that
  — LCP[i1] < LCP[i2], at each step setting the larger position in POS
  — corresponding to equal LCP to point leftwards to the smaller one,
  — if it exists; if not, then POS[i] ← i.
    q ← SA[i2]; ℓ2 ← LCP[i2]
    assign(POS, LEN, p, q)
    while LCP[i1] = ℓ2 do
        i1 ← pop(S)
        assign(POS, LEN, p, q)
    SA[i1] ← q
  — Reset pointers for the next stage.
    if i1 > 1 then
        i2 ← i1; i1 ← pop(S)
    else
        i2 ← i3; i3 ← i3+1

procedure assign(POS, LEN, p, q)
p ← SA[i1]
if p < q then
    POS[q] ← p; LEN[q] ← ℓ2; q ← p
else
    POS[p] ← q; LEN[p] ← ℓ2
```

FIGURE 1. Algorithm CPS1: computing $\mathrm{LZ}_{\boldsymbol{x}}$

Each sequence $i_1, i_1{+}1, \ldots, i_2$ identified by this process (which for convenience we relabel $s_1, s_2, \ldots, s_k$ for some $k \geq 2$) specifies corresponding positions (suffixes)

$$p_1 = \mathrm{SA}[s_1],\ p_2 = \mathrm{SA}[s_2],\ \ldots,\ p_k = \mathrm{SA}[s_k]$$

in $\boldsymbol{x}$. In order to compute values in POS, it suffices to process these positions in pairs $p_{h-1}, p_h,\ h = k, k-1, \ldots, 2.$ in descending order, assigning $POS[p] \leftarrow q$, where p is the greater of p_{h-1}, p_h and q the lesser. At the same time, to ensure that a leftmost position in $\boldsymbol{x}$ is always available, we must effectively at each step implement the replacement $\mathrm{SA}[s_{h-1}] \leftarrow q$. For each position p in POS assigned, the corresponding value of $\mathrm{LEN}[p]$ will just be $\mathrm{LCP}[i_2]$.

After each sequence of repeating substrings is processed. corresponding to the current lcp, the pointer values are reset (generally by $i_2 \leftarrow i_1$ and popping S into i_1) to determine whether another sequence of POS/LEN pairs should be processed at this position.

This processing does not guarantee that, for equal LCP (LEN), each corresponding position in POS necessarily points to the *leftmost* occurrence in $\boldsymbol{x}$, as normally required for LZ factorization; however, the Main and KK algorithms do not require this property for their correct functioning, they require only that each position in POS should point left. Similarly, the leftmost occurrence is not required for most data compression applications. In other terminology, what is in fact computed by CPS1 is a **quasi suffix array** (QSA) [10].

In order to implement the processing described above, CPS1 uses three pointers i_1, i_2, i_3 to positions in SA that at each step of the algorithm satisfy the invariant $i_1 < i_2 < i_3$. For the example string

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a, \end{array}$$

Figure 2 shows how these pointers are manipulated as $\boldsymbol{x}$ is scanned. Note that it may not be true that $i_2 = i_1+1$, nor that $i_3 = i_2+1$. Note also that the shaded areas in the SA and LCP arrays, once made use of in a POS/LEN calculation, are never thereafter used again, a fact that can be exploited, as described below, to reduce CPS1's space requirement.

The basic CPS1 algorithm, that we call CPS1a, is shown in Figure 1.

We have observed that none of the position pointers i_1, i_2, i_3 in CPS1 will ever point to any position i in SA such that $\mathrm{POS}\big[\mathrm{SA}[i]\big]$ has been previously set. It follows that the storage for SA and LCP can be dynamically reused to specify the location and contents of the array POS, thus saving $4n$ bytes of storage — neither the Main nor the KK algorithm mentioned above requires SA/LCP; moreover, these data structures are not generally required in string compression applications. Reuse of SA/LCP is easily accomplished by inserting the instruction $i_2 \leftarrow i_1$ at the beginning of the second inner **while** loop of Figure 1, then replacing

$$\begin{aligned} &\mathrm{POS}[q] \leftarrow p \ \text{ by } \ \mathrm{SA}[i_2] \leftarrow q;\ \mathrm{LCP}[i_2] \leftarrow p \\ &\mathrm{POS}[p] \leftarrow q \ \text{ by } \ \mathrm{SA}[i_2] \leftarrow p;\ \mathrm{LCP}[i_2] \leftarrow q \end{aligned}$$

POS can then be computed by a straightforward in-place compactification of SA and LCP into SA (now redefined as POS). We call this second algorithm CPS1b.

But more storage can be saved. Remove all reference to LEN from CPS1b, so that it computes only POS and in particular allocates no storage for LEN. Then, after POS is computed, the space previously required for LCP becomes free and can be reallocated to LEN. Observe that only those positions in LEN that are required for the LZ factorization need to be computed, so that the total computation time for LEN is $\Theta(n)$. In fact, without loss of efficiency, we can avoid computing LEN as an array and compute it only when required; given a sentinel

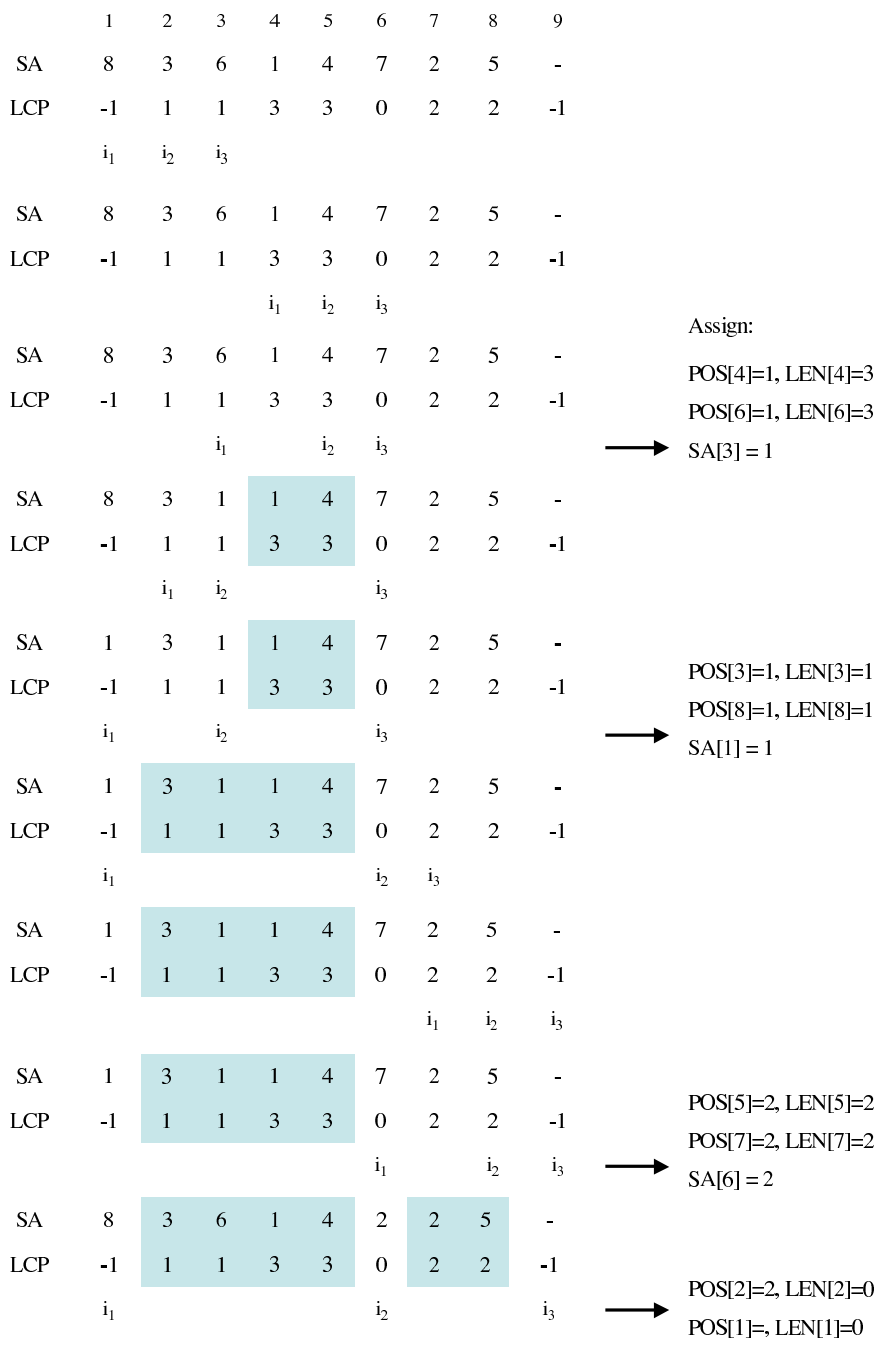

FIGURE 2. Execution of CPS1 on $\boldsymbol{x} = abaababa$

value POS[n+1] = \$, the simple function of Figure 3 computes LEN corresponding to POS[i]. We call the third version CPS1c.

```
function LEN(x, POS, i)
j ← POS[i]
if j = i then
    LEN ← 0
else
    ℓ ← 1
    while x[i+ℓ] = x[j+ℓ] do
        ℓ ← ℓ+1
    LEN ← ℓ
```

FIGURE 3. Computing LEN corresponding to POS[i]

Since at least one position in POS is set at each stage of the main **while** loop, it follows that the execution time of CPS1 is linear in n. For CPS1a space requirements total $17n$ bytes (for $\boldsymbol{x}$, SA, LCP, POS & LEN) plus $4s$ bytes for a stack of maximum size s. For $\boldsymbol{x} = a^n$, $s = n$, and as we discover in Section 3, s can be $n/2$ or more for artificially run-rich strings; however, in practical cases s will be close to the maximum height of $\mathrm{SA}_{\boldsymbol{x}}$ and so $s \in O(\log_\alpha n)$ [14].

For CPS1b and CPS1c, the maximum space required is $13n$ and $9n$ bytes, respectively, plus stack. Observe that for CPS1a and CPS1b the original (and somewhat faster) method [15] for computing LCP can be used, since it requires $13n$ bytes of storage, not greater than the total space requirements of these two variants. For CPS1c, however, to achieve $9n$ bytes of storage, the Manzini variant [30] for computing LCP must be used. In fact, as described below, we test two versions of CPS1c, one that uses the original LCP calculation (therefore requiring $13n$ bytes, but no additional space for the stack), the other using the Manzini variant (hence requiring $9n$ bytes plus stack).

We remark that all versions of Algorithm CPS1 can easily be modified (with the introduction of another stack) to compute the LZ factorization in its usual form.

2.2. CPS2

An undesirable aspect of the CPS1 family of algorithms described above for LZ factorization is that they output an item for each position in the string, regardless of whether a factor actually begins there or not. Ideally we would like the output to contain only information about the positions where factors start; the difficulty is that it is hard to tell in advance where the factors will begin. The pseudocode in Figure 4 specifies a function that computes LZ factor information for an arbitrary position in the string $\boldsymbol{x}$. It makes use only of SA and a data structure $\mathrm{RMQ}_{\mathrm{SA}}$ for answering **range minimum queries** (RMQs) on SA [12, 3]. $\mathrm{RMQ}_{\mathrm{SA}}(i, j)$ provides the index of the minimum value among $\mathrm{SA}[i], \mathrm{SA}[i+1], \ldots, \mathrm{SA}[j]$ (or the leftmost

such index should more than one occurrence of the minimum be present in the range). It has recently been shown [9] that RMQs can be implemented using less than n bytes, while with appropriate preprocessing any RMQ can be answered in constant time.

```
— Using SA_x and RMQ_SA compute the position
— and length of the LZ factor beginning at i in x.
function lzfactor(x, SA, i)
match ← i
lb ← 1; rb ← n; j ← i
repeat
    (lb, rb) ← refine(lb, rb, j−i, x[j])
    min ← SA[RMQ_SA(lb, rb)]
    if min < i then
        match ← min; j ← j+1
until min ≥ i or j > n
return (match, j−i)
```

FIGURE 4. Algorithm to find the length and previous occurrence of an LZ factor at a given position i in string $\boldsymbol{x}$

CPS2 maintains the invariant that interval SA[$lb..rb$] contains *all* the suffixes prefixed with $\boldsymbol{x}[i..j-1]$ and that at least one of the suffixes in that range begins at some position $p < i$ in $\boldsymbol{x}$. This condition is enforced by the `refine` function in concert with the range minimum query using $\mathrm{RMQ}_{\mathrm{SA}}$. Given a match of $\boldsymbol{x}[i..j-1]$ of length $j-i$ with

$$\boldsymbol{x}\big[\mathrm{SA}[\ell]..\mathrm{SA}[\ell]+j-1\big]$$

over a maximum-length range of positions $\ell \in lb..rb$ of SA, `refine` computes a maximum-length subinterval of $lb..rb$ that matches $\boldsymbol{x}[i..j]$. Narrowing of the interval only occurs if at least one of the suffixes in that range begins at some position $p < i$, which is determined from $\mathrm{RMQ}_{\mathrm{SA}}(lb, rb)$. Note that `refine` will never return an empty interval because we are searching using a suffix of the string itself as a pattern, so a suffix prefixed with $\boldsymbol{x}[i..j]$ is guaranteed to be found.

One way to implement `refine` is via a linear scan of SA[$lb..rb$] to determine the maximum subarray SA[$nlb..nrb$] such that

$$\boldsymbol{x}[\mathrm{SA}[nlb]] = \boldsymbol{x}[\mathrm{SA}[nlb+1]] = \ldots = \boldsymbol{x}[\mathrm{SA}[nrb]] = \boldsymbol{x}[j].$$

This requires $O(rb-lb)$ time per call. A more efficient method is to use two binary searches to determine the upper and lower bounds. We can use binary search because the j^{th} letter of each suffix in SA[$lb..rb$] is in lexorder. This is really the SA search algorithm of Manber and Myers [27] being used incrementally. Each call to `refine` now takes $O(\log n)$ time.

Alternatively we could search using the $O(|\Sigma|)$ algorithm of [1], but doing so requires the LCP array, which is precisely what we are trying to avoid. The

so-called "backward search" algorithm [8, 38] for suffix arrays is seemingly of no use either, as it searches the pattern right to left, and the pattern in our case is a suffix of the string. However, backward search can be used in CPS2 to produce the LZ factorization of the *reverse* string, which may be acceptable for some applications, such as computing runs [17] and repeats [20, 18]. This way `refine` could be implemented in $O(\log|\Sigma|)$ time.

Every line of the function `lzfactor` executes in constant time except the call to `refine`. To produce the entire LZ factorization we make a total of at most $n-1$ calls to `refine`. This gives a total running time of $O(n \log n)$ (if the $O(\log n)$ version of `refine` is used). Observe that because RMQ gives us the minimum value each time, CPS2 associates the leftmost occurrence with the starting position of each factor.

output $(1, 1)$
$i \leftarrow 2$
while $i \leq n$ **do**
    $(\text{POS}, \text{LEN}) \leftarrow$ `lzfactor`$(\boldsymbol{x}, \text{SA}, i)$
    output (POS, LEN)
    $i \leftarrow i+\text{LEN}$

FIGURE 5. Algorithm CPS2 for computing the LZ-factorization

2.3. CPS3

This algorithm combines the idea of a QSA [10], mentioned above, with that of a q-gram [41] — that is, a substring of length q. A preprocessing stage of CPS3 (Figure 6) builds a QSA, called QSA(q), in which matches between positions i and QSA[i] in $\boldsymbol{x}$ are restricted to at most some specified length q. The extra space required for preprocessing is $4|\Sigma|^q$ bytes, as we keep track of the last position of occurrence of every substring of length q. The resulting QSA can be thought of as an inverted file for $\boldsymbol{x}$ based on t-grams, $t \in 1..q$.

As shown in Figure 7, CPS3 uses QSA(q) to compute the (POS, LEN) pairs required for the LZ factorization. For negative QSA[i] values, the position POS is known, and less than q letter comparisons are required in the function `match` to determine LEN. When QSA[i] is positive, however, the length LEN of the longest match could exceed q, and so the QSA chain needs to be traversed left-to-right to locate the longest match. With the value of q, CPS3 offers a space/time tradeoff: the bigger the value of q, the less time spent traversing chains in the QSA (as the chains are shorter), but the greater the size of array *rightmost* in Figure 6. Note that if no factor is longer than q, CPS3 computes the LZ factorization in its traditional form: POS is necessarily the leftmost position in $\boldsymbol{x}$ that achieves the match of maximum length LEN with the current position i.

```
  — For every nonempty string z on Σ of length at most q,
  — initialize its rightmost position in x to zero.
for i ← 1 to q do
    ∀ string z of length i do
        rightmost[z] ← 0
  — Compute QSA(q).
for i ← 1 to n do
    j ← 0
    while j < q and i+j ≤ n do
        ℓ ← rightmost[x[i..i+j]]
          — The new position i is now rightmost.
        rightmost[x[i..i+j]] ← i
          — Positions for which the maximum match is less
          — than q letters are "easy": mark them negative.
        if j < q−1 then
            QSA[i] ← −ℓ
        else
            QSA[i] ← ℓ
        j ← j+1
```

FIGURE 6. CPS3 preprocessing — computing QSA(q)

3. Experimental Results

We implemented the three versions of CPS1 described above, with two variants of CPS1c; we call them `cps1a`, `cps1b`, `cps1c` ($13n$-byte LCP calculation) and `cps1c′` ($9n$-byte LCP calculation). We also implemented `cps2` with the $O(\log n)$ `refine` function described previously, and `cps3`. We had `cps3` choose the value of q for each file, so that the extra space used in preprocessing was around n bytes.

Finally, we implemented the other SA-based LZ-factorization algorithm, `ako` of [1]. The implementation `kk-lz` of Kolpakov and Kucherov's algorithm was obtained from [19]. All programs were written in C or C++. We are confident that all implementations tested are of high quality.

As indicated in Table 1, experiments were conducted on four main classes of input strings:

- strings that do not occur in practice, but that nevertheless are of interest: those with many runs (Fibonacci strings, binary strings constructed in [11]) and those with very few (random strings on small and large alphabets);
- DNA strings on alphabet $\{a, c, g, t\}$ that Algorithm KK-LZ was specifically tailored to;
- protein sequences on an alphabet of 20 letters;
- strings on large alphabets (English-language, ASCII characters).

All experiments were conducted on a 2.6GHz AMD Opteron processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow)

```
output (1, 1)
i ← 2
while i ≤ n do
    if QSA[i] < 0 then
        — Left maximum match is at |QSA[i]| of length < q.
        POS ← −QSA[i]; LEN ← match(x[QSA[i]..n], x[i, n])
    else
        — Left maximum match may exceed q.
        LEN ← 0; i′ ← i
        repeat
            i′ ← QSA[i′]; ℓ ← match(x[i′..n], x[i, n])
            if ℓ ≥ LEN then POS ← i′; LEN ← ℓ
        until QSA[i′] ≤ 0
    output (POS, LEN)
    — Locate next i.
    if LEN = 0 then
        i ← i+1
    else
        i ← i+LEN
```

FIGURE 7. CPS3 computes (POS, LEN) pairs from QSA(q)

running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) executed with the -O3 option. The running times shown in Table 3 are the average of four runs and do not include time spent reading input files. Times were recorded with the standard C `getrusage` function. Table 2 isolates the time spent just on SA/LCP construction; comparison with Table 3 shows that a very high proportion of the time spent by the CPS1 family of algorithms is devoted to these preprocessing activities.

Table 4 shows memory usage over the experiments performed as measured by the `memusage` command available with most Linux distributions.

In Table 3 times given for the `cps1` implementations and `ako` include that required for SA and LCP array construction; `cps2` times include times for SA and RMQ construction. The implementation of `kk-lz` is only suitable for strings on small alphabets ($|\Sigma| \leq 4$) so times are only given for some files. Results are not given for `ako` on some files because the memory required exceeded the capacity of the test machine. Results are not given for `cps3` on some large alphabet files because the runtime exceeded 1000 seconds, at which point we abandoned the experiment. Files `chr22` and `chr1819` were originally on an alphabet of five symbols A,C,G,T,N which we reduced by replacing occurrences of N with random selection of the other four symbols. The N's represent ambiguities in the sequencing process.

We conclude:

(1) The tailored KK algorithm remains the algorithm of choice for DNA strings of moderate size.

(2) For other strings encountered in practice, CPS1b is consistently faster than AKO except for very large alphabets (perhaps an atypical result); it also uses substantially less space, especially on run-rich strings.
(3) Overall, and especially for strings on alphabets of size greater than 4, CPS1c′ is probably preferable since it will be more robust for main-memory use on very large strings: its storage requirement is consistently low (about 50% greater than that of CPS2, just over half that of CPS1a, generally less than half that of AKO) and it is only 25–30% slower than CPS1b, generally much faster than CPS2.
(4) If memory is especially tight, CPS2 offers predictable runtimes on all types of data and uses only $6n$ bytes. It is usually around 30% slower than CPS1c but is faster on strings having very few factors. For a similar memory cost CPS2 gives much more stable runtimes than CPS3.
(5) On files with very small numbers of factors, CPS2 and CPS3 perform best. This is expected as they, in some sense, only do work proportional to the number of factors.
(6) The $|\Sigma|^q$ term in the space required for preprocessing makes CPS3 very sensitive to $|\Sigma|$ and n. Generally, large $|\Sigma|$ forces q to be small, which in turn means chains in QSA are larger and more chain traversals and letter comparisons are required per factor. If q is able to be picked close to the average factor length, CPS3 performs well – unfortunately even a moderate size $|\Sigma|$ will preclude this if memory usage is to remain within acceptable limits.

4. Discussion

The algorithms presented here make use of full-size suffix arrays, but there have been many "succinct" or "compressed" suffix structures proposed in the literature [28], that make use of as little as n bytes of storage. We would like to explore the use of such structures in this context, as well as the use of compressed inverted files.

More generally, we remark that all known algorithms that compute runs or repetitions need to compute all the information required to compute **repeats** — that is, not necessarily adjacent repeating substrings. All these algorithms compute some form of suffix structure that implicitly specifies all the repeats; runs/repetitions are then computed by some sort of refinement of the repeats. Since runs generally occur sparsely in strings [17], much less frequently than repeats, it seems that they should somehow be computable directly with less heavy machinery. Recent results [11, 37, 6] may suggest more economical methods.

This study also exposes the relatively high cost of computing the LCP array for some strings. For applications where the LCP array is required, suffix arrays can only be a convincing substitute for suffix trees if LCP computation can be speeded up. One perhaps fruitful line of investigation might be to determine which of the fast SACAs [34] could be adapted to output LCP information also, or in lieu of

TABLE 1. Description of the data set used in experiments.

String	Size (bytes)	Σ	# factors	max	Description
fibo35	9227465	2	34	3524578	35th Fibonacci string (see [39])
fibo36	14930352	2	35	5702887	36th Fibonacci string
fss9	2851443	2	40	1217712	9th run rich string of [11]
fss10	12078908	2	44	5158310	10th run rich string of [11]
random2	8388608	2	385232	42	Random string, small alphabet
random21	8388608	21	1835235	9	Random string, larger alphabet
ecoli	4638690	4	432791	2805	E.Coli Genome
chr22	34553758	4	2554184	1768	Human Chromosome 22
chr19	63811651	4	4411679	3397	Human Chromosome 19
chr1819	139928804	4	9560771	3397	Human Chromosomes 18 & 19
prot-a	16777216	23	2751022	6699	Small Protein dataset
prot-b	33554432	24	5040051	16190	Medium Protein dataset
prot-c	67108864	24	8391184	16190	Large Protein dataset
bible	4047392	62	337558	549	King James Bible
howto	39422105	197	3063929	70718	Linux Howto files
mozilla	51220480	256	3823511	41323	Mozilla binaries
rfc	116421901	120	5656068	3317	IETF Request for comments

TABLE 2. Runtime in milliseconds for suffix array construction and LCP computation.

String	`saca`	`lcp13n`	`lcp9n`
fibo35	5530	2130	3090
fibo36	10440	3510	5000
fss9	1490	660	960
fss10	8180	2810	4070
rand2	2960	2360	3030
rand21	2840	2620	3250
ecoli	1570	1340	1700
chr22	14330	12450	16190
chr19	28400	25730	31840
chr1819	74210	70110	77470
prot-a	6170	5230	6660
prot-b	13580	12460	14720
prot-c	29680	27650	31460
bible	1140	1020	1270
howto	12080	11750	14490
mozilla	12850	13790	17320
rfc	40680	39540	49590

TABLE 3. Runtime in milliseconds for various LZ factorization algorithms.

String	cps1a	cps1b	cps1c	cps1c′	cps2	cps3	ako	kk-lz
fibo35	9360	8560	9240	10200	9190	2960	12870	10060
fibo36	16730	15420	16240	17730	16050	4820	23160	18680
fss9	2680	2430	2690	2990	2570	910	3740	1270
fss10	13240	12170	13390	14650	12730	4030	17890	7850
rand2	6950	6130	7010	7680	15340	6830	9920	9820
rand21	7100	6270	7130	7760	11320	7090	7810	–
ecoli	3800	3350	3830	4190	4280	2270	4740	1610
chr22	35240	30320	36480	40220	46580	41790	65360	18240
chr19	70030	61230	71910	78020	93480	128130	–	40420
chr1819	188410	162760	187290	194650	248710	304400	–	105640
prot-a	14780	12990	14920	16350	26180	72190	17070	–
prot-b	33530	29470	34150	36410	57880	212410	38810	–
prot-c	73460	64640	75980	79790	129410	102270	–	–
bible	2930	2540	2970	3220	6950	162020	3670	–
howto	32150	27750	33760	36500	79850	–	23830	–
mozilla	36630	31330	39860	43390	96730	–	–	–
rfc	107280	92910	119630	129680	255700	–	–	–

TABLE 4. Peak memory usage in bytes per input symbol for the LZ factorization algorithms.

String	cps1a	cps1b	cps1c	cps1c′	cps2	cps3	ako	kk-lz
fibo35	19.5	15.5	13.0	11.5	6.0	5.9	26.9	19.9
fibo36	19.5	15.5	13.0	11.5	6.0	5.6	26.9	20.8
fss9	19.1	15.1	13.0	11.1	6.0	6.5	25.4	21.3
fss10	19.1	15.1	13.0	11.1	6.0	5.7	25.4	22.5
rand2	17.0	13.0	13.0	9.0	6.0	6.0	17.0	11.8
rand21	17.0	13.0	13.0	9.0	6.0	5.5	17.0	–
ecoli	17.0	13.0	13.0	9.0	6.0	6.2	17.0	11.0
chr22	17.0	13.0	13.0	9.0	6.0	5.7	17.0	11.1
chr19	17.0	13.0	13.0	9.0	6.0	6.4	–	11.1
chr1819	17.0	13.0	13.0	9.0	6.0	5.6	–	10.7
prot-a	17.2	13.2	13.0	9.2	6.0	5.3	39.0	–
prot-b	17.1	13.1	13.0	9.1	6.0	5.1	40.0	–
prot-c	17.0	13.0	13.0	9.0	6.0	7.1	–	–
bible	17.0	13.0	13.0	9.0	6.0	5.3	17.0	–
howto	17.0	13.0	13.0	9.0	6.0	–	17.0	–
mozilla	17.7	13.7	13.0	9.7	6.0	–	–	–
rfc	17.0	13.0	13.0	9.0	6.0	–	–	–

this, information that can be used later to expedite LCP array computation. This line of research was essentially discontinued with the publication of [15].

Acknowledgment

We thank Johannes Fischer for making his RMQ code available to us.

References

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

[2] Alberto Apostolico & Franco P. Preparata, **Optimal off-line detection of repetitions in a string**, *Theoret. Comput. Sci. 22* (1983) 297–315.

[3] Michael A. Bender & Martin Farach-Colton, **The LCA problem revisited**, *Latin American Theoretical Informatics* (2000) 88–94.

[4] Maxime Crochemore, **An optimal algorithm for computing the repetitions in a word**, *Inform. Process. Lett. 12–5* (1981) 244–250.

[5] Jean-Pierre Duval, Roman Kolpakov, Gregory Kucherov, Thierry Lecroq & Arnaud Lefebvre, **Linear-time computation of local periods**, *Theoret. Comput. Sci. 326–1-3* (2004) 229–240.

[6] Kangmin Fan, Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A new periodicity lemma**, *SIAM J. Discrete Math. 20–3* (2006) 656–668.

[7] Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc.* 38^{th} *IEEE Symp. Found. Computer Science* (1997) 137–143.

[8] Paolo Ferragina & Giovanni Manzini, **Opportunistic data structures with applications**, *Proc.* 41^{st} *IEEE Symp. Found. Computer Science* (2000) 390–398.

[9] Johannes Fischer & Volker Heun, **Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE**, *Proc.* 17^{th} *Annual Symp. Combinatorial Pattern Matching*, M. Lewenstein & G. Valiente (eds.) (2006) 36–48.

[10] Frantisek Franek, Jan Holub, W. F. Smyth & Xiangdong Xiao, **Computing quasi suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 593–606.

[11] Frantisek Franek, R. J. Simpson & W. F. Smyth, **The maximum number of runs in a string**, *Proc.* 14^{th} *Australasian Workshop on Combinatorial Algs.*, M. Miller & K. Park (eds.) (2003) 26–35.

[12] Dov Harel & Robert E. Tarjan, **Fast algorithms for finding nearest common ancestors**, *SIAM J. Computing 13–2* (1984) 338–355.

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

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

[15] T. Kasai, G. Lee, H. Arimura, S. Arikawa & K. Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc.* 12^{th} *Annual Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181–192.

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

[17] Roman Kolpakov & Gregory Kucherov, **On maximal repetitions in words**, *J. Discrete Algs. 1* (2000) 159–186.

[18] Roman Kolpakov & Gregory Kucherov, **Finding repeats with fixed gap**, *Proc. Seventh Symposium on String Processing & Information Retrieval*, (2000) 162–168.

[19] Roman Kolpakov & Gregory Kucherov, `http://bioinfo.lifl.fr/mreps/`.

[20] Roman Kolpakov & Gregory Kucherov, **Finding approximate repetitions under Hamming distance**, *Theoret. Comput. Sci. 303–1* (2003) 135–156.

[21] Stefan Kurtz, **Reducing the space requirement of suffix trees**, *Software, Practice & Experience 29–13* (1999) 1149–1171.

[22] Abraham Lempel & Jacob Ziv, **On the complexity of finite sequences**, *IEEE Trans. Information Theory 22* (1976) 75–81.

[23] André Lentin & Marcel P. Schützenberger, **A combinatorial problem in the theory of free monoids**, *Combinatorial Mathematics & Its Applications*, R. C. Bose & T. A. Dowling (eds.), University of North Carolina Press (1969) 128–144.

[24] Michael G. Main, **Detecting leftmost maximal periodicities**, *Discrete Applied Maths. 25* (1989) 145–153.

[25] Michael G. Main & Richard J. Lorentz, *An $O(n \log n)$ Algorithm for Recognizing Repetition*, Tech. Rep. CS-79–056, Computer Science Department, Washington State University (1979).

[26] Michael G. Main & Richard J. Lorentz, **An $O(n \log n)$ algorithm for finding all repetitions in a string**, *J. Algs. 5* (1984) 422–432.

[27] Udi Manber & Gene Myers, **Suffix arrays: a new method for on-line string searches**, *SIAM J. Computing 22–5* (1993) 935–948.

[28] Veli Mäkinen & Gonzalo Navarro, **Compressed full-text indices**, *ACM Computing Surveys* (2006) to appear.

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

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

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

[32] Edward M. McCreight, **A space-economical suffix tree construction algorithm**, *J. Assoc. Comput. Mach. 32–2* (1976) 262–272.

[33] Mark Nelson & Jean-loup Gailly, *The Data Compression Book*, M&T Books (1995) 541 pp.

[34] Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys* (2007) to appear.

[35] Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **Inverted files versus suffix arrays for in-memory pattern matching**, *Proc.* 13th *Symposium on String Processing & Information Retrieval* (2006) 122–133.

[36] Wojciech Rytter, **Grammar compression, LZ-encodings, and string algorithms with implicit input**, *Proc.* 31st *Internat. Colloq. Automata, Languages & Programming* (2004) 15–27.

[37] Wojciech Rytter, **The number of runs in a string: improved analysis of the linear upper bound**, *Proc.* 23rd *Symp. Theoretical Aspects of Computer Science*, B. Durand & W. Thomas (eds.), LNCS 2884, Springer-Verlag (2006) 184–195.

[38] J.S. Sim, D.K. Kim, H. Park & K. Park, **Linear-time search in suffix arrays**, *Proc.* 14th *Australasian Workshop on Combinatorial Algs.* (2003) 139–146.

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

[40] Axel Thue, **Über unendliche zeichenreihen**, *Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7* (1906) 1–22.

[41] Esko Ukkonen, **Approximate string-matching with q-grams and maximal matches**, *Theoret. Comput. Sci. 92* (1992) 191–211.

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

[43] Peter Weiner, **Linear pattern matching algorithms**, *Proc. 14th Annual IEEE Symp. Switching & Automata Theory* (1973) 1–11.

[44] Jacob Ziv & Abraham Lempel, **A universal algorithm for sequential data compression**, *IEEE Trans. Information Theory 23* (1977) 337–343.

Gang Chen
Department of Computing & Software
McMaster University
Hamilton, Ontario, Canada L8S 4K1
e-mail: `cheng4@mcmaster.ca`

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

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

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