Computing quasi suffix arrays*

Leila Baghdadi[1,2], František Franěk[1], W. F. Smyth[1,3], and Xiangdong Xiao[1]

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

[2] Department of Medical Biophysics, Medical Sciences Building,
University of Western Ontario, London, Ontario, Canada N6A 5C1

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

June 27, 2002

Abstract. We describe a simple space- and time-efficient algorithm that computes an integer array $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$ whose elements identify all the distinct substrings in a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Essentially, these elements give the lengths of longest common prefixes (LCPs) of suffixes of $\boldsymbol{x}$. A second integer array $\boldsymbol{\lambda} = \boldsymbol{\lambda}[1..n]$ optionally output by the algorithm identifies locations in $\boldsymbol{x}$ whose LCPs are given by $\boldsymbol{\pi}$, thus exactly the information provided by a suffix array or suffix tree. We therefore refer to these two arrays collectively as a ***quasi suffix array***. The distinction between quasi suffix arrays and standard suffix arrays is that the locations specified in $\boldsymbol{\lambda}$ do not necessarily give the suffixes of $\boldsymbol{x}$ in lexicographical order; indeed, our algorithm does not depend upon any specific ordering of the alphabet. Thus for problems that do not require lexicographical order — for example, pattern-matching, calculation of repeating substrings — quasi suffix arrays are sufficient. Compared to suffix tree construction, it appears that in practice our algorithm executes faster, while the standard suffix array construction algorithm is several times slower. We present four variants of the algorithm: a prototype, two that require $O(n \log n)$ expected time, and one that we conjecture executes in expected time $O(n)$.

1 Introduction

In string processing and pattern-matching problems on an ordered alphabet, a collection of closely-related standard data structures are

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

frequently employed to promote algorithmic efficiency: suffix trees [19], directed acyclic word graphs (DAWGs) [7], and suffix arrays [13], 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 [15], suffix trees were introduced by Weiner [19], who also described the first of several worst-case $O(n \log n)$-time algorithms [14, 18] 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 [10], but its space requirements are very large and the overhead associated with the large space so signficant, 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 (Algorithm C) for the computation of all repetitions in $\boldsymbol{x}$ [8]. Indeed Algorithm C 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 algorithm.

Suffix trees are important and useful in many pattern-matching contexts [1, 4, 16, 2, 5], as well as for data compression [3], 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 [12] as well as alternative structures, such as DAWGs [7, 9] and suffix arrays [13]. 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 [13]. 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 $\boldsymbol{\pi}[i]$ (zero if no such location exists). Together these arrays provide 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***.

In this paper we first introduce a very simple algorithm (Algorithm DIST1) that computes quasi suffix arrays quickly in practice using only a single integer array of length n as additional storage. We then go on to describe DIST2, an extension of DIST1 that requires one more integer array and one bit array as working storage, but that executes in $O(n \log n)$ expected time. We also discuss a variant DIST3 of DIST2 that may reduce expected time at a cost of further extra storage. Finally we describe Algorithm DIST4 that we conjecture requires only $O(n)$ time in the average case. 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 [17].

The quasi suffix array construction algorithms introduced in this paper are competitors for many applications to the suffix array construction algorithm (Algorithm MM) of Manber & Myers [13]. The storage required for the quasi suffix array 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 quasi suffix array 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 suffix tree construction. The space requirements for DIST and MM are comparable.

In Section 2 we describe Algorithm DIST1 and its outputs, then go on in Section 3 to discuss DIST2 and DIST3. Section 4 then introduces DIST4 and finally Section 5 discusses applications and some of the connections among quasi suffix arrays, suffix arrays and suffix trees.

2 Algorithm DIST1

As noted in the Introduction, Algorithm DIST performs refinements of classes of equal substrings just as Crochemore's repetitions algorithm [8] does. The first steps of Algorithms C & DIST are identical: 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 C and DIST differ in the form in which the data is held. For example, if the Fibonacci string

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

were given, Algorithm C would compute classes (implemented as doubly-linked lists)

$$\begin{array}{lll} p = 1 & <1,3,4,6,8,9,11,12> & <2,5,7,10,13>, \\ & a & b \end{array}$$

while DIST computes an equivalent array

$$\begin{array}{rl} & \begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \end{array} \\ \boldsymbol{c_1} = & \begin{array}{cccccccccccccc} 0 & 0 & 1 & 3 & 2 & 4 & 5 & 6 & 8 & 7 & 9 & 11 & 10 \end{array} \end{array}$$

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, Algorithm C refines the classes for substrings of length p into those for $p+1$; in the above example the classes at level 2 are

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

On the other hand, DIST in its simplest form computes a new array $\boldsymbol{c_{p+1}}$ by following a descending chain of locations $\boldsymbol{c_p}[i]$ 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

$$\begin{array}{rl} & \begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \end{array} \\ \boldsymbol{c_2} = & \begin{array}{cccccccccccccc} 0 & 0 & 0 & 1 & 2 & 4 & 5 & 3 & 6 & 7 & 8 & 9 & 0 \end{array} \end{array}$$

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:

$$\begin{array}{rl} & \begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \end{array} \\ \boldsymbol{x} = & \begin{array}{cccccccccccccc} a & b & a & a & b & a & b & a & a & b & a & a & b & \$ \end{array} \\ \boldsymbol{c_1} = & \begin{array}{cccccccccccccc} 0 & 0 & 1 & 3 & 2 & 4 & 5 & 6 & 8 & 7 & 9 & 11 & 10 \end{array} \\ \boldsymbol{c_2} = & \begin{array}{cccccccccccccc} & & 0 & 1 & 2 & 4 & 5 & 3 & 6 & 7 & 8 & 9 & 0 \end{array} \\ \boldsymbol{c_3} = & \begin{array}{cccccccccccccc} & & & 1 & 0 & 4 & 2 & 3 & 6 & 7 & 8 & 0 \end{array} \\ \boldsymbol{c_4} = & \begin{array}{cccccccccccccc} & & & 0 & & 1 & 2 & 3 & 6 & 7 & 0 \end{array} \\ \boldsymbol{c_5} = & \begin{array}{cccccccccccccc} & & & & & 1 & 2 & 0 & 6 & 0 \end{array} \\ \boldsymbol{c_6} = & \begin{array}{cccccccccccccc} & & & & & 1 & 0 & & 0 \end{array} \\ \boldsymbol{c_7} = & \begin{array}{cccccccccccccc} & & & & & 0 \end{array} \end{array} \qquad (1)$$

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,

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

Furthermore, if we record the value $\boldsymbol{c_{p-1}}[i]$ 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 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:

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

Let us denote by $\mathrm{lcp}(i, j)$ the length of the LCP of $\boldsymbol{x}[i..n]$ and $\boldsymbol{x}[j..n]$. Then the above output of DIST tells us that $\mathrm{lcp}(13, 10) = 1$, while $\mathrm{lcp}(9, 6) = 5$, and so on. In general,

$$\mathrm{lcp}\big(i, \boldsymbol{\lambda}[i]\big) = \boldsymbol{\pi}[i] \tag{2}$$

for every $i \in 1..n$, $\boldsymbol{\lambda}[i] > 0$. As discussed further in Section 5, the information in these two arrays is sufficient to specify the suffix tree of $\boldsymbol{x}$ — though, as discussed above, without the notion of lexicographic order.

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

$$\boldsymbol{c^{(0)}}[i] = i, \boldsymbol{c^{(1)}}[i], \boldsymbol{c^{(2)}}[i], \ldots, \boldsymbol{c^{(k^*)}}[i] = 0 \tag{3}$$

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:

```
 — 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

  p ← p+1
```

Fig. 1. Algorithm DIST1

– (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 just 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 [11] that the expected length of the maximum repeating substring in a string $\boldsymbol{x}[1..n]$ on an alphabet of size α is

$$p^* = 2\log_\alpha n + O(1). \tag{4}$$

It follows immediately from (4) 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 $\Theta(n \log n)$. 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 (at most three).

3 Algorithms DIST2 & DIST3

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 $L[1..\alpha]$ in which $L[\mu]$ 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);
- 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 L is α memory locations. Thus altogether for S, S' and 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 1. *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. Thus the processing required for each value of p is $\Theta(n)$, and since by (4) the expected number of levels is $O(\log_\alpha n)$, the result follows. □

An obvious strategy for reducing the time requirement of Algorithm DIST2 is to reduce the number of locations that need to be

```
— Given a string x[1..n] containing α distinct letters,
   compute π_x and λ_x in expected O(n log n) time
   on an ordered or indexed alphabet.

compute c = c_1[1..n]   — a byproduct is an indexed alphabet 1..α

π_x ← 0^n; λ_x ← 0^n; nzero ← n−α; p ← 1
PROC ← (FALSE)^n
L ← 0^{α+1}   — L[α+1] = 0 corresponds to x[n+1] = $

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

Fig. 2. Algorithm DIST2

visited in each level p+1. We can accomplish this by introducing additional data structures that allow locations i for which $\boldsymbol{c}[i] = 0$ to be "hopped". There are two basic approaches to do so. One amounts to maintaining what is essentially a doubly-linked list of non-zero entries and these are the only ones being processed, or maintaining a circular queue of hops, i.e. pairs of positions $(from, to)$ indicating to "hop" from the position $from$ to the position to. For DIST3 we briefly discuss the first approach, while for DIST4 we employ the queue approach.

We initialize for $p = 1$ a doubly-linked list linking together the nonzero entries in $\boldsymbol{c_1}$:

- *left*[1..n+1] is an array in which for every $i \in 1..n+1$, *left*[i] is the rightmost nonzero entry in $\boldsymbol{c}[i]$ to the left of location i (zero if no such location exists);
- *right*[0..n] is an array in which for every $i \in 0..n$, *right*[i] is the leftmost nonzero entry in $\boldsymbol{c}[i]$ to the right of location i (n+1 if no such location exists).

The arrays *left* and *right* are then used to ensure that only nonzero entries in $\boldsymbol{c}$ are visited for each value of p; whenever an entry at location i in $\boldsymbol{c}$ becomes zero, i is removed from the doubly-linked list.

Since now the processing in each level p is proportional only to the nonzero elements in $\boldsymbol{c_p}$, the time bound of Theorem 1 holds *a fortiori* for DIST3:

Theorem 2. *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 $4n$+$O(1)$ words and n bits of additional storage.*

In fact it is unclear whether DIST3 will run more quickly than DIST2 in the average case: the extra housekeeping required to maintain the arrays *left* and *right*, and the generally less efficient update of PROC, could negate any advantage gained by hopping zero entries in $\boldsymbol{c}$. Our main purpose in introducing DIST3 is to show how the idea of hopping can be implemented; we will extend this technique further in the next section.

4 Algorithm DIST4

Here we revert to the methodology of DIST1 — processing individual entries in $\boldsymbol{c}$ rather than entire refinement classes — combined with the hop strategy introduced in DIST3. However, we hop not only zero entries in $\boldsymbol{c}$ but also ***triangles*** induced by ***runs*** of the form

$$\boldsymbol{c_p}[i..i+k] = q, q+1, \ldots, q+k \tag{5}$$

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

$$\boldsymbol{c_1}[6..8] = 4, 5, 6$$

and

$$\boldsymbol{c_2}[9..12] = 6, 7, 8, 9.$$

The following result, easily proved, shows how a triangle in levels $p, p+1, \ldots, p+k-1$ can be inferred from a run of k 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 array $\boldsymbol{c}[1..n+1]$ that we employ in all the versions of DIST, DIST4 also makes use of two circular hop queues, $Ohop$ (zero-hops) and $Thop$ (triangle-hops), each of size $\leq \lceil \frac{n}{2} \rceil$.

During the processing we keep track of ***Lend***, the position of the first non-zero entry in $\boldsymbol{c}[1..n]$ from the left, and ***Rend***, the position of the first non-zero entry from the right. We maintain the zero-hops from the non-zero to the next non-zero position, e.g. 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 maitain the hops from the rightmost position of the triangle to the first position to the left that is not a part of the triangle, e.g. for $\boldsymbol{c}[4..8] =$ 6 10 11 12 15 we will hop from $\boldsymbol{c}[7] = 12$ to $\boldsymbol{c}[4] =$ 6. Thus, it is possible that the end of a hop may be to the left of a start of another hop (in just one case, when a triangle-hop is followed immediately by a zero-hop; an illustration of the situation: $\boldsymbol{c}[3..11] =$ 1 0 0 0 3 4 5 6 10, where we have a triangle-hop from $\boldsymbol{c}[10] = 6$ to $\boldsymbol{c}[6] = 0$ and a zero-hop from $\boldsymbol{c}[7] = 3$ to $\boldsymbol{c}[3] = 1$). We also allow that the end of a hop (the value of *to*) be smaller than ***Lend***.

For simplicity, in the high-level pseudocode for DIST4 we shall use the following terminology:

- *to delete hop* means to remove it from the queue
- *to transfer hop* means: the hop is deleted, the first value (hop $from$) is decremented by 1, and if the new value of $from$ is > 0, then the new (shorter) hop is inserted at the end of the queue
- *to copy hop* means to delete it and to insert it at the end as is (i.e. unchanged).
- *to extend hop to r* means to raplace the *to* value of the hop by the smaller value of r (extending the hop to the left).
- *to create hop* means to insert the appropriate pair $from$ and *to* at the end of the queue.

A broad outline of the algorithm in a high-level pseudocode is given in Figure 3. The body of the while loop consists of a rather anwieldy nested if statement. When we implemented the algorithm in C++, it was replaced by multiply nested if statements in much more practical way, but much less conducive for presentation. The nested if statement represents 29 rules of how to maintain the zero-hop queue and the triangle-hop queue and how to manipulate the index i from left to right making all the "hops".

In order to estimate space complexity of DIST4 as $3*n$ of machine words, we need to ascertain that each hop queue can have at most

```
— as in DIST1, compute c[1..n] for level 1
    set zero-hop and triangle-hop queues for level 1
    set Lend to the position of the first non-zero entry of c from the left
    set Rend to the position of the first non-zero entry of c from the right
p ← 1
i ← Rend
while i ≥ Lend do
  cprev ← c[i]      - rember original value of c[i]
  j ← i             - remember original value of i
  calculate new vale of c[i] as in DIST1 (note that c[i] is necessarily non-zero)
  if i = Rend and c[i] = 0 and there is a zero-hop (i, m) then
    delete the hop; Rend ← i ← m
  elsif i = Rend and c[i] = 0 and there is a triangle-hop (i, m) then
    transfer the hop; Rend ← Rend−1; i ← m
  elsif i = Rend and c[i] = 0 then
    Rend ← Rend−1; i ← i−1
  elsif i = Lend and c[i] = 0 and there is a zero-hop (m, i) then
    delete the hop; Lend ← m
  elsif i = Lend and c[i] = 0 then
    Lend ← Lend+1
  elsif 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 ← m
  elsif 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 ← m
  elsif c[i] = 0 and there is a zero-hop (r, i) then
    extend the hop to i+1; i ← i−1
  elsif 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 ← m
  elsif c[i] = 0 and there is a zero-hop (i, m) then
    delete the hop; create a new zero-hop (i+1, m); i ← m
  elsif c[i] = 0 then
    create a new triangle-hop (i+1, i−1); i ← i−1
  elsif i = Rend and c[i] ≠ 0 and there is a zero-hop (i, m) then
    copy the hop; i ← m
  elsif i = Rend and c[i] ≠ 0 and there is a triangle-hop (i, m) and
    cprev = c[i] then
    copy the hop; i ← m
  elsif i = Rend and c[i] ≠ 0 and there is a triangle-hop (i, m) and
    cprev ≠ c[i] then
    transfer the hop; i ← m
  elsif i = Lend and c[i] ≠ 0 and c[i]+1 = c[i+1] and
    there is a triangle-hop (r, i) then
    extend the hop to i−1; i ← i−1
  elsif i = Lend and c[i] ≠ 0 and c[i]+1 = c[i+1] then
    create a new triangle-hop (i+1, i−1); i ← i−1
  elsif i = Lend and c[i] ≠ 0 then
    i ← i−1
```

con't on next page

Fig. 3. DIST4 - the maintanance of zero-hop and traingle-hop queues

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

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

Fig. 4. DIST4 - continuation from the previous page

$\lceil \frac{n}{2} \rceil$ elements as stated previously. The following easy lemma provides the proof.

Lemma 2. *For any input string* $\boldsymbol{s}$ *and any level of processing, the number of the same hops (zero-hops or triangle-hops) is* $\leq \lceil \frac{n}{2} \rceil$

Proof. The following argument can be made for either zero-hops or triangle-hops. So we will just talk about hops. Let $f(r)$ be the number of hops that start at a position $\leq r$. We will show by induction over r that $f(r) \leq \lceil \frac{r}{2} \rceil$. $\boldsymbol{c}$ refers to the array as produced by the algorithm.
For r=2, we can have at most 1 triangle hop or 1 zero-hop.
For r+1 there are 3 possibilities:

1. There is no hop from r+1 to some $m \leq r$. Then $f(r+1) = f(r) \leq \lceil \frac{r}{2} \rceil \leq \lceil \frac{r+1}{2} \rceil$.
2. There is a hop from r+1, and there is no hop from r, then $f(r+1) \leq f(r)+1 = f(r-1)+1 \leq \lceil \frac{r-1}{2} \rceil + 1 = \lceil \frac{r+1}{2} \rceil$.
3. There is a hop from r and from r+1. If the hop from r+1 were a zero-hop, then $\boldsymbol{c}[r+1] \neq 0$ while $\boldsymbol{c}[r]=0$. But no hop starts at a position with a zero value, a contradiction with the fact that there is a hop starting at r. Hence the hop from r+1 must be a triangle hop, and so does the hop starting from r. But an easy inspection of DIST4 shows the algorithm maitains the hops of the same kind in a strictly decreasing manner when a hop's *to* is strictly bigger than the next hop's $from$. Therefore, this case cannot happen.

Note that each "rule" in the body of the while loop in DIST4 is executed in constant time and that only one "rule" is used in a single pass through the while loop body. Thus, the time complexity of DIST4 is that of DIST1 less all the processing of the "hopped" entries. To properly estimate the number of steps "saved" by not processing the "hopped" entries has so far resisted our attempts at analysis. As a consequence we are not able to state either worst-case or average-case nontrivial upper bounds on the asymptotic complexity of DIST4 (besides the obvious fact that it must be no worse than DIST1). However, we have performed experiments on pseudorandom strings and on long strings drawn from the Calgary corpus [6], as well as on special strings, such as Fibonacci strings, that should be close

to worst case for our algorithm. The behaviour of the algorithm on pseudorandom and Calgary corpus strings appears to be linear, and on Fibonacci strings marginally greater than linear. We therefore state

Conjecture 1. On strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an indexed alphabet, Algorithm DIST4 executes in $O(n \log n)$ worst-case and $O(n)$ average-case time, while using $2n+O(1)$ additional words of memory.

5 Concluding Remarks

In this paper we have introduced the idea of a quasi suffix array, then presented four variants of an algorithm DIST to construct it, two of which (DIST2 & DIST3) execute in $O(n \log n)$ average case time, one of which we conjecture to execute in $O(n)$ average case time. The quasi suffix array provides the same information that a suffix array does, but the suffixes are not necessarily sorted in lexicographic order; this is not a disadvantage for many applications (for example, pattern-matching or computing repeating substrings), and allows the construction algorithms to be simpler and therefore faster. Indeed, as we have seen, since the construction of the first level $\boldsymbol{c_1}$ yields an indexed alphabet, an order is implicitly imposed on the suffixes of $\boldsymbol{x}$, even though it does not have to be lexicographic.

To see the relationship between suffix arrays and quasi suffix arrays, we return to our original example:

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

Recall that the quasi suffix array is

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

reflecting the relationship

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

for every $i \in 3..n$. In this case the corresponding suffix array in its usual form would be

$$\begin{array}{rl} & \begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \end{array} \\ \boldsymbol{\lambda'_x} = & 8\ 3\ 11\ 6\ 1\ 9\ 4\ 12\ 7\ 2\ 10\ 5\ 13 \\ \boldsymbol{\pi'_x} = & 0\ 4\ 3\ 1\ 6\ 5\ 3\ 2\ 0\ 5\ 4\ 2\ 1, \end{array}$$

reflecting the relationship

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

for every $i \in 2..n$. Observe that $\boldsymbol{\pi'_x}$ is just a permutation of $\boldsymbol{\pi_x}$.

References

[1] Alberto Apostolico, **The myriad virtues of subword trees**, *Combinatorial Algorithms on Words (NATO ASI Series F12)*, Alberto Apostolico & Zvi Galil (eds.), Springer-Verlag (1985) 85-96.

[2] Alberto Apostolico & Andrzej Ehrenfeucht, **Efficient detection of quasiperiodicities in strings**, *TCS 119* (1993) 247-265.

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

[4] Alberto Apostolico & Franco P. Preparata, **Optimal off-line detection of repetitions in a string**, *TCS 22* (1983) 297-315.

[5] Gerth S. Brodal & Christian N. S. Pedersen, **Finding maximal quasiperiodicities in strings**, *Proc. Eleventh Annual Symp. Combinatorial Pattern Matching*, Raffaele Giancarlo & David Sankoff (eds.), Lecture Notes in Computer Science 1848, Springer-Verlag (2000) 397-411.

[6] Calgary Corpus

`http://links.uwaterloo.ca/calgary.corpus.html`

[7] M. T. Chen & Joel Seiferas, **Efficient and elegant subword-tree construction**, *Combinatorial Algorithms on Words (NATO ASI Series F12)*, Alberto Apostolico & Zvi Galil (eds.), Springer-Verlag (1985) 97-107.

[8] Maxime Crochemore, **An optimal algorithm for computing the repetitions in a word**, *IPL 12-5* (1981) 244-250.

[9] Maxime Crochemore & R. Verin, **Direct construction of compact directed acyclic word graphs**, *Proc. Eighth Annual Symp. Combinatorial Pattern Matching*, Lecture Notes in Computer Science 1264, Springer-Verlag (1997) 116-129.

[10] Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc.* 38^{th} *Annual IEEE Symp. Foundations of Computer Science* (1997) 137-143.

[11] 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.

[12] S. Kurtz, *Reducing the Space Requirement of Suffix Trees*, Tech. Rep. 98-03, University of Bielefeld (1998).

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

[14] Edward M. McCreight, **A space-economical suffix tree construction algorithm**, *JACM 32-2* (1976) 262-272.

[15] Donald R. Morrison, **PATRICIA — practical algorithm to retrieve information coded in alphanumeric**, *JACM 15-4* (1968) 514-534.

[16] Jens Stoye & Dan Gusfield, **Simple and flexible detection of contiguous repeats using a suffix tree**, *Proc. Ninth Annual Symp. Combinatorial Pattern Matching*, Martin Farach-Colton (ed.), Lecture Notes in Computer Science 1448, Springer-Verlag (1998) 140-152.

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

[18] Esko Ukkonen, **Constructing suffix trees on-line in linear time**, *Proc. IFIP 92*, vol. I (1992) 484-492.

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