Computing quasi suffix arrays*

František Franěk[1] W. F. Smyth[1,2] Xiangdong Xiao[1]
Jan Holub[1,3]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
smyth@mcmaster.ca or franek@mcmaster.ca
www.cas.mcmaster.ca/cas/research/groups.html

[2] School of Computing, Curtin University, GPO Box U-1987
Perth WA 6845, Australia

[3] Department of Computer Science and Engineering, Faculty of Electrical Engineering,
Czech Technical University, Karlovo náměstí 13, CZ-121 35 Praha 2, Czech Republic

Abstract

We introduce quasi suffix arrays as a generalization of suffix arrays for character strings. We show that a quasi suffix array encodes enough of the structure of the string to be a useful construct for many applications where the full power of suffix arrays is not necessary, notably in problems that do not require lexicographical order; for example, pattern-matching or calculation of repeating substrings. We are interested in quasi suffix arrays because we believe that they can be computed by simple, fast, space-efficient algorithms. As a first step toward this goal, we describe a family DIST of algorithms (inspired by Crochemore's repetitions algorithm) that compute the quasi suffix array in the average-case in $O(|\boldsymbol{x}| \log |\boldsymbol{x}|)$ time, where $\boldsymbol{x}$ is the input string. Based on experiments conducted by one of us (Xiao), it appears that in practice the best DIST algorithms execute faster than all suffix tree and most suffix array construction algorithms. Though

*Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

at this time we can only prove that the average-case complexity is $O(|\boldsymbol{x}| \log |\boldsymbol{x}|)$, tests carried out by one of us (Holub) strongly suggest that the worst-case complexity should be the same as the average-case time complexity; at the same time, the possibility exists that the average-case time complexity of the best DIST algorithms is actually $O(|\boldsymbol{x}|)$, at least for strings on large alphabets.

1 Introduction

In string processing and pattern-matching problems on an ordered alphabet, a collection of closely-related standard data structures are frequently employed to promote algorithmic efficiency: suffix trees [22], directed acyclic word graphs (DAWGs and CDAWGs) [8, 10], and suffix arrays [15], in particular. This paper describes a simple data structure, closely related to suffix arrays, that efficiently specifies all the distinct substrings of a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Unlike the other structures, however, ours does not require dependence on an ordering of the alphabet.

Based on the earlier idea of a Patricia trie [17], suffix trees were introduced by Weiner [22], who also described the first of several worst-case $O(n \log n)$-time algorithms [16, 21] for their construction. (The claim of $O(n)$ time for these algorithms depends on choosing to suppose that the alphabet is fixed.) More recently, for the common case of an "indexed" alphabet (that is, treatable as a sequence $1, 2, \ldots, \alpha \in O(n)$ of integers), a genuine $\Theta(n)$-time suffix tree construction algorithm was discovered [11], but its space requirements are very large and the associated overhead so significant, that apparently the $O(n \log n)$ algorithms are faster in practice. Another algorithm not generally recognized as a suffix tree construction algorithm is Crochemore's algorithm for the computation of all repetitions in $\boldsymbol{x}$ [9]. Indeed this algorithm is particularly important for this paper: not only does it effectively compute suffix trees in $O(n \log n)$ time if its refinement process is allowed to continue to its natural conclusion, but moreover the refinements it performs are the basis of our new algorithms.

Suffix trees are important and useful in many pattern-matching contexts [2, 5, 19, 3, 7], as well as for data compression [4], but the space requirement of a suffix tree $T_{\boldsymbol{x}}$ is inevitably many times that of $\boldsymbol{x}$ itself — a serious problem when n is large, say several tens of millions to billions of letters. In such cases the time advantage resulting from suffix tree use can be lost because of the memory swaps or disk accesses necessitated by a huge data structure. In an effort to reduce this disadvantage, clever storage mechanisms have been devised [14] as well as alternative structures, such as DAWGs [8, 10, 6, 12] and suffix arrays [15]. Of these approaches, by far the most effective for storage reduction is the use of suffix arrays, that require storage of only kn integers, where k may vary from 1 to 4 depending on the functionality required. Moreover, suffix arrays, together with their associated longest common prefix (LCP) information, can be computed in $O(n \log n)$ worst-case time using somewhat more than $3n$ words of storage and with some significant increase in storage requirements in $O(n)$ expected time [15]. Despite the reduced storage requirement, suffix arrays can be used as efficiently as suffix trees and DAWGs for many, but not all, applications.

As noted above, we deal here with a data structure that is closely related to a suffix array. The first component of this structure is an array $\boldsymbol{\pi}_{\boldsymbol{x}} = \boldsymbol{\pi}[1..n]$ of integers in the range $0..n-1$. We call $\boldsymbol{\pi}_{\boldsymbol{x}}$ the ***prefix array*** because $\boldsymbol{\pi}[i]$ is the length of the longest prefix of $\boldsymbol{x}[i..n]$ that is also a prefix of some $\boldsymbol{x}[j..n]$, $1 \leq j < i$ (zero if the longest such prefix is empty). Thus over all $i \in 1..n$, $j \in i{+}\boldsymbol{\pi}[i]..n$, the strings $\boldsymbol{x}[i..j]$ are exactly the distinct substrings of $\boldsymbol{x}$, while for $\boldsymbol{\pi}[i] > 0$ the nonempty strings $\boldsymbol{x}[i..i{+}\pi[i]{-}1]$ identify occurrences of longest repeating substrings in $\boldsymbol{x}$. The second component is another integer array $\boldsymbol{\lambda}_{\boldsymbol{x}} = \boldsymbol{\lambda}[1..n]$ (called the ***location array***) that for each location $i \in 2..n$ in $\boldsymbol{x}$ identifies another location $\boldsymbol{\lambda}[i] < i$ in $\boldsymbol{x}$ that shares the same longest prefix $\boldsymbol{\pi}[i]$ ($\boldsymbol{\lambda}[i] = 0$ if no such location exists). As we shall see, these arrays together provide essentially the same information as suffix arrays (and suffix trees), but in a form that does not depend on lexicographical order. Collectively, we refer to these two arrays as a ***quasi suffix array*** or ***QSA***.

In this paper we first introduce a very simple algorithm (Algorithm DIST1) that computes QSAs quickly in practice using only a single integer array of length n as additional storage, though its theoretical complexity is at least $O(n^2)$. We then go on to describe DIST2, an extension of DIST1 that requires one more array than DIST1 does, but that executes in $O(n \log n)$ expected time. We also discuss a variant DIST3 that may reduce expected time at a cost of further extra storage. We briefly present empirical test results conducted by one of us (Xiao) comparing DIST algorithms with well-known algorithms for constructing suffix arrays or suffix trees, and some empirical results due to Holub suggesting that the worst-case complexity of DIST3 should be the same as the average-case complexity.

Minor modifications to the DIST algorithms permit repetitions (adjacent repeating substrings) or all repeating substrings to be output as a byproduct of their execution. In this context, DIST has found application as a component of a strategy for data compression [20].

The QSA construction algorithms introduced in this paper are competitors for many applications to the suffix array construction algorithm (Algorithm MM) of Manber & Myers [15]. The storage required for the QSA itself (including what is effectively LCP information) is the same as that required for the suffix array. Although the currently-provable average-case time bound for our QSA algorithm DIST is greater than that of the most sophisticated version of Algorithm MM, DIST is a very much simpler algorithm and so in practice runs more quickly: Manber & Myers comment that their suffix array construction algorithm requires more time, by a factor of 3 to 10, than a corresponding suffix tree construction algorithm — DIST on the other hand executes faster than Algorithm MM, with space requirements that are comparable. According to our tests, however, the recent suffix array algorithm due to Sadakane [18] outperforms the fastest DIST algorithms in the average case.

In Section 2 we define various forms of suffix array and investigate their basic properties. In Section 3 we describe the algorithm DIST1 and its outputs, then go on in Sections 4 and 5 to discuss DIST2 and DIST3, respectively. Finally Section 6 describes the tests we performed and Section 7 presents concluding remarks.

2 Quasi Suffix Arrays

In this section we investigate the properties of QSAs, especially as they relate to other suffix structures.

We denote by $\mathrm{LCP}(u,v)$ the ***longest common prefix*** of two strings $\boldsymbol{u}$ and $\boldsymbol{v}$, with length $\mathrm{lcp}(u,v) = |\mathrm{LCP}(u,v)|$. When $\boldsymbol{u} = \boldsymbol{x}[i..n]$ and $\boldsymbol{v} = \boldsymbol{x}[j..n]$ are suffixes of a designated string $\boldsymbol{x}$, we abbreviate this notation to $\mathrm{LCP}(i,j)$ and $\mathrm{lcp}(i,j)$, respectively. Note that, according to the definition of QSA given in the introduction,

$$\boldsymbol{\pi}[i] = \mathrm{lcp}\big(i, \boldsymbol{\lambda}[i]\big) \tag{1}$$

whenever $\boldsymbol{\lambda}[i] > 0$, with the requirement that $\boldsymbol{\lambda}[i] < i$. Note also that $\boldsymbol{\lambda}[i] = 0$ if and only if $\boldsymbol{x}[i]$ is the leftmost occurrence in $\boldsymbol{x}$ of some letter.

In [15] the suffix array (SA) of $\boldsymbol{x} = \boldsymbol{x}[1..n]$ was introduced as a location array $\boldsymbol{\lambda}'_{\boldsymbol{x}} = \boldsymbol{\lambda}'[1..n]$ together with a prefix array $\boldsymbol{\pi}'_{\boldsymbol{x}} = \boldsymbol{\pi}'[1..n]$, where

- $\boldsymbol{\lambda}'[i] = j$ if and only if $\boldsymbol{x}[j..n]$ is the i^{th} largest suffix of $\boldsymbol{x}$ in lexicographical order (by convention a unique sentinel letter \$, lexicographically greater than any other letter, is placed in position $n{+}1$ of $\boldsymbol{x}$);

- $\boldsymbol{\pi}'[1] = 0$, while for $i > 1$,

$$\boldsymbol{\pi}'[i] = \text{lcp}\big(\boldsymbol{\lambda}'(i{-}1), \boldsymbol{\lambda}'(i)\big). \tag{2}$$

The order imposed by this definition on the elements of $\boldsymbol{\lambda}'$ is equivalent to listing the terminal nodes of the corresponding suffix tree visited during a preorder traversal. Thus, in the location array, suffixes form adjacency groupings that reflect the groupings of terminal nodes under the internal (lcp) nodes in the suffix tree. If lexicographical order is not required, a ***relaxed suffix array*** (***RSA***) can be employed just as well, in which only these groupings need to be respected. Thus, for example, $\boldsymbol{x} = abaab\$$ would give rise to SA

$$\boldsymbol{\lambda}' = 31425, \quad \boldsymbol{\pi}' = 01201,$$

while possible RSAs would include in addition

$$\begin{aligned} \boldsymbol{\lambda}' &= 52413, & \boldsymbol{\pi}' &= 01021, \\ \boldsymbol{\lambda}' &= 34152, & \boldsymbol{\pi}' &= 01201, \end{aligned}$$

but not

$$\boldsymbol{\lambda}' = 54213, \quad \boldsymbol{\pi}' = 00001.$$

The corresponding QSA for this example would be

$$\boldsymbol{\lambda} = 00112, \quad \boldsymbol{\pi} = 00121,$$

telling us, in view of (1), that

$$\boldsymbol{\pi}[3] = \text{lcp}(3,1) = 1,\ \boldsymbol{\pi}[4] = \text{lcp}(4,1) = 2,\ \boldsymbol{\pi}[5] = \text{lcp}(5,2) = 1.$$

Both RSA and QSA are constrained to reference each terminal node (at least once) and each lcp node (at least once) of the corresponding suffix tree of $\boldsymbol{x}$.

We remark that the QSA can also be generalized (to a GQSA) by relaxing the requirement that $\boldsymbol{\lambda}[i] < i$. Then by a straightforward algorithm, every RSA can be transformed into an equivalent GQSA:

$$
\begin{array}{l}
\boldsymbol{\lambda}\big[\boldsymbol{\lambda}'[n]\big] \leftarrow 0;\ \boldsymbol{\pi}\big[\boldsymbol{\lambda}'[n]\big] \leftarrow 0 \\
\textbf{for } i \leftarrow 2 \textbf{ to } n \textbf{ do} \\
\quad \boldsymbol{\lambda}\big[\boldsymbol{\lambda}'[i-1]\big] \leftarrow \boldsymbol{\lambda}'[i] \\
\quad \boldsymbol{\pi}\big[\boldsymbol{\lambda}'[i-1]\big] \leftarrow \boldsymbol{\pi}'[i]
\end{array}
$$

Thus the GQSA family (of which QSA is a member) is at least as numerous as the RSA family of suffix structures. We intend to investigate SA families and their properties further in future work.

3 Algorithm DIST1

As noted in the introduction, Algorithm DIST performs refinements of classes of equal substrings just as Crochemore's repetitions algorithm [9] does. The first step of Crochemore's algorithm and of DIST is the same: the equal substrings of length 1 are identified and their locations in $\boldsymbol{x}$ are placed in the same class. In the very common case that the alphabet is indexed, this calculation requires $\Theta(n)$ time; in the very rare case that the alphabet is not totally ordered, the time requirement is $O(n^2)$. We may in any case suppose that after execution of the first step the alphabet is indexed, hence arranged in some (not necessarily lexicographic) order.

But Crochemore's algorithm and DIST differ in the form in which the data is held. For example, if the Fibonacci string

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

were given, Crochemore's algorithm would compute classes (implemented as doubly-linked lists)

$$
p = 1 \qquad \underset{a}{<1,3,4,6,8,9,11,12>} \quad \underset{b}{<2,5,7,10,13>},
$$

while DIST computes an equivalent array

$$
\begin{array}{ccccccccccccccc}
 & & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} \\
\boldsymbol{c}_{(1)} & = & 0 & 0 & 1 & 3 & 2 & 4 & 5 & 6 & 8 & 7 & 9 & 11 & 10
\end{array}
$$

in which $\boldsymbol{c}_{(1)}[i]$ is the greatest $j < i$ such that $\boldsymbol{x}[j] = \boldsymbol{x}[i]$ (zero if no such j exists). Observe that the classes identified by $\boldsymbol{c}_{(1)}$ are singly-linked: they can be processed right-to-left but not left-to-right.

At subsequent stages, Crochemore's algorithm refines the classes for substrings of length p into those for $p{+}1$; in the above example the classes at level 2 are

$$\begin{array}{ccccc} p=2 & <1,4,6,9,12> & <3,8,11> & <2,5,7,10> & <13>\ . \\ & ab & aa & ba & b\$ \end{array}$$

On the other hand, DIST in its simplest form computes a new array $\boldsymbol{c}_{(\boldsymbol{p}+\mathbf{1})}$ by following a descending chain of locations $\boldsymbol{c}_{(\boldsymbol{p})}[i]$ from i until j is found for which $\boldsymbol{x}[j{+}p] = \boldsymbol{x}[i{+}p]$ (zero if no such j exists). In our example, the array

$$\begin{array}{r@{\,}ccccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} \\ \boldsymbol{c}_{(\mathbf{2})} = & 0 & 0 & 0 & 1 & 2 & 4 & 5 & 3 & 6 & 7 & 8 & 9 & 0 \end{array}$$

represents the classes at level $p = 2$. In fact, leaving columns blank below the first occurrence of zero, we may represent the entire calculation for the example string as follows:

$$\begin{array}{r@{\,}cccccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} & {\scriptstyle 14} \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a & a & b & a & a & b & \$ \\ \boldsymbol{c}_{(\mathbf{1})} = & 0 & 0 & 1 & 3 & 2 & 4 & 5 & 6 & 8 & 7 & 9 & 11 & 10 & \\ \boldsymbol{c}_{(\mathbf{2})} = & & & 0 & 1 & 2 & 4 & 5 & 3 & 6 & 7 & 8 & 9 & 0 & \\ \boldsymbol{c}_{(\mathbf{3})} = & & & & 1 & 0 & 4 & 2 & 3 & 6 & 7 & 8 & 0 & & \\ \boldsymbol{c}_{(\mathbf{4})} = & & & & 0 & & 1 & 2 & 3 & 6 & 7 & 0 & & & \\ \boldsymbol{c}_{(\mathbf{5})} = & & & & & & 1 & 2 & 0 & 6 & 0 & & & & \\ \boldsymbol{c}_{(\mathbf{6})} = & & & & & & 1 & 0 & & 0 & & & & & \\ \boldsymbol{c}_{(\mathbf{7})} = & & & & & & 0 & & & & & & & & \end{array} \tag{4}$$

Observe that the level p at which the first zero occurs in column i is the length of the shortest string $\boldsymbol{x}[i..i{+}p{-}1]$ at location i of $\boldsymbol{x}$ that is equal to *no* string $\boldsymbol{x}[j..j{+}p{-}1]$, $1 \le j < i$. In other words, we can write down the prefix array for $\boldsymbol{x}$ simply by recording $p{-}1$ for each i; that is, one less than the level at which the first zero appears. Thus, for our example,

$$\begin{array}{r@{\,}ccccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} \\ \boldsymbol{\pi}_{\boldsymbol{x}} = & 0 & 0 & 1 & 3 & 2 & 6 & 5 & 4 & 5 & 4 & 3 & 2 & 1 \end{array}$$

Furthermore, if we record the value $\boldsymbol{c}_{(\boldsymbol{p}-\mathbf{1})}[i]$ as well — that is, the final location $j < i$ such that $\boldsymbol{x}[j..j{+}p{-}2] = \boldsymbol{x}[i..i{+}p{-}2]$ — we are thus able to

```
— Given a string x[1..n] containing α distinct letters,
  compute π_x and λ_x.

compute c = c_(1)[1..n+1]   — c[n+1] = 0 since x[n+1] = $
π_x ← 0^n;
λ_x ← 0^n; nzero ← n−α; p ← 1

while nzero > 0 do
 — Compute level p+1 from level p.
  cprev ← 0
  for i ← n−p+1 downto 1 do
    if c[i] > 0 then
      if cprev = 0 then
        j ← 0

      elsif c[i] < cprev then
          — Skip to search for a match with x[i+p].
        j ← c[i]
        while j > 0 and x[i+p] ≠ x[j+p] do
          j ← c[j]

      else
          — Skip to search for a match with x[i].
        j ← cprev−1
        while j > 0 and x[i] ≠ x[j] do
          j ← c[j+1]−1

      if j = 0 then
        π[i] ← p; λ[i] ← c[i]; nzero ← nzero−1
      cprev ← c[i]; c[i] ← j
    else
      cprev ← 0

  p ← p+1
```

Figure 1: Algorithm DIST1

identify the pair of strings whose lcp is given by $\boldsymbol{\pi}[i]$. Denoting this second array by $\boldsymbol{\lambda_x}$, we have for our example string:

$$\begin{array}{rccccccccccccc} & \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 & \scriptstyle 11 & \scriptstyle 12 & \scriptstyle 13 \\ \boldsymbol{\lambda_x} = & 0 & 0 & 1 & 1 & 2 & 1 & 2 & 3 & 6 & 7 & 8 & 9 & 10. \end{array}$$

Then, in accordance with (1), these DIST outputs tell us that $\mathrm{lcp}(13, 10) = 1$, while $\mathrm{lcp}(9, 6) = 5$, and so on.

We display in Figure 1 pseudocode for the simplest form of our algorithm, which we call DIST1. After computing the first level $\boldsymbol{c} = \boldsymbol{c_{(1)}}$ of the array, DIST1 computes the remaining levels in the same storage space by methodically visiting each location i from right to left until every value $\boldsymbol{c}[i]$ is zero.

Setting $\boldsymbol{c}_{(1)}[i] = \boldsymbol{c}[i]$, we may define

$$\boldsymbol{c}_{(k)}[i] = \boldsymbol{c}\big[\boldsymbol{c}_{(k-1)}[i]\big]$$

for every $k > 1$ such that $\boldsymbol{c}_{(k-1)}[i] \neq 0$. We call the sequence

$$\boldsymbol{c}_{(0)}[i] = i, \boldsymbol{c}_{(1)}[i], \boldsymbol{c}_{(2)}[i], \ldots, \boldsymbol{c}_{(k^*)}[i] = 0 \tag{5}$$

the ***chain*** corresponding to i, and we observe that each chain corresponds to a class in the Crochemore refinement process. For every i such that $\boldsymbol{c}[i] > 0$ at level p, it is necessary to locate in the chain an ***antecedent*** $j = \boldsymbol{c}_{(k)}[i]$ of i (if it exists) such that $\boldsymbol{x}[i..i{+}p] = \boldsymbol{x}[j..j{+}p]$. This requirement can be met by performing ***skips*** down each chain that operate in either of two ways:

- (for $\boldsymbol{c}[i] < \boldsymbol{c}[i{+}1]$) determine an antecedent j of i such that $\boldsymbol{x}[i{+}p] = \boldsymbol{x}[j{+}p]$;
- (for $\boldsymbol{c}[i{+}1] < \boldsymbol{c}[i]$) determine an antecedent $j{+}1$ of $i{+}1$ such that $\boldsymbol{x}[i] = \boldsymbol{x}[j]$.

The selection between these two forms of skip is a heuristic that seems on average to reduce number of skips, but it is just a minor issue (the algorithm would work with either one and with the same time complexity). Observe that in order to compute $\boldsymbol{c}$ *in situ*, we need to use an extra variable *cprev* that stores the value of $\boldsymbol{c}[i{+}1]$ in level p so that it can be used in these calculations.

The worst-case time complexity of DIST1 is at least order n^2, attained for example by the string $\boldsymbol{x} = a^n$. We deal with the reduction of worst-case processing time later, in more refined versions of DIST. But even for DIST1 it is instructive to consider average-case complexity, making use of the result [13] that the expected length of the maximum repeating substring in a string $\boldsymbol{x}[1..n]$ on an alphabet of size α is

$$p^* = 2\log_\alpha n{+}O(1). \tag{6}$$

It follows immediately from (6) that p^* is the expected number of levels that need to be computed by DIST1, hence that the expected total number of steps that need to be taken in the **for** loop over all values of p is $O(n \log n)$. Except for the internal skip (**while**) loops, only constant time is required for each instruction within the **for** loop; thus, in order to estimate the expected time of DIST1, we need to be able to bound the expected time requirement of the skip loops. This is something that we are currently unable to do; however, in the next section we describe modifications to DIST1 that bypass the skipping problem by ensuring that at each level p each location in $\boldsymbol{c}$ is visited a constant number of times.

Even though the following theorem applies to all algorithms of the DIST family (in fact to Crochemore's repetitions algorithm as well), we present it here to show that DIST1 indeed computes QSAs.

Theorem 1 *Every algorithm from the DIST family computes a QSA.*

Proof For each $i \in 1..n$, every DIST algorithm computes the first level $p+1 \geq 1$ for which $c_{(p+1)}[i] = 0$; it then sets $\boldsymbol{\pi}[i] \leftarrow p$ and $\boldsymbol{\lambda}[i] \leftarrow c_{(p)}[i]$. Thus $\boldsymbol{\pi}[i]$ determines the substring $\boldsymbol{x}\big[i..i+\boldsymbol{\pi}[i]\big]$ at i that is the shortest one at that position equal to no preceding substring occurring at any position $j < i$. This is exactly in accordance with the definition of the prefix array given in the introduction. Furthermore, for $j = \boldsymbol{\lambda}[i] < i$, it must be true that $\boldsymbol{\pi}[i] = \mathrm{lcp}\big(i, \boldsymbol{\lambda}[i]\big)$, in accordance with the definition of the location array. $\square$

4 Algorithm DIST2

Our first modification to DIST1 is to replace the *ad hoc* skip loops of Figure 1 by processing that deals with each chain (refinement class) in its entirety. DIST2 refines each chain at level p into one or more subchains at level $p+1$ according to the letters $\mu = \boldsymbol{x}[i+p]$ that are found in the $(p+1)^{\text{th}}$ location of the substrings that occur in the chain. The revised algorithm is outlined in Figure 2.

DIST2 makes use of the following additional storage:

- a bit vector PROC$[1..n]$ in which PROC$[i]$ is `TRUE` if and only if $\boldsymbol{c}[i]$ has already been recomputed in the current level;
- a stack S of location values that allows each chain in level $p+1$ to be processed in left-to-right order;
- an array $\boldsymbol{L}[1..\alpha]$ in which $\boldsymbol{L}[\mu]$ gives the current rightmost location i in the current chain corresponding to the letter $\mu = \boldsymbol{x}[i+p]$ that is being traversed from left to right (zero if no such location exists);

— *Given a string $\boldsymbol{x}[1..n]$ containing α distinct letters, compute $\boldsymbol{\pi_x}$ and $\boldsymbol{\lambda_x}$ in expected $O(n \log n)$ time on an ordered or indexed alphabet.*

compute $\boldsymbol{c} = \boldsymbol{c_1}[1..n+1]$ — *a byproduct is an indexed alphabet* $1..\alpha$
$\boldsymbol{\pi} \leftarrow 0^n$; $\boldsymbol{\lambda} \leftarrow 0^n$; $nzero \leftarrow n-\alpha$; $p \leftarrow 1$
$\boldsymbol{L} \leftarrow 0^{\alpha+1}$ — $\boldsymbol{L}[\alpha+1] = 0$ *corresponds to* $\boldsymbol{x}[n+1] = \$$

```
while nzero > 0 do
 — Compute level p+1 from level p.
  PROC ← (FALSE)^n
  for i ← n−p+1 downto 1 do
    if not PROC[i] then
       — Process the chain for j = i.
      j ← i
       — (1) Put the locations j of the chain on stack S.
      repeat
        push(S, j); PROC[j] ← TRUE; j ← c[j]
      until j = 0
       — (2) Pop S to form subchains according to μ = x[j+p].
      repeat
        pop(S, j); μ ← x[j+p]
        if L[μ] = 0 then
          push(S′, μ)
          if c[j] > 0 then
            π[j] ← p; λ[j] ← c[j]; nzero ← nzero−1
        c[j] ← L[μ]; L[μ] ← j
      until empty(S)
       — (3) After processing the chain, reset location pointers.
      repeat
        pop(S′, μ); L[μ] ← 0
      until empty(S′)
   — Process next level if need be by outermost while
  p ← p+1
```

Figure 2: Algorithm DIST2

- a stack S' of the distinct letters $\mu = \boldsymbol{x}[i+p]$ that have occurred in the current chain.

Observe that the maximum possible number of integer entries in stack S is $n-\alpha+1$ — that is, the maximum number of occurrences of the most frequent letter in $\boldsymbol{x}$. Observe further that an entry can be pushed onto stack S' only after an entry has already been deleted from S. Thus the storage required for S and S' can be shared so that no more than $n-\alpha+1$ memory locations are required. The storage required for the array $\boldsymbol{L}$ is α memory locations. Thus altogether for S, S' and $\boldsymbol{L}$ at most $n+1$ memory locations are required, each sufficient to store an integer of magnitude at most n. Assuming that a memory location is a computer word, we can therefore state:

Theorem 2 *Given a string $\boldsymbol{x}[1..n]$ on an ordered or indexed alphabet of size α, Algorithm DIST2 computes the quasi suffix array of $\boldsymbol{x}$ in $O(n \log_\alpha n)$ time in the average case, using at most $2n+O(1)$ words and n bits of additional storage.*

Proof Storage requirements have been dealt with above. To establish the expected time bound, observe that the time required for the processing of each chain is proportional to the number of entries in it: each location in $\boldsymbol{c}$ is visited once when a stack S is formed, once when it is emptied, and at most once by the action of the **for** loop — at most three times altogether. Thus the processing required for each value of p is $\Theta(n)$, and since by (6) the expected number of levels is $O(\log_\alpha n)$, the result follows. $\square$

In fact, at a small cost in execution time (increasing from three to five the maximum number of times each position in $\boldsymbol{c}$ is visited), the n bits of storage required for the PROC array can be saved, reducing the space required to at most $2n+O(1)$ words. The revised algorithm, called DIST2$'$, replaces S, S' and $\boldsymbol{L}[1..\alpha]$ by a new auxiliary integer array $\boldsymbol{I}[1..n]$ and makes temporary use of the output arrays $\boldsymbol{\pi}$ and $\boldsymbol{\lambda}$ as a means of suppressing skips. The arrays $\boldsymbol{\pi}$ and $\boldsymbol{\lambda}$ are initialized to special null values.

When starting to process level p, DIST2$'$ initializes $\boldsymbol{I}$ to zeros. The arrays $\boldsymbol{\pi}$ and $\boldsymbol{\lambda}$ still contain `NULL` in all positions that belong to classes that contain more than one element; that is, all still unresolved positions. When DIST2$'$ processes a position i_k that is the largest position in a class $\langle i_1, i_2, \ldots, i_k \rangle$, the assignments $\boldsymbol{\pi}[i] \leftarrow$ `NULL` and $\boldsymbol{\lambda}[i] \leftarrow$ `NULL` are made for all $i \in \langle i_1, i_2, \ldots, i_k \rangle$. DIST2$'$ uses these `NULL` spaces temporarily for its own processing.

Thus, DIST2$'$ traverses the class right-to-left (according to the values stored in $c_{(p-1)}[i]$ that represent "left" links), and in $\boldsymbol{\pi}[i]$ it stores the "right" link (so it can traverse the class left-to-right to return to the starting position i_k). When DIST2$'$ reaches the beginning of the class, the values in $\boldsymbol{I}$ are guaranteed to be all zeros. While traversing back to i_k, at each position i DIST2$'$ checks the letter $\boldsymbol{x}[i{+}p]$ and in $\boldsymbol{I}\big[\boldsymbol{x}[i{+}p]\big]$ it finds the last position that ended with the same letter. It sets $\boldsymbol{\lambda}[i] \leftarrow \boldsymbol{I}\big[\boldsymbol{x}[i{+}p]\big]$ and sets $\boldsymbol{I}\big[\boldsymbol{x}[i{+}p]\big] \leftarrow i$, and continues traversing. When it reaches i_k, it returns to the beginning of the class and while traversing back to i_k does some "house-keeping"; more precisely, at each position i it sets $\boldsymbol{I}\big[\boldsymbol{x}[i{+}1]\big] \leftarrow 0$ and before moving to the position $\boldsymbol{\pi}[i]$, it sets $\boldsymbol{\pi}[i] \leftarrow \texttt{NULL}$. Thus each position i of the class $\langle i_1, i_2, \ldots, i_k\rangle$ has been visited four times. It is visited for the fifth time when DIST2$'$ comes to process the position i in its basic right-to-left traversal. But it finds in $\boldsymbol{\lambda}[i]$ a "precomputed" value for $c_{(p)}[i]$. It uses it and sets $\boldsymbol{\lambda}[i] \leftarrow \texttt{NULL}$. (Note that DIST2$'$ recognizes that i is the last element of a class by the fact that $\boldsymbol{\lambda}[i] = \texttt{NULL}$.)

Thus DIST2$'$ keeps moving steadily right to left, one position at a time, and only when it encounters the end of a class does it traverse the class twice forth and back. Hence:

Theorem 3 *Given a string $\boldsymbol{x}[1..n]$ on an ordered or indexed alphabet of size α, Algorithm DIST2$'$ computes the quasi suffix array of $\boldsymbol{x}$ in $O(n \log_\alpha n)$ time in the average case, using at most $2n{+}O(1)$ words of additional storage.* $\Box$

5 Algorithm DIST3

In this algorithm we improve on the performance of DIST2 by introducing a strategy to hop zero entries in $\boldsymbol{c}$ and also ***triangles*** induced by ***runs*** of the form

$$\boldsymbol{c}_{(\boldsymbol{p})}[i..i{+}k] = q, q{+}1, \ldots, q{+}k \tag{7}$$

for some $q \geq 1$ and some maximum $k \geq 1$. Examples of such triangles can be found in (3), where

$$\boldsymbol{c}_{(\mathbf{1})}[6..8] = 4, 5, 6$$

and

$$\boldsymbol{c}_{(\mathbf{2})}[9..12] = 6, 7, 8, 9.$$

The following result, easily proved, shows how a triangle ranging over levels $p, p{+}1, \ldots, p{+}k$ can be inferred from a run of $k{+}1$ elements in level p:

Lemma 1 *Suppose that for some $i \in 1..n{-}1$, some $k \in 1..n{-}i$, and some $q \geq 1$,*

$$\boldsymbol{c_{(p)}}[i..i{+}k] = q, q{+}1, \ldots, q{+}k.$$

Then for every $j \in 1..k$,

$$\boldsymbol{c_{(p+j)}}[i..i{+}k{-}j] = q, q{+}1, \ldots, q{+}k{-}j. \quad \square$$

We see therefore that every occurrence of a run of length $k{+}1$ allows us to predict (and therefore potentially to avoid processing) a total of $\binom{k+1}{2}$ locations in $\boldsymbol{c_{(p+j)}}$, $j = 1, 2, \ldots, k$. Since runs are a symptom of the frequent occurrence of repeating substrings in $\boldsymbol{x}$, and since it is repeating substrings of length p that ensure the existence of nonzero elements at level p, the idea of avoiding the processing of triangles is an attractive one. Accordingly we introduce processing and data structures that allow us to (a) recognize runs of length $k{+}1$ as they occur in level p, (b) hop over the processing of the portions of the corresponding triangle that occur in levels $p{+}1, p{+}2, \ldots, p{+}k$.

In addition to the arrays $\boldsymbol{c}[1..n]$ and $\boldsymbol{I}[1..n]$ employed in DIST2′, DIST3 also makes use of a circular queue of pairs *Thop* (for triangle-hop) containing at most $\lceil \frac{n}{2} \rceil$ integer pairs (i, j). Each entry i in *Thop* designates the index in $\boldsymbol{c}$ to hop from, while j is the index to hop to. We actually use the array $\boldsymbol{c}$ itself to control zero hopping, utilising the fact that since the value n can never occur in $\boldsymbol{c}$, it can therefore be used as another way of representing zero. If there is an isolated zero entry, we leave it alone; otherwise, if there is a block of zero entries such as

$$\boldsymbol{c}[j] = \boldsymbol{c}[j{+}1] = \cdots = \boldsymbol{c}[k] = 0, \quad k > j,$$

we represent it by

$$\boldsymbol{c}[j] = \boldsymbol{c}[j{+}1] = \cdots = \boldsymbol{c}[k{-}1] = n, \quad \boldsymbol{c}[k] = j{-}1.$$

Thus the values 0, n, and "$j-1$ preceded by n" are all treated as zeros. Moreover, "$j-1$ preceded by n" indicates a hop to position $j-1$; that is, a hop over a block of zero entries.

During the processing we keep track of ***Lend***, the position of the leftmost nonzero entry in $\boldsymbol{c}[1..n]$, and ***Rend***, the position of the rightmost nonzero entry. We maintain zero-hops from each nonzero position followed by a zero to the next nonzero position — for example, for $\boldsymbol{c}[2..6] = 1\ 0\ 0\ 0\ 3$, we will hop from $\boldsymbol{c}[6] = 3$ to $\boldsymbol{c}[2] = 1$. For triangles we maintain hops from the rightmost position in the triangle to the first position to the left that is not a part of the triangle — for example, for $\boldsymbol{c}[4..8] = 6\ 10\ 11\ 12\ 15$, we will hop from $\boldsymbol{c}[7] = 12$ to $\boldsymbol{c}[4] = 6$. It is possible that the end of a hop may be to the left of the start of another hop; for example, when a triangle-hop is followed immediately by a zero-hop, as with

$$\boldsymbol{c}[3..11] = 1\ 0\ 0\ 0\ 3\ 4\ 5\ 6\ 10,$$

yielding a triangle-hop from $\boldsymbol{c}[10] = 6$ to $\boldsymbol{c}[6] = 0$ and a zero-hop from $\boldsymbol{c}[7] = 3$ to $\boldsymbol{c}[3] = 1$. It may also happen that the end of a hop (the value of *to*) is smaller than $\boldsymbol{Lend}$.

For simplicity, in the high-level pseudocode for DIST3 we use the following terminology:

- *delete hop* means remove the hop from the queue;
- *transfer hop* means remove the hop, decrement its *from* value by one, and if the new value of *from* is positive, then insert the updated hop at the end of the queue;
- *copy hop* means delete the hop from the queue and insert it at the end unchanged;
- *extend hop r* means to replace the *to* value of the hop by the smaller value r (extending the hop to the left);
- *create hop* means insert the appropriate pair (*from*,*to*) at the end of the queue.

A broad outline of the algorithm in high-level pseudocode is given in Figures 3 and 4. The body of the **while** loop consists of a rather unwieldy nested **if** statement. When we implemented the algorithm in C++, the single **if** was replaced by multiply-nested **if** statements that were more efficient but less conducive to presentation. This single **if** represents 29 rules for maintaining the zero-hop queue and the triangle-hop queue and for manipulating the index i that controls the hops from left to right.

The following lemma justifies the claim made earlier about the space required by the *Thop* queue:

Lemma 2 *For any input string $\boldsymbol{s}$ and any level p of processing by DIST3, the number of triangle-hops is at most $\lceil \frac{n}{2} \rceil$.*

— *as in DIST1, compute* $\boldsymbol{c}[1..n]$ *for level 1*
set zero-hop and triangle-hop queues for level 1
set Lend to the position of the first nonzero entry of $\mathbf{c}$ *from the left*
set Rend to the position of the first nonzero entry of $\mathbf{c}$ *from the right*

$p \leftarrow 1$
$i \leftarrow Rend$
while $i \geq Lend$ **do**
  $cprev \leftarrow \boldsymbol{c}[i]$ — *remember original value of* $\boldsymbol{c}[i]$
  $j \leftarrow i$ — *remember original value of* i
  calculate new value of $\boldsymbol{c}[i]$ as in DIST1 *(note that* $\boldsymbol{c}[i]$ *is necessarily nonzero)*
  if $i = Rend$ **and** $\boldsymbol{c}[i] = 0$ **and** there is a zero-hop (i, m) **then**
    delete the hop; $Rend \leftarrow i \leftarrow m$
  elsif $i = Rend$ **and** $\boldsymbol{c}[i] = 0$ **and** there is a triangle-hop (i, m) **then**
    transfer the hop; $Rend \leftarrow Rend{-}1$; $i \leftarrow m$
  elsif $i = Rend$ **and** $\boldsymbol{c}[i] = 0$ **then**
    $Rend \leftarrow Rend{-}1$; $i \leftarrow i{-}1$
  elsif $i = Lend$ **and** $\boldsymbol{c}[i] = 0$ **and** there is a zero-hop (m, i) **then**
    delete the hop; $Lend \leftarrow m$
  elsif $i = Lend$ **and** $\boldsymbol{c}[i] = 0$ **then**
    $Lend \leftarrow Lend{+}1$
  elsif $\boldsymbol{c}[i] = 0$ **and** there is a zero-hop (r, i) **and** a zero-hop (i, m) **then**
    extend the first zero-hop from r to m;
    delete the second zero-hop; $i \leftarrow m$
  elsif $\boldsymbol{c}[i] = 0$ **and** there is a zero-hop (r, i) **and** a triangle-hop (i, m) **then**
    transfer the triangle-hop; extend the zero-hop to $i{+}1$; $i \leftarrow m$
  elsif $\boldsymbol{c}[i] = 0$ **and** there is a zero-hop (r, i) **then**
    extend the hop to $i{+}1$; $i \leftarrow i{-}1$
  elsif $\boldsymbol{c}[i] = 0$ **and** there is a triangle-hop (i, m) **then**
    transfer the hop; create a new zero-hop $(i{+}1, i{-}1)$; $i \leftarrow m$
  elsif $\boldsymbol{c}[i] = 0$ **and** there is a zero-hop (i, m) **then**
    delete the hop; create a new zero-hop $(i{+}1, m)$; $i \leftarrow m$
  elsif $\boldsymbol{c}[i] = 0$ **then**
    create a new triangle-hop $(i{+}1, i{-}1)$; $i \leftarrow i{-}1$
  elsif $i = Rend$ **and** $\boldsymbol{c}[i] \neq 0$ **and** there is a zero-hop (i, m) **then**
    copy the hop; $i \leftarrow m$
  elsif $i = Rend$ **and** $\boldsymbol{c}[i] \neq 0$ **and** there is a triangle-hop (i, m) **and**
    $cprev = \boldsymbol{c}[i]$ **then**
    copy the hop; $i \leftarrow m$
  elsif $i = Rend$ **and** $\boldsymbol{c}[i] \neq 0$ **and** there is a triangle-hop (i, m) **and**
    $cprev \neq \boldsymbol{c}[i]$ **then**
    transfer the hop; $i \leftarrow m$

(continued on next page)

Figure 3: DIST3: the maintenance of zero-hop and triangle-hop queues

elsif $i = \mathit{Lend}$ **and** $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i]{+}1 = \boldsymbol{c}[i{+}1]$ **and**
  there is a triangle-hop (r, i) **then**
  extend the hop to $i{-}1$; $i \leftarrow i{-}1$
elsif $i = \mathit{Lend}$ **and** $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i]{+}1 = \boldsymbol{c}[i{+}1]$ **then**
  create a new triangle-hop $(i{+}1, i{-}1)$; $i \leftarrow i{-}1$
elsif $i = \mathit{Lend}$ **and** $\boldsymbol{c}[i] \neq 0$ **then**
  $i \leftarrow i{-}1$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i{-}1] + 1 = \boldsymbol{c}[i]$ **and** $\boldsymbol{c}[i] + 1 = \boldsymbol{c}[i{+}1]$ **and**
  there are triangle-hops (r, i) and (i, m) **then**
  delete hop (i, m); extend hop (r, i) to m; $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i] + 1 = \boldsymbol{c}[i{+}1]$ **and**
  there are triangle-hops (r, i) and (i, m) **then**
  transfer hop (i, m); extend hop (r, i) to m; $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i{-}1] + 1 = \boldsymbol{c}[i]$ **and**
  there are triangle-hops (r, i) and (i, m) **then**
  copy hop (i, m); $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** there are triangle-hops (r, i) and (i, m) **then**
  transfer hop (i, m); $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i{-}1] + 1 = \boldsymbol{c}[i]$ **and** $\boldsymbol{c}[i] + 1 = \boldsymbol{c}[i{+}1]$ **and**
  there is a triangle-hop (i, m) **then**
  delete the hop; create a new triangle-hop $(i{+}1, m)$; $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i{-}1] + 1 = \boldsymbol{c}[i]$ **and** there is a triangle-hop (i, m) **then**
  copy the hop; $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i] + 1 = \boldsymbol{c}[i{+}1]$ **and** there is a triangle-hop (i, m) **then**
  transfer the hop; create a new triangle-hop $(i{+}1, i{-}1)$; $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** there is a triangle-hop (i, m) **then**
  transfer the hop; $i \leftarrow m$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i] + 1 = \boldsymbol{c}[i{+}1]$ **and** there is a triangle-hop (r, i) **then**
  extend the hop to $i{-}1$; $i \leftarrow i{-}1$
elsif $\boldsymbol{c}[i] \neq 0$ **and** there is a triangle-hop (r, i) **then**
  $i \leftarrow i{-}1$
elsif $\boldsymbol{c}[i] \neq 0$ **and** $\boldsymbol{c}[i] + 1 = \boldsymbol{c}[i{+}1]$ **then**
  create a new triangle-hop $(i{+}1, i{-}1)$; $i \leftarrow i{-}1$
elsif $\boldsymbol{c}[i] \neq 0$ **then**
  $i \leftarrow i{-}1$

if $\mathit{cprev} > 0$ **and** $\boldsymbol{c}[j] = 0$ **then**
  $\boldsymbol{\pi}[j] \leftarrow p$; $\boldsymbol{\lambda}[j] \leftarrow \boldsymbol{c}[j]$
$p \leftarrow p{+}1$ — *back to start of* **while** *loop on previous page*

Figure 4: DIST3 (continued from previous page)

Proof Let $f(r)$ be the number of hops that start at positions $i \le r$. We will show by induction over r that $f(r) \le \lceil \frac{r}{2} \rceil$.

For $r = 2$, there is at most one triangle-hop. Suppose the lemma holds for all positive integers less than some r; then for $r{+}1$ there are three possibilities:

(a) There is no hop from $r{+}1$ to any $m \le r$. Then $f(r{+}1) = f(r) \le \lceil \frac{r}{2} \rceil \le \lceil \frac{r+1}{2} \rceil$.

(b) There is a hop from $r{+}1$, but there is no hop from r. Then $f(r{+}1) \le f(r){+}1 = f(r{-}1){+}1 \le \lceil \frac{r-1}{2} \rceil {+}1 = \lceil \frac{r+1}{2} \rceil$.

(c) There is a hop from both r and $r{+}1$. Inspection of DIST3 shows that the algorithm maintains the hops in a strictly decreasing manner when a hop's *to* value is strictly bigger than the next hop's *from* value. Therefore, this case cannot occur.

$\square$

Note that each "rule" in the body of the **while** loop in DIST3 is executed in constant time and that only one rule is used in a single pass. Thus the time complexity of DIST3 is that of DIST2$'$ less all the processing of the hopped entries. An estimate of the number of steps saved by not processing the hopped entries has so far resisted our attempts at analysis, and so our formal time complexity result for DIST3 is the same as for DIST2$'$:

Theorem 4 *Given a string $\boldsymbol{x}[1..n]$ on an ordered or indexed alphabet of size α, Algorithm DIST3 computes the quasi suffix array of $\boldsymbol{x}$ in $O(n \log_\alpha n)$ time in the average case, using at most $3n{+}O(1)$ words of additional storage.* $\square$

However, as discussed in the next section, there is experimental evidence that leads us to believe that DIST2, DIST2$'$ and DIST3 all execute, at least in the average case, faster than the formal results indicate.

6 Testing Performed

We discuss here experiments that we have carried out in an effort to estimate

(1) sharper upper bounds (if such exist) on the asymptotic time requirements, both worst- and average-case, of the DIST algorithms (DIST2, DIST2$'$ and DIST3);

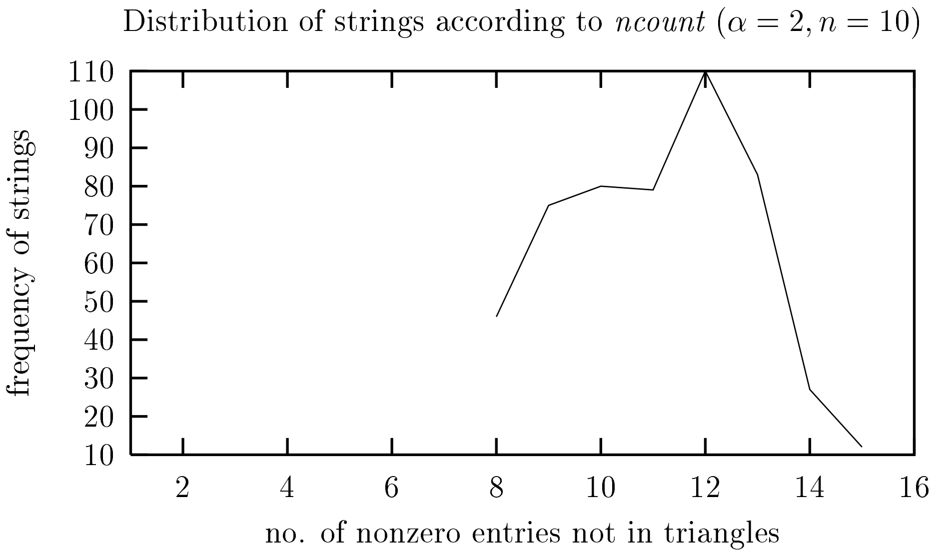

Figure 5: DIST3-complexity distribution of binary strings of length 10

(2) the speed in practice of the various DIST algorithms when applied to long strings on alphabets of various sizes;

(3) the speed in practice of the various DIST algorithms in comparison to the fastest suffix tree and suffix array construction algorithms.

In order to say something about (1), we have first studied occurrences of triangles in the array $\boldsymbol{c}$. For small alphabets and short strings, it is possible to generate all strings, and for each generated string to compute the number of nonzero entries in $\boldsymbol{c}$ that are not a part of a triangle sequence. This number is a rather good approximation of the number of steps DIST3 must perform on the string (up to multiplication by a constant). We were able to determine the complete distributions of strings on a binary alphabet for strings of lengths up to 26, and on a ternary alphabet up to 19. Selected graphs are presented in Figures 5–9. The graphs strongly suggest a normal distribution of the number of nonzero non-triangle entries, with a maximum value less than twice the average. On this basis, we state

Conjecture 1 *The worst-case time complexity of Algorithm DIST3 is of the same order as the average-case time complexity.*

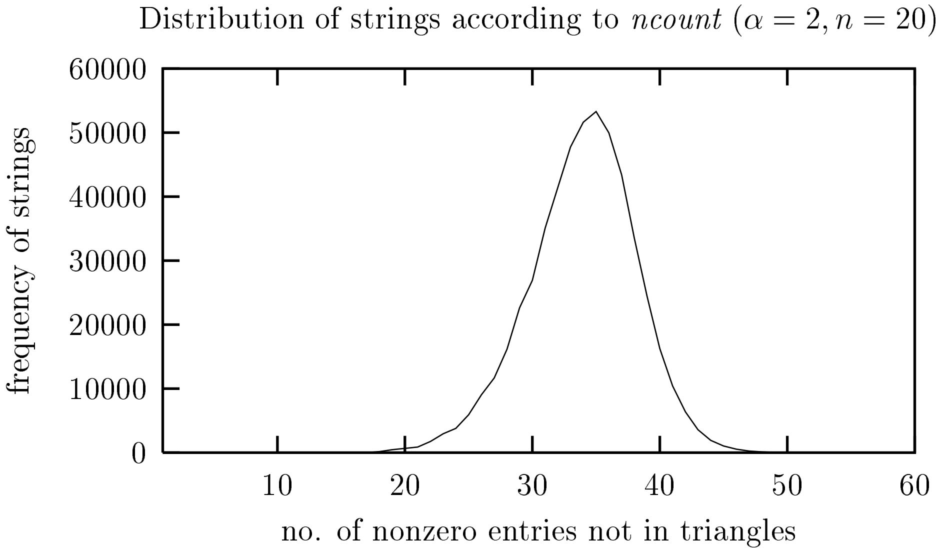

Figure 6: DIST3-complexity distribution of binary strings of length 20

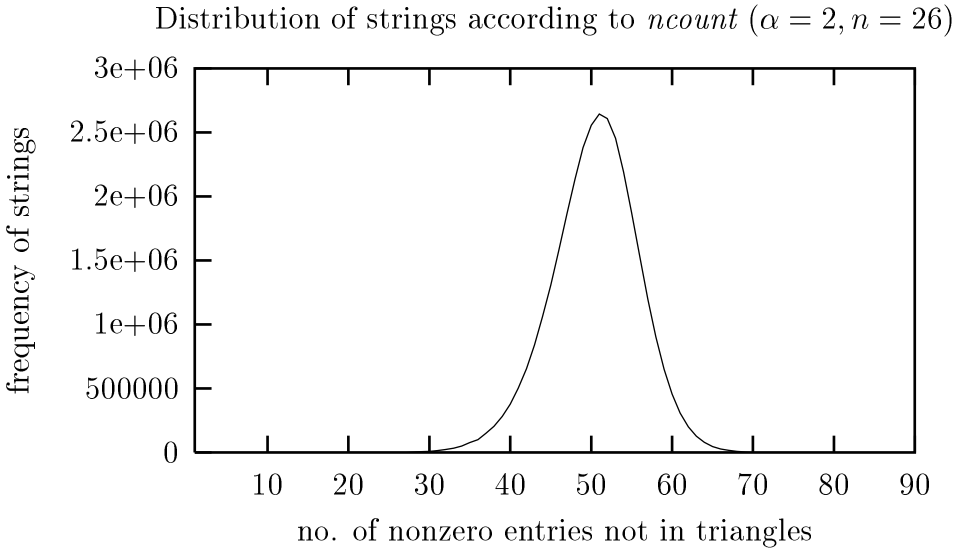

Figure 7: DIST3-complexity distribution of binary strings of length 26

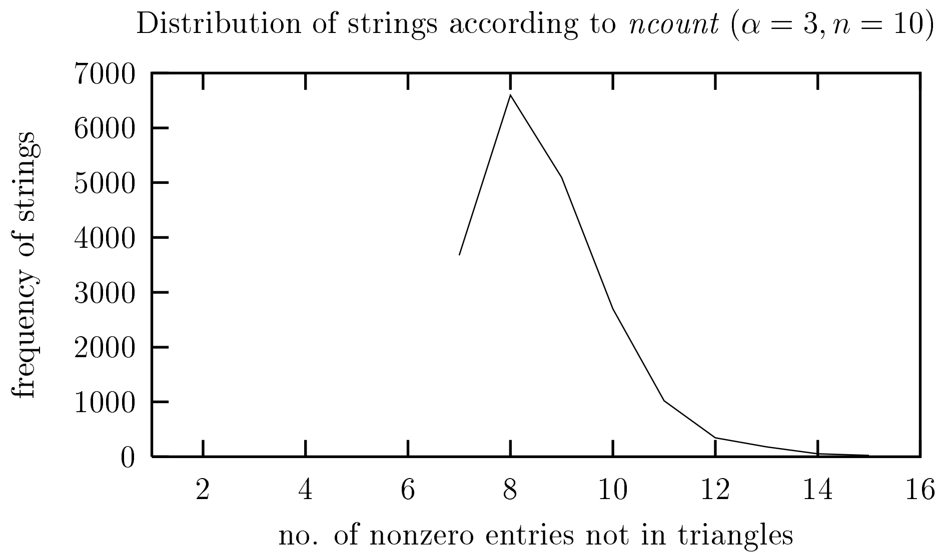

Figure 8: DIST3-complexity distribution of ternary strings of length 10

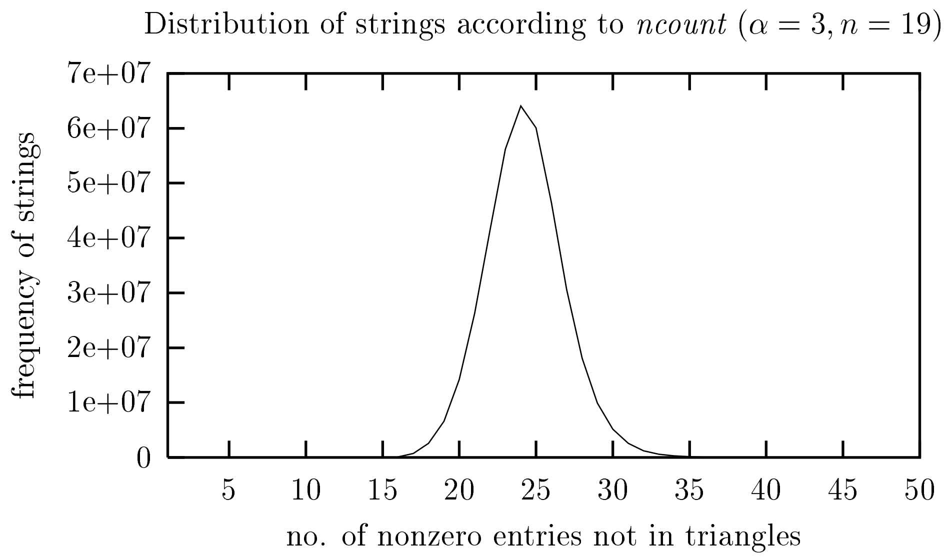

Figure 9: DIST3-complexity distribution of ternary strings of length 19

In order to address (2) and (3), extensive tests have been conducted [23] on pseudorandom strings and on strings drawn from the Calgary corpus [1], with lengths ranging up to two megabytes. The tests have involved DIST1, DIST2, and a simpler version of DIST3 (called DIST3′) that performs zero-hops but not triangle-hops. The following tentative conclusions are supported by the evidence:

- For strings on small alphabets ($\alpha = 2, 4$), DIST1 is generally fastest, while for strings on large alphabets ($\alpha = 26, 85, 128$), DIST3′ is generally fastest.
- For strings on large alphabets, both DIST2 and DIST3′ appear to have time requirement roughly linear in string length; there is some evidence that for large alphabets and much longer strings (100+ megabytes), DIST2 would be the algorithm of choice.

Tests were also performed on two versions (M1 and M2) of McCreight's suffix tree construction algorithm as recently implemented very efficiently by Kurtz [14], as well as on Sadakane's improved suffix tree algorithm (Algorithm S) [18]. The main conclusion was as follows:

- DIST3′ has some advantage over M1 and M2 for strings on large alphabets, but is consistently somewhat slower than Algorithm S.

Future testing is planned to include experiments on DIST2′ and DIST3, also making use of much longer input strings. Meanwhile, we make the following conjecture:

Conjecture 2 *Applied to strings on large alphabets, the average-case time complexity of* DIST2, DIST2′, DIST3 *and* DIST3′ *is* $\Theta(n)$.

Observe that if both the above conjectures hold, DIST3 has *worst-case* time complexity $\Theta(n)$.

7 Concluding Remarks

In this paper we have introduced the idea of a quasi suffix array, then described variants of an algorithm DIST to construct it, three of which (DIST2, DIST2′ & DIST3) provably execute in $O(n \log n)$ average-case time. We have conjectured that DIST3 executes in the same class in the worst case, possibly in $\Theta(n)$. We showed that a QSA provides essentially the same information that suffix arrays do, and thus may be used for applications where lexicographic order is not important. We see as a main advantage of QSAs the fact that they should be computable by simple, fast, and space-efficient algorithms. We believe that a profitable line of research will be to develop QSA construction algorithms that take better advantage of their partially unordered nature than we have so far been able to do. Toward this end, we believe that, as suggested in Section 2, a closer analysis of suffix array structures will be useful.

References

[1] Calgary corpus. `http://links.uwaterloo.ca/calgary.corpus.html`.

[2] A. Apostolico. The myriad virtues of suffix trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85–96. Springer-Verlag, Berlin, 1985.

[3] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993.

[4] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proc. IEEE*, 88(11):1733–1744, 2000.

[5] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theor. Comput. Sci.*, 22(3):297–315, 1983.

[6] Miroslav Balík. DAWG automaton versus suffix array. In *Proceedings of the 7th Conference on Implementation and Application of Automata*, pages 239–244, Tours, France, 2002.

[7] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in Lecture Notes in Computer Science, pages 397–411, Montréal, Canada, 2000. Springer-Verlag, Berlin.

[8] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms*

on Words, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 97–107. Springer-Verlag, Berlin, 1985.

[9] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[10] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In A. Apostolico and J. Hein, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, number 1264 in Lecture Notes in Computer Science, pages 116–129, Aarhus, Denmark, 1997. Springer-Verlag, Berlin.

[11] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, FL, 1997.

[12] J. Holub and M. Crochemore. On the implementation of Compact DAWG's. In *Proceedings of the 7th Conference on Implementation and Application of Automata*, pages 293–298, Tours, France, 2002.

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

[14] S. Kurtz. Reducing the space requirements of suffix trees. *Softw. Pract. Exp.*, 29(13):1149–1171, 1999.

[15] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[16] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. Algorithms*, 23(2):262–272, 1976.

[17] D. R. Morrison. PATRICIA — practical algorithm to retrieve coded in alphanumeric. *J. Assoc. Comput. Mach.*, 15(4):514–534, 1968.

[18] K. Sadakane. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. In J. A. Storer and M. Cohn, editors, *Proceedings Data Compression Conference*, pages 129–138, Snowbird, UT, 1998. IEEE Computer Society Press.

[19] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 140–152, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

[20] A. Turpin and W. F. Smyth. An approach to phrase selection for offline data compression. In M. Oudshoorn, editor, *Proc. 25th Australasian Computer Science Conference*, pages 267–273, 2002.

[21] E. Ukkonen. Constructing suffix trees on-line in linear time. In J. van Leeuwen, editor, *Proceedings of the 12th IFIP World Computer Congress*, pages 484–492, Madrid, Spain, 1992. North-Holland.

[22] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[23] X. Xiao. Computing quasi suffix arrays. Master's thesis, Department of Computing & Software, McMaster University, 2002.