Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 387 (2007) 258–272
Theoretical Computer Science
www.elsevier.com/locate/tcs

Faster suffix sorting

N. Jesper Larsson[a], Kunihiko Sadakane[b,*]

[a] *Apptus Technologies, Ideon Science Park, SE-22370 Lund, Sweden*
[b] *Department of Computer Science and Communication Engineering, Kyushu University, Motooka 744, Nishi-ku, Fukuoka 819-0395, Japan*

Abstract

We propose a fast and memory-efficient algorithm for lexicographically sorting the suffixes of a string, a problem that has important applications in data compression as well as string matching.

Our algorithm eliminates much of the overhead of previous specialized approaches while maintaining their robustness for degenerate inputs. For input size n, our algorithm operates in only two integer arrays of size n, and has worst-case time complexity $O(n \log n)$.

We demonstrate experimentally that our algorithm has stable performance compared with other approaches.

Keywords: Suffix arrays; Burrows–Wheeler transform

1. Introduction

Suffix sorting is the problem of lexicographically ordering all the suffixes of a string. The suffixes are represented as a list of integers denoting their starting positions.

This has at least two important applications. One is construction of a *suffix array* [20] (also known as *PAT array* [9]), a data structure for pattern matching that supports some of the operations of a *suffix tree* [31], generally slower than the suffix tree but requiring less space. When additional space is allocated to supply a *bucket* array or a *longest common prefix* array, the time complexity of basic operations closely approaches those of the suffix tree.

Another application is in data compression. The *Burrows–Wheeler Transform* [5] is a process that has the capability of concentrating repetitions in a string, which facilitates data compression. Even in a rudimentary form, Burrows–Wheeler compression matches substantially more complex modelling schemes in compression performance, which is shown theoretically [22,7] and experimentally [1,30,28]. Suffix sorting is a computational bottleneck in the Burrows–Wheeler Transform, and an efficient sorting method is crucial for any implementation of this compression scheme. We refer to the cited material for details. Recent developments [10,8] showed that the suffix array is implicitly stored in the Burrows–Wheeler Transform, implying that efficient pattern matching can be done on the compressed string.

Suffix sorting differs from ordinary string sorting in that the elements to sort are overlapping strings, whose lengths are linear in the input size n. This implies that a comparison-based algorithm, which requires $\Omega(n \log n)$ comparisons,

* Corresponding author. Tel.: +81 92 802 3647; fax: +81 92 802 3647.
E-mail addresses: jesper.larsson@apptus.com (N.J. Larsson), sada@csce.kyushu-u.ac.jp (K. Sadakane).

0304-3975/$ - see front matter
doi:10.1016/j.tcs.2007.07.017

may take $\Omega(n^2 \log n)$ time for suffix sorting, and analogously a normal radix sorting algorithm may take $\Omega(n^2)$ time. Fortunately, these bounds can be surpassed with specialized methods. Manber and Myers [20] presented an elegant radix-sorting-based algorithm which takes $O(n \log n)$ time, though its actual running time is not fast.

Theoretically suffix sorting can be done in linear time by building a suffix tree and obtaining the sorted order from its leaves [6]. However, a suffix tree involves considerable overhead, particularly in space requirements, which commonly makes it too expensive to use for suffix sorting alone.

The goal of this paper is to develop a suffix sorting algorithm which has good worst-case time complexity and actual running time using memory with reasonable size. Our time complexity is $O(n \log n)$. It is the same asymptotic bound as that of Manber and Myers, but our algorithm avoids a large portion of the work compared to theirs, which is demonstrated by a large advantage in practical performance.

After the preliminary version of this algorithm has been presented [27], many algorithms for suffix sorting have been proposed which have worst-case linear time complexity [15,19,18], and worst-case $O(n \log n)$ time complexity using $o(n)$ additional space [4]. These algorithms were surveyed by Puglisi et al. [25]. However, although several alternatives exist that are space efficient and faster for many types of input [14,23], running in only $O(n)$-bit ($O(n/\log n)$-word) memory space [13,12], our evaluation as well as others' [26,25] have shown that our algorithm possesses a superior combination of robustness (running time does not degenerate for any input) with overall competitive performance. Because of its robustness, our algorithm is used as a backup algorithm of the widely-used compression program `bzip2` [30], and as a subroutine of the suffix sorting algorithm of Burkhardt and Kärkkäinen [4].

Section 2 recapitulates this and other approaches connected with our algorithm. Section 3 presents our algorithm. (A preliminary version of this algorithm has previously been presented by Sadakane [27].) Section 4 analyzes time complexity. Section 5 present various refinement techniques. Section 6 presents a practical implementation that includes the refinements, and results of an experimental comparison with other suffix sorting implementations. Finally, Section 7 concludes by recapitulating our findings.

Problem definition

We consider a string $X = x_0 x_1 \ldots x_n$ of $n+1$ symbols, where the first n symbols comprise the actual input string and $x_n = \$$ is a unique sentinel symbol. We choose to regard \$, which may or may not be represented as an actual symbol in the implementation, as having a value below all other symbols. By S_i, for $0 \leq i \leq n$, we denote the suffix of X beginning in position i. Thus, $S_0 = X$, and $S_n = \$$ is the first suffix in lexicographic suffix order.

The output of suffix sorting is a permutation of the S_i, contained in an integer array I. Throughout the algorithm, I holds all integers in the range $[0, n]$. Ultimately, these numbers are placed in order corresponding to lexicographic suffix order, i.e., $S_{I[i-1]}$ lexicographically precedes $S_{I[i]}$ for all $i \in [1, n-1]$. We refer to this final content of I as the *sorted suffix array*.

Thus, suffix sorting in more practical terms means sorting the integer array I according to the corresponding suffixes. We interchangeably refer to the integers in I and the suffixes they represent; i.e., *suffix* i, where i is an integer, denotes S_i.

2. Background

This section presents the background material for our algorithm as well as previous work and alternative approaches to suffix sorting.

2.1. Suffix sorting in logarithmic number of passes

An obvious technique for suffix sorting is to start by sorting according to only the first symbol of each suffix, then successively refining the order by expanding the considered part of each suffix. If one additional symbol per suffix is considered in each pass, the number of passes required in the worst case is $\Omega(n)$. However, fewer passes are needed if we exploit the fact that each suffix is a prefix of another suffix.

The key for reducing the number of passes is a doubling technique, originating from Karp, Miller, and Rosenberg [16], which allows the positions of the suffixes after each sorting pass to be used as the sorting keys for preceding suffixes in the next pass.

Define the *h-order* of the suffixes as their order when sorting lexicographically, considering only the initial h symbols of each suffix. The h-order is not necessarily unique when $h < n$. Now consider the following observation:

Observation 1 (*Manber and Myers*). *Sorting the suffixes using, for each suffix S_i, the position in the h-order of S_i as the primary key, and the position of $S_i + h$ in the same order as the secondary key, yields the $2h$-order.*

Using this observation, we first sort the suffixes according to the first symbol of each suffix, using the actual contents of the input, i.e., x_i is the sorting key for suffix i. This yields the 1-order. Then, in pass j, for $j \geq 1$, we use the position that suffix $i + 2^{j-1}$ obtained in pass $j - 1$ (where pass 0 refers to the initial sorting step) as the sorting key for suffix i. This doubles the number of considered symbols per suffix in each pass, and only $O(\log n)$ passes in total are needed.

Manber and Myers [20] used this observation to obtain an $O(n \log n)$ time algorithm through bucket sorting in each pass. An auxiliary integer array, which we denote V, is employed to maintain constant-time access to the positions of the suffixes in I.

The main implementation given by Manber and Myers uses, in addition to storage space for X, I, and V, an integer array with n elements, to store counts. However, they sketch a method for storing counts in temporary positions in V with maintained asymptotic complexity.

A substantially cleaner solution with reduced constant factors has been presented as source code by McIlroy [24]. Some properties of McIlroy's implementation are discussed in Section 5.3.

2.2. Ternary-split quicksort

The well known Quicksort algorithm [11] recursively partitions an array into two parts, one with smaller elements than a *pivot element* and one with larger elements. Then the parts are processes recursively until the whole array is sorted.

Where traditional Quicksort partitioning mixes the elements equal to the pivot into – depending on the implementation – one or both of the parts, a ternary-split partition generates *three* parts: one with elements smaller than the pivot, one with elements equal to the pivot, and one with larger elements. The *smaller* and *larger* parts are then processed recursively while the *equal* part is left as is, since its elements are already correctly placed.

Bentley and McIlroy [2] analyze and implement an in-place ternary-split partitioning method called *split-end partitioning*. The comparison-based subroutine used in our algorithm is directly derived from their implementation.

2.3. Ternary string-sorting and trees

Bentley and Sedgewick [3] employ a ternary-split Quicksort to the problem of sorting an array of strings, which results in the following algorithm: Start by partitioning the whole array based on the first symbol of each string. Then process the *smaller* and *larger* parts recursively in exactly the same manner as the whole array. The *equal* part is also sorted recursively, but the partitioning is done considering the *second* symbol of each string. Continue this process recursively: each time an *equal* part is being processed, move the position considered in each string forward by one symbol.

The result is a fast string sorting algorithm which, although it is not specialized for suffix sorting, has been used successfully for suffix sorting in the widely spread Burrows–Wheeler compression program `bzip2` [30].

Our proposed algorithm does not explicitly make use of the Bentley–Sedgewick string sorting method, but the techniques are related. This is apparent from the time complexity analysis in Section 4: Bentley and Sedgewick considered the implicit *ternary tree* that emerges from their algorithm when regarding each call to the partitioning routine as a node with three outgoing edges, one for each part of the splitting. We use this tree as a tool for our analysis.

3. A faster suffix sort

Usually, the sorted order of suffixes can be achieved by considering only the first few symbols of each suffix. This holds for random data, as well as common real-life data (see Section 6.2). As a result, a robust specialized suffix sorting method based on the algorithm of Manber and Myers can often be outperformed in practice by an ad hoc string sorting method, optimized for sorting short strings.

To improve the Manber–Myers algorithm, we need to remove the unnecessary scanning and idle reorganizing of already sorted parts of the I array. Still, we wish to maintain the robust worst-case behaviour for repetitive strings

which *do* also occur in practice. Furthermore, we do not want to increase the amount of auxiliary space, which would be necessary if a suffix tree was used.

We now present a suffix sorting algorithm that accomplishes this. The various techniques explained in Section 2 are components of our algorithm. This section describes a basic version of the algorithm. In Section 5, we describe refinements to the algorithm that improve both running time and storage space. (Sadakane [27] presented a preliminary version of this algorithm.)

Our algorithm inherits the use of Observation 1 to double the number of considered symbols over a number of sorting passes, as well as the array V to gain constant-time access to suffix positions, from Manber and Myers (see Section 2.1). To refrain from scanning the whole array in each pass, we mark which sections of the suffix array are already finished and skip over them when sorting. We use ternary-split Quicksort (Section 2.2) as our sorting subroutine.

The following concepts enable us to express the rules of individual sorting passes:

Definition 2. The following applies when I is in h-order:

- A maximal sequence of adjacent suffixes in I which have the same initial h symbols is a *group*.
- A group containing at least two suffixes is an *unsorted group*.
- A group containing only one suffix one is a *sorted group*.
- A maximal sequence of adjacent sorted groups is a *combined sorted group*.

We number the groups so that the numbers reflect the order in which the groups appear in I. This is necessary to allow groups to be used as sorting keys. It is convenient to define the number of a group $I[\, f \ldots g \,]$ as one of the numbers $f \ldots g$. For reasons that will become apparent in Section 5, we choose the following group numbering:

Definition 3. The *group number* of a group that occupies the subarray $I[\, f \ldots g \,]$ is g.

During sorting, the array V stores group numbers. $V[\, i \,] = g$ reflects that suffix i is currently in group number g.

Furthermore, we employ a conceptual array L that holds the lengths of unsorted groups and combined sorted groups, for all starting positions of such groups. To distinguish between them, we store positive numbers for the former and negative numbers (the negated length) for the latter. Thus, if the subarray $I[\, f \ldots g \,]$ is an unsorted group, we store $g - f + 1$ in $L[\, f \,]$; if it is a combined sorted group, we store $-(g - f + 1)$ instead. In Section 5.1, we show how the relevant information of L can be superimposed on I without extra storage space.

Note the difference in treatment of sorted groups between V and L: in L, we store lengths of *combined* sorted groups; in V, we store group numbers for unit length sorted groups.

The first step of the algorithm places the suffixes – represented as numbers 0 through n – into I, sorted according to the first symbol of each suffix. This step consists of integer sorting where the keys are drawn from the input alphabet. After this step, I is in 1-order. We initialize V and L accordingly.

Then a number of passes for further sorting follow. At the beginning of the jth such pass, I is in h-order, where $h = 2^{j-1}$. Note the following:

Observation 4. *When I is in h-order, each suffix in a sorted group is uniquely distinguished from all other suffixes by its first h symbols.*

This implies that all suffixes in sorted groups are already in their final location, and only unsorted groups need to be rearranged.

We sort the unsorted groups using the group number of suffix $i + h$ as the key for suffix i, which, by Observation 1, places I in $2h$-order. We then split groups between suffixes with non-equal keys, updating V and L. When setting the lengths in L, we combine adjacent groups so that they can be efficiently skipped over in subsequent passes.

Fig. 1 shows the basic algorithm. Its time complexity is analyzed in Section 4. The key to the good performance of this algorithm is the utilization of Observation 4 in Step 4: the group lengths stored in L allow us to skip over sorted groups completely while we continue to process unsorted groups. Even though this is not crucial for asymptotic time complexity, it allows a substantial reduction of running times compared to the Manber–Myers algorithm, whose sorting approach does not allow skipping the parts of the array that are already completely sorted. For marking of groups in Step 5, we can use the sign bits of I. With the refinement shown in Section 5.2, the necessity of this marking disappears.

(1) Place the suffixes, represented by the numbers $0, \ldots, n$, in I. Sort the suffixes using x_i as the key for i. Set h to 1.
(2) For each $i \in [0, n]$, set $V[i]$ to the group number of suffix i, i.e., the last position in I that holds a suffix with the same initial symbol as suffix i.
(3) For each unsorted group or combined sorted group occupying the subarray $I[f \ldots g]$, set $L[f]$ to its length or negated length respectively
(4) Process each unsorted group in I with ternary-split Quicksort, using $V[i + h]$ as the key for suffix i.
(5) Mark splitting positions between non-equal keys in the unsorted groups.
(6) Double h. Create new groups by splitting at the marked positions, updating V and L accordingly.
(7) If I consists of a single combined sorted group, then stop. Otherwise, go to 4.

Fig. 1. The basic version of our proposed algorithm.

Note that Step 4 does not check that $i + h$ is in the legal range (at most n) when referring to $V[i + h]$. This is not necessary, because of the unique \$ symbol that terminates X: All suffixes $n - h + 1, \ldots, n$ have length at most h, and the \$ symbol is therefore included in the considered length of these suffixes, which implies that their positions in the sorted suffix array must already have been uniquely determined. They are therefore all in sorted groups, and we never attempt to access their sorting keys.

Fig. 2 shows a run of the algorithm with the string 'tobeornottobe' as input. The top section of the figure shows X, the input with the unique \$ symbol attached to the end. The second section shows the result of sorting the suffixes according to their first symbols. Negative numbers in $L[0]$, $L[5]$ and $L[10]$ denote that suffixes 0, 5 and 10 are already sorted.

The next, single line, section of the figure shows the keys used for the $h = 1$ sorting pass. In this pass, the sorting key of suffix i is $V[I[i] + 1]$. Suffixes in groups 2 ($I[1 \ldots 2]$), 4 ($I[3 \ldots 4]$), 9 ($I[6 \ldots 9]$) and 13 ($I[11 \ldots 13]$) are sorted separately, according to these keys. The result, shown in the next section of the figure, is that suffixes are sorted according to their first two symbols. Groups have been split by updating $L[i]$ and $V[i]$ for i ranging over the just sorted groups.

Analogously, the next sorting pass, for $h = 2$, processes still unsorted groups (2, 7, and 12) by sorting according to $V[I[i] + 2]$, and obtains the suffix order according to the first four symbols of each suffix. Finally, the single remaining unsorted group (12) is sorted according to $V[I[i] + 4]$, again doubling the number of considered symbols. This concludes the suffix sorting, since the longest repeated string in the input is shorter than eight symbols, and leaves I holding the sorted suffix array as shown at the bottom of the figure.

4. Time complexity

Consider the algorithm in Fig. 1. The time for the first sorting step is between $O(n)$ and $O(n \log n)$ depending on the sorting method used. Initialization of V and L in Step 2 and Step 3 are both performed in linear time in a left-to-right sweep. The dominant part of the algorithm is thus the loop comprising Step 4 through Step 7, which is performed up to $\log n$ times. Clearly, the time for each run through this loop can be bounded by $n \log n$ – the time to sort all of I with a comparison-based sorting method – yielding an upper bound of $O(n(\log n)^2)$ for the total time complexity. However, the more detailed complexity analysis below gives an $O(n \log n)$ worst-case bound.

Our sorting subroutine is Quicksort with a ternary-split partition, such as the split-end partition of Bentley and McIlroy (see Section 2.2). We assume that the true median is chosen as pivot element to guarantee that the array is partitioned as evenly as possible. This requires that the median is located in linear time, for example using the algorithm of Schönhage, Paterson, and Pippenger [29], as part of the partitioning routine. In practice, this is rarely desirable, due to large constant factors, and hardly necessary. There exist a range of pivot-choice methods which balance guaranteed worst case versus expected performance [2].

For simplicity, we assume in the following analysis that the same Quicksort method is used for the initial sorting in Step 1. Employing a different sorting algorithm for initial sorting (considered in Section 5) may improve practically the practical behaviour of the algorithm, but does not influence the asymptotic worst-case time complexity.

We view the sorting process as a construction of an implicit ternary tree, analogous to the search tree discussed by Bentley and Sedgewick [3]. In this tree, each call to the partitioning routine corresponds to a node in the tree. The first partitioning of the whole array in Step 1 corresponds to the root of the tree. Each node has three subtrees:

h	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	x_i	t	o	b	e	o	r	n	o	t	t	o	b	e	$
	$I[i]$	13	2	11	3	12	6	1	4	7	10	5	0	8	9
	$V[I[i]]$	0	2	2	4	4	5	9	9	9	9	10	13	13	13
	$L[i]$	−1	2		2		−1	4				−1	3		
1	$V[I[i]+h]$		4	4	9	0		2	10	13	2		9	13	9
	$I[i]$		2	11	12	3		1	10	4	7		0	9	8
	$V[I[i]]$		2	2	3	4		7	7	8	9		12	12	13
	$L[i]$	−1	2		−3			2		−3			2		−1
2	$V[I[i]+h]$		8	0				4	3				2	2	
	$I[i]$		11	2				10	1				0	9	
	$V[I[i]]$		1	2				6	7				12	12	
	$L[i]$	−11											2		−1
4	$V[I[i]+h]$												8	0	
	$I[i]$												9	0	
	$V[I[i]]$												11	12	
	$L[i]$	−14													
	$I[i]$	13	11	2	12	3	6	10	1	4	7	5	9	0	8

Fig. 2. Example run of the basic algorithm with the input string 'tobeornottobe'. Time flow is from the top down in the table. Sections with h values show the keys used when sorting the entries that have equal $V[I[i]+h]$ values. Other sections show the parts of the contents of X, I, V, and L that are accessed at each sorting stage.

a middle subtree which corresponds to the subarray containing elements equal to the pivot after the partitioning, and left- and right-subtrees corresponding to the subarrays holding smaller and larger elements respectively. All internal nodes have non-empty middle subtrees, while their left- or right-subtrees are empty for subarrays with less than three distinct keys. The tree has n leaves, corresponding to all the elements in sorted order. Fig. 3 shows an example ternary tree that corresponds to the same input and sorting process as Fig. 2.

The following lemma bounds the height of the ternary tree:

Lemma 5. *The path length from the root to any leaf in the ternary tree is at most* $2\log n + 3$.

Proof. Consider first the number of middle-subtree roots on a walk from the root to a leaf in the tree. At the first such node encountered, only the first symbol of each suffix is considered by the sorting. Then, at each subsequent middle-subtree root encountered, the number of symbols considered by the sorting is twice as large as at the previous one. Consequently, the full length of any suffix is considered after encountering at most $\log n + 1$ middle-subtree roots, at which time sorting is done.

Now consider the left- and right-subtree roots. For each such node encountered on a walk from the root to a leaf, the number of leaves in its subtree is at most half compared to the previous one, since partitioning is done as evenly as possible. Thus, we are down to a single leaf after encountering at most $\log n + 1$ left- or right-subtree roots.

Summing the root and the maximum number of middle-, left-, and right-subtree roots on a path, we have a path length of at most $2\log n + 3$. □

We now consider the amount of work that corresponds to each depth level of the ternary tree.

Lemma 6. *Partitioning operations corresponding to all the nodes of any given depth of the tree take at most* $O(n)$ *time.*

Proof. Partitioning a subarray takes time linear in its size. The initial array, whose partitioning corresponds to the root, has $n + 1$ elements, and since no overlapping subarrays are ever assigned to different subtrees of any node, the total number of elements in all subarrays at any given depth is at most $n + 1$. The total time for partitioning at this depth is thus $O(n)$. □

We can now state the following tight bound:

Theorem 7. *Suffix sorting with the algorithm in Fig. 1 can be done in* $O(n \log n)$ *worst-case time.*

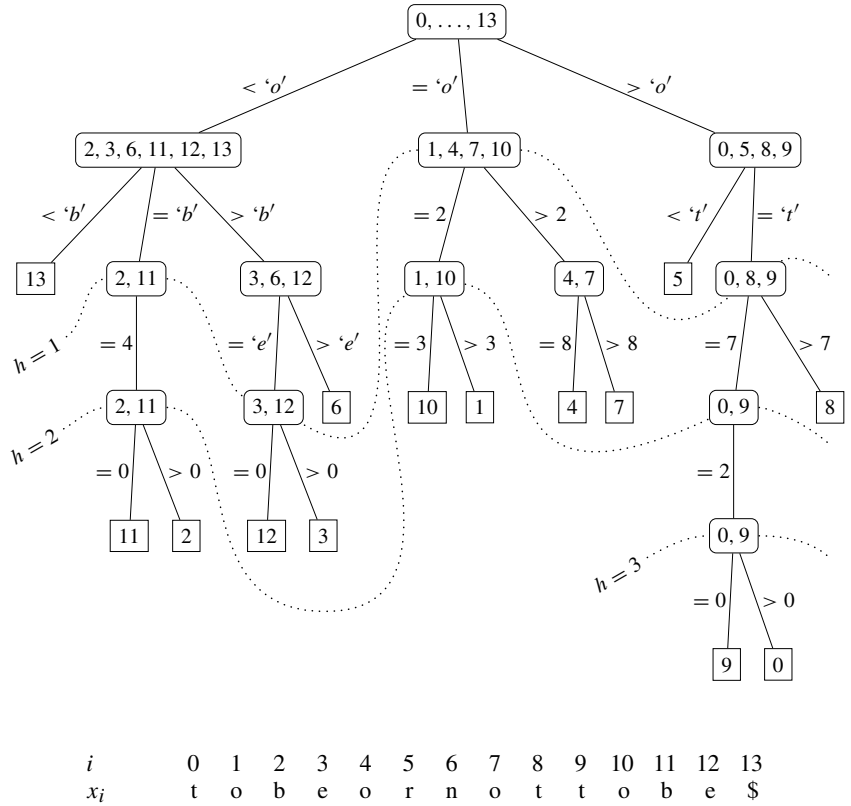

Fig. 3. An implicit ternary tree that emerges from sorting with the input string 'tobeornottobe'. The sorting process corresponds to that of Fig. 2. The suffixes processed in each partition operation are listed inside the node corresponding to that operation. The outgoing edges of each partitioning node are labeled with relation operations and pivot keys that determine the results of partitioning. (Different choices of pivot elements lead to different trees.) Dotted lines indicate the sorting passes of the algorithm.

Proof. Partitioning asymptotically dominates sorting time; splitting and combining groups are done in linear time on subarrays which are already sorted.

From Lemma 6, the total partitioning cost is at most $O(n)$ times the height of the ternary tree. Lemma 5 implies that the height of the tree is $O(\log n)$, and consequently the total partitioning time is $O(n \log n)$. □

5. Algorithm refinements

This section lists a number of refinements that reduce the time and space requirements of our algorithm. These are incorporated into the practical implementation described in Section 6.1.

5.1. Eliminating the length array

The only use of the information stored in the array L is to find right endpoints of groups in the *scanning-and-sorting* phase of the algorithm (Step 4 in Fig. 1). For combined sorted groups, this is to be able to skip over them in constant time, and for unsorted groups to use the endpoint as a parameter to the sorting subroutine. However, the endpoint of unsorted groups is directly known without using L, since it is equal to the group number according to Definition 3.

Consequently, we need only find alternative storage for the lengths of combined sorted groups to be able to get rid of the L array. For this, note that once a suffix has been included in a combined sorted group, the position in I where it resides is never accessed again. Therefore, we can reuse the subarrays of I that span sorted groups for other purposes, without compromising the correctness of the algorithm.

Of course, overwriting parts of I with other information means that I does not hold the desired output, the sorted suffix array, when the algorithm terminates. However, the information needed to quickly reconstruct this is present

in V. When the algorithm finishes, all parts of the suffix array are sorted groups, and since V holds group numbers of single-length sorted groups it is in fact at this point the inverse permutation of the sorted suffix array occurs. Hence, setting $I[\,V[\,i\,]\,]$ to i for all $i \in [\,0, n\,]$ reconstructs the sorted suffix array in I.

This allows us to use the first position of each combined sorting groups for storing its negated length. When we probe the beginning of the next group in the left-to-right scanning-and-sorting step, we check the sign of the number $I[\,i\,]$ in this position. If it is negative, $I[\,i \ldots i - I[\,i\,] + 1\,]$ is a combined sorted group; otherwise $I[\,i \ldots V[\,I[\,i\,]\,]\,]$ is an unsorted group.

5.2. Combining sorting and updating

After sorting, the algorithm in Fig. 1 scans the processed parts twice, in order to update the information in V and L. This is true both for the initial sorting and for each run through the loop in Step 4 through Step 7. We now show how this additional scanning can be eliminated.

First, note that concatenation of adjacent sorted groups into the maximal combined sorting groups can be delayed and performed as part of the scanning-and-sorting (Step 4) of the following iteration. This change is straightforward.

Furthermore, all further updating of group numbers and lengths can be incorporated into the sorting subroutine. This change requires some more consideration, since changing group numbers of some suffixes affects sorting keys of other suffixes. Therefore, updating group numbers before all unsorted groups have been processed must be done in such an order that no group is ever, not even temporarily, given a lower group number than a group residing in a higher part of I. With the ternary-split sorting routine we use, this poses no difficulty. We give the sorting routine the following schedule:

(1) Partition the subarray into three parts: smaller than, equal to, and larger than the pivot.
(2) Recursively sort the *smaller* part.
(3) Update group number and size of the *equal* part, which becomes a group of its own.
(4) Recursively sort the *larger* part.

Since the group numbers stored in V never increase – splitting groups always only involve *decreasing* group numbers – this keeps the sorting keys consistent. If the equal or the larger part were processed before the smaller part, some suffix in that part would get a smaller group number than a suffix in the smaller part. This would violate the lexicographic orders of the suffixes, and sorting the smaller part based on these contradicting group numbers would not produce the correct result.

This change may still influence the sorting process, but only in a positive direction: Some elements may now be directly sorted according to the keys they would otherwise achieve *after* the current sorting pass, and this effect may propagate through several groups. Although this does not affect the worst-case time complexity, it causes a non-trivial improvement in time complexity for some input distributions.

5.3. Input transformation

If we assume that the input alphabet is small enough for a symbol to be represented as a non-negative integer (which is invalid for only a few, less than practical, machine models), we can start by transferring the contents of X to V, and perform the initial sorting in Step 1 using $V[i]$ as the key for suffix i. This has the following potential advantages, all which to some degree originate from McIlroy [24]:

- By setting $h = 0$, we can use the exact same sorting subroutine for initial sorting as for subsequent sorting passes.
- Since we no longer access X, we do not need to keep it in primary storage during sorting. Indeed, if we do not wish to retain X, we can overlay V on X, eliminating the memory usage for this array completely.
- When transferring symbols from X to V, the alphabet can undergo any transformation as long as the order between the symbols is maintained.

McIlroy's implementation requires an alphabet transformation that represents the unique \$ symbol by zero, and maps the original symbols to integers in the range $[\,1, k)$, where $k - 1$ is the number of distinct symbols in the input. This transformed alphabet facilitates bucket sorting, which is essential in McIlroy's implementation, since it is based on the Manber–Myers algorithm.

We now develop alphabet transforms that can benefit our algorithm even though we do not use bucket sorting (except possibly for initial sorting, see Section 5.4). We assume for the remainder of this section that the input consists of integers in the range $[\,l, k)$ for some k and l, not counting the \$ symbol.

The possibility to introduce an explicit representation of the \$ symbol is a small but convenient effect of alphabet transformation. The simplest way to achieve this is to set $V[\,i\,]$ to $x_i - l + 1$ for all $i \in [\,0, n)$ when transferring from X, and set $V[\,n\,]$ to zero. Now, the rest of the algorithm does not have to pay any attention to range or alphabet limits.

A transform with direct impact on time complexity, similar to a variation described by Manber and Myers [20, page 944], is possible when the input range is small enough for several symbols to be aggregated into one integer. Let K denote $k - l + 1$, the size of the original alphabet including \$, and let r be the largest integer such that $K^r - 1$ can be held in one machine word. Now, for all $i \in [\,0, n\,]$, set

$$V[\,i\,] := \sum_{j=1}^{r} x_{i+j-1} \cdot K^{r-j}$$

where we define $x_i = 0$ for $i \geq n$.

This has the effect that initial sorting, where $V[\,i\,]$ is used as the key for suffix i, concerns not only the first symbol of each suffix, but the first r symbols. Therefore, subsequent sorting passes can start with h set to r instead of 1, and the number of sorting passes is reduced.

The transform can be computed in linear time independent of r through the alternative form

$$V[\,i+1\,] := (V[\,i\,] \bmod K^{r-1}) \cdot K + x_{i+r}$$

for $i > 0$. If K is rounded up to the next power of two, the multiplication and modulo operation can be replaced by faster *shift* and *and* operations.

Since r is highly dependent on K and thereby on k and l – the limits of the input alphabet range – it can be fruitful to tighten these limits as much as possible before computing the transform. Checking the minimum and maximum symbol values that actually occur in the input and adjusting k and l accordingly is a simple task that commonly yields a noticeable improvement.

A further improvement can be gained in many cases by compacting the alphabet prior to the symbol aggregating transform. Denote the set of symbols that occur in the input $\Sigma = \{s_1, \ldots, s_{|\Sigma|}\}$, where $s_i < s_j$ if and only if $i < j$. Replacing each symbol s_i in the input with its ordinal number i allows us to set $l = 0$ and $k = |\Sigma|$. If only a small subset of the allowed input alphabets is used, this can result in a substantially larger value of r than would otherwise be possible.

We denote the allowed range size of the original alphabet K_0. Unless K_0 is very large, the preparatory compaction transform can be efficiently computed using an auxiliary array of size K_0 (which may be overlaid on I): Positions in the array corresponding to used symbol numbers are marked, and ordinal numbers are then accumulated in the same array. The time complexity is $O(n + K_0)$.

5.4. Initial bucket sorting

The initial sorting step is quite separate from the rest of the algorithm and is not required to use the same sorting method as the following passes. Since this step must process all of the input in one single sorting operation, a substantial improvement can be gained by using a linear time bucket sorting algorithm, instead of a comparison-based algorithm that requires $\Omega(n \log n)$ time.

At this stage the array I does not yet contain any data. Therefore, if the alphabet size is at most $n + 1$, we can use I as an auxiliary bucketing array, not requiring any extra space. If the input alphabet is larger than $n + 1$ and cannot be readily renumbered, we cannot use this technique. However, in practice, this is unusual unless n is very small, in which case there is no need for a sophisticated sorting algorithm. (Note also that the Manber–Myers suffix sorting algorithm and similar techniques cannot function at all if the alphabet size is larger than $n + 1$.)

An even more substantial improvement can be gained by combining bucket sorting with transformation of the input alphabet as described in Section 5.3. In this case, when choosing the value of r – the number of original symbols to aggregate into one – we require not only that $K^r - 1$ can be held in one machine word, but also that it is at most n. The resulting transformed alphabet can be larger than the original one, but it still allows bucket sorting without allocating

extra space. Thus, using only linear time preprocessing, we allow the initial order of the suffixes to be sorted according to the first r symbols of each suffix. This commonly takes a substantial load off the main sorting routine.

6. Implementation and experiments

This section describes a practical implementation of the proposed suffix sorting algorithm, and an experimental comparison between this and other suffix sorting methods.

6.1. Implementation

We describe an implementation of our algorithm that includes the refinements of Section 5, and present source code in the C programming language [17]. Since the details for implementation of alphabet transformation (described in Section 5.3) and bucket sorting (described in Section 5.4) is not central to this work, we omit the source code for the functions that perform those operations. The full implementation, including alphabet transformation and bucket sorting, is available in the file *qsufsort.c* available from http://www.larsson.dogma.net/research.html.

The main suffix sorting routine is shown in Fig. 4. The parameters to this function are pointers to two arrays, that are to be used as the V and I arrays of the algorithm, and integers representing n, the input size, and the input alphabet limits k and l (see Section 5.3). When this function is called, the input should already have been transferred to the V array, but the alphabet not yet transformed, other than possibly with the initial compaction described in the last paragraph of Section 5.3.

After setting global variables that allow main arrays to be accessed by other functions, the first section of the suffix sort function performs alphabet transformation and initial sorting:

The *transform* function implements techniques described in Section 5.3. It transforms the alphabet and changes the contents of V accordingly, while maintaining the lexicographic order between suffixes:

- $V[\,n\,]$ is set to zero, representing the \$ symbol, and the previous n cells of the V array are assigned positive integers.
- r symbols of the original alphabet are aggregated into one, where r is the maximum integer such that $(k-l+1)^r \leq q$, and q is the last parameter in the call to *transform*. The value of r is kept as a global variable.

The transformed alphabet is $\{0, \ldots, j-1\}$ for some alphabet size $j \leq q+1$, where 0 represents the unique \$ symbol and q is a parameter to the transform function. The value returned by this function is j. (To simplify the bucket sorting routine, our *transform* implementation also under some circumstances *compacts* the alphabet after symbol aggregation, so that all integers less than j occur at least once in V.)

Before the call to *transform*, we check if n is large enough for I to be used as a bucket array for the given alphabet range, i.e., if $n \geq k-l$. If so, we call transform with the q parameter set to n, to guarantee that bucketing is still possible for the transformed alphabet. Then we use bucket sorting for initialization of I through a call to a separate function.

If the given alphabet range is larger than $n+1$ we do not use bucket sorting, since this would require extra space. In this case, we may just as well use the largest possible symbol aggregation, so we call the transform function with q value INT_MAX. Then we initialize I with the numbers 0 through n, and use our main ternary-split Quicksort subroutine *sort_split* for initial sorting. By setting h to zero before this call, we get the desired effect that the contents of $V[\,i\,]$ is used as the sorting key for suffix i.

This concludes the initialization phase. The suffix array has been sorted according to the first r symbols of each suffix, i.e., we can set h to r. I contains suffix numbers for unsorted groups, and negative group length values for sorted groups, according to the storage technique described in Section 5.1. (At this point, the sorted group length values are all -1, since the groups have yet to be combined.)

The main *while* loop of the routine runs for as long as I does not consist of a single combined sorted group of length $n+1$, i.e., until the first cell of I has got the value $-(n+1)$. The inner part of the loop consists of combining sorted groups that emerged from the previous sorting pass, with each other and with previously combined sorted groups, and refining the order in unsorted groups through calls to the function sort_split. This process follows the description in Section 5.1 and Section 5.2.

Finally, I, now filled with negative numbers denoting lengths of sorted sequences, is restored to the sorted suffix array from its inverse permutation, which the algorithm has produced in V. If the application of suffix sorting is

```
void suffixsort(int *x, int *p, int n, int k, int l)
{
  int *pi, *pk;
  int i, j, s, sl;

  V=x; I=p;              /* set global values.*/
  if (n >= k-l) {        /* if bucketing possible,*/
    j=transform(V, I, n, k, l, n);
    bucketsort(V, I, n, j); /* bucketsort on first r positions.*/
  } else {
    transform(V, I, n, k, l, INT_MAX);
    for (i=0; i<=n; ++i)
      I[i]=i;            /* initialize I with suffix numbers.*/
    h=0;
    sort_split(I, n+1);  /* quicksort on first r positions.*/
  }
  h=r;                   /* no of symbols aggregated by transform.*/

  while (I[0] >= -n) {   /* while not single combined sorted group.*/
    pi=I;                /* pi is first position of group.*/
    sl=0;                /* sl is negated length of sorted groups.*/
    do {
      if ((s=*pi) < 0) {
        pi-=s;           /* skip over sorted group.*/
        sl+=s;           /* add negated length to sl.*/
      } else {
        if (sl) {
          *(pi+sl)=sl;   /* combine sorted groups before pi.*/
          sl=0;
        }
        pk=I+V[s]+1;     /* pk-1 is end of unsorted group.*/
        sort_split(pi, pk-pi);
        pi=pk;           /* next group.*/
      }
    } while (pi <= I+n);
    if (sl)              /* if the array ends with a sorted group.*/
      *(pi+sl)=sl;       /* combine sorted groups at end of I.*/
    h=2*h;               /* double sorted-depth.*/
  }
  for (i=0; i<=n; ++i)   /* reconstruct suffix array from inverse.*/
    I[V[i]]=i;
}
```

Fig. 4. Function *suffixsort*. Parameters x and p should point to integer arrays with $n+1$ elements each, where the first n elements of the x array hold a representation of the input string as non-negative integers in the range $[l, k)$. On return, p points to the sorted suffix array and x to its inverse permutation. Functions *transform* and *bucketsort* implement operations described in Section 5.3 and Section 5.4. Function *sort_split* is shown in Fig. 5. V, I, h, and r are global variables in the program.

Burrows–Wheeler transform, this step can be replaced by an analogous one that computes the transformed string instead.

Fig. 5 shows the ternary-split Quicksort routine. The implementation is directly based on Program 7 of Bentley and McIlroy [2] with two exceptions: the sorting method for the smallest subarrays, and the incorporation of group updates.

Group updates are handled in the last section of the routine, between the recursive calls, as explained in Section 5.2. This is implemented as a separate function, shown in Fig. 6.

For fast handling of very small subarrays, we use a non-recursive sorting routine for subarrays with less than 7 elements, implemented as a separate function. Since group updating is difficult in insertion sorting – the common algorithm to use in this situation – we use a variant of selection sorting that picks out one new group at a time, left-to-right, by repeatedly finding all elements with the smallest key value and moving them to the beginning of the subarray. This is easily combined with group updating.

6.2. Experimental results

We show suffix sorting time for various text files. We use a DELL PowerEdge SC1420 workstation (Xeon 3.4 GHz CPU and 8 GB memory) running Solaris 2.9. The programs were compiled with the Gnu C compiler

```
static void sort_split(int *p, int n)
{
   int *pa, *pb, *pc, *pd, *pl, *pm, *pn;
   int f, v, s, t, tmp;
#  define SWAP(p, q)            (tmp=*(p), *(p)=*(q), *(q)=tmp)

   if (n<7) {
      select_sort(p, n); /* special sorting for smallest arrays.*/
      return;
   }
   v=choose_pivot(p, n);
   pa=pb=p; pc=pd=p+n-1;
   while (1) {
      while (pb<=pc && (f=KEY(pb))<=v) {
         if (f==v) { SWAP(pa, pb); ++pa; }
         ++pb;
      }
      while (pc>=pb && (f=KEY(pc))>=v) {
         if (f==v) { SWAP(pc, pd); --pd; }
         --pc;
      }
      if (pb>pc) break;
      SWAP(pb, pc); ++pb; --pc;
   }
   pn=p+n;
   if ((s=pa-p)>(t=pb-pa)) s=t;
   for (pl=p, pm=pb-s; s; --s, ++pl, ++pm) SWAP(pl, pm);
   if ((s=pd-pc)>(t=pn-pd-1)) s=t;
   for (pl=pb, pm=pn-s; s; --s, ++pl, ++pm) SWAP(pl, pm);

   s=pb-pa; t=pd-pc;
   if (s>0) sort_split(p, s);
   update_group(p+s, p+n-t-1);
   if (t>0) sort_split(p+n-t, t);
}
```

Fig. 5. Function *sort_split*, an adaptation of ternary-split Quicksort of Bentley and McIlroy [2, Program 7] with group updates incorporated. Parameters are a pointer to the beginning of subarray to be sorted, and the number of elements in the subarray. Function *select_sort* is an alternative sorting function, faster for small subarrays. Function *update_group* is shown in Fig. 6.

```
static void update_group(int *pl, int *pm)
{
   int g=pm-I;          /* group number.*/
   V[*pl]=g;            /* update group number of first position.*/
   if (pl==pm) *pl=-1; /* one element, sorted group.*/
   else do              /* more than one element, unsorted group.*/
      V[*(++pl)]=g;     /* update group numbers.*/
   while (pl<pm);
}
```

Fig. 6. Function *update_group*. Called to assert that a subarray of I previously part of an unsorted group should constitute a group of its own. Parameters are pointers to the first and last position of the subarray.

version 3.4.3, with option *–O3* for maximum optimization and *–m64* to use 64-bit addressing. The reported times are user times, measured with the *ftime* function.

For example input, we use a set of large files, listed in Table 1. The files are obtained from the *Pizza&Chile Corpus*.[1] Note that the file *ENGLISH* is truncated to 500 MB and 768 MB so that our algorithm can handle it. We also use repetitive files *aaaa* ,*abab* and *rand-rep-k* with length 2 MB to show the robustness of our algorithm. The table shows the size and average and maximum LCP length for each file. The LCP between two strings is the length of the longest common prefix between them, and the average (maximum) LCP for a string is the average (maximum) LCP

[1] http://pizzachili.di.unipi.it.

Table 1
Input data set used for algorithm comparison

File	Contents	Size	σ	LCP (avg, max)	
SOURCES	C/Java source code.	210 866 607	230	371.80	307871
PITCHES	MIDI Pitch values.	55 832 855	133	262.00	25178
PROTEINS	Protein sequences.	66 804 271	24	33.46	6380
DNA	DNA sequences.	403 927 746	16	2420.73	1378596
ENGLISH-1	English texts.	524 288 000	226	6675.21	987770
ENGLISH-2	English texts.	805 306 368	235	14589.35	1095313
XML	XML files.	296 135 874	97	44.91	1084
aaaa	Sequence of 'aaa...'	2 097 152	1	1048575	2097151
abab	Sequence of 'abab...'	2 097 152	2	1048574	2097150
rand-rep-k	Repetition of length-k random string	2 097 152	k	$\approx 2^{20}$	$\approx 2^{21}$

LCP (avg, max) are the average and maximum lengths of the longest common prefix for adjacent suffixes in lexicographic order for each file, respectively. The alphabet size (number of distinct characters) is denoted by σ. The alphabet size of *DNA* is not 4 because it contains some special characters.

Table 2
Algorithm implementations participating in the comparison

Program	Algorithm	Time	Space (bytes)
m05	The MSufSort algorithm [21]. Modified by us to compute the suffix array. http://www.michael-maniscalco.com/msufsort.htm	$O(n^2 \log n)$	$< 6n$
bese	The string sorting algorithm of Bentley and Sedgewick (see Section 2.3 with an initial bucket sorting step. Implementation by Kurtz.	$O(n^2)$	$5n$
ds	The deep-shallow algorithm [23]. http://www.mfn.unipmn.it/~	$O(n^2 \log n)$	$5.01n$
dc32	The difference-cover algorithm [4]. http://www.stefan-burkhardt.net/CODE/cpm_03.tar.gz	$O(n \log n)$	$5.88n$
skew	The skew algorithm [15]. http://www.mpi-sb.mpg.de/~	$O(n)$	$> 10n$
mcil	McIlroy's suffix sorting implementation using a variant of the Manber–Myers algorithm [24]. http://cm.bell-labs.com/cm/cs/who/doug/source.html	$O(n \log n)$	$8n$
LS	Our algorithm with compacted input alphabet (see Section 5.3). http://www.larsson.dogma.net/research.html	$O(n \log n)$	$8n$

for all pairs of adjacent suffixes in the suffix array. These values give a good estimate of the repetitiveness of the files. Maximum LCP is equivalent to the length of the longest repeated substring.

The programs included in the comparison are listed in Table 2. The *m05* program is the MSufSort algorithm [21] version 2.2. The space and time complexities are analyzed by Puglisi et al. [25]. Note that the original program of *m05* does not compute the suffix array; instead it computes the inverse of the suffix array and Burrows–Wheeler transform. We changed it so that after computing the inverse suffix array it computes the true suffix array by a simple linear time in-place inversion algorithm for a permutation. The *bese* program was kindly supplied by Stefan Kurtz of Bielefeld University. The *mcil* program is the implementation by McIlroy [24], referred to in Section 2.1 and Section 5.3. The algorithm is a variant of the Manber–Myers algorithm [20], with improvements that causes it to outperform a direct implementation of that algorithm. In McIlroy's file *ssarray.c*, the suffix sorting routine contains error checks and calculation of parameters that we regard as inputs. These computations, which would lead to unjustly large execution times for this program, have been removed from the code used in our experiments. The *ds* program is the Deep-Shallow algorithm [23], which is said to be the fastest for many inputs. The *dc32* program is the difference-cover algorithm [4] with difference-cover modulo 32. This program is compiled with 32-bit mode, otherwise the working space is doubled. The *skew* program is the skew algorithm [15] which has worst-case linear time complexity and which is the fastest among the linear time suffix sorting algorithms [26]. The *LS* program is our algorithm.

Table 3
Sorting times in seconds

File	*m05*	*bese*	*ds*	*dc32*	*Skew*	*mcil*	*LS*
SOURCES	71.01	447.70	104.19	291.97	815.79	2484.59	226.64
PITCHES	15.32	68.67	16.79	43.87	170.22	367.67	37.14
PROTEINS	22.96	36.80	34.20	75.70	262.68	587.22	51.69
DNA	1030.19	3842.78	269.20	813.32	—	7029.11	616.16
ENGLISH-1	231.11	12990.07	463.81	1011.30	—	8734.19	946.94
ENGLISH-2	394.70	—	860.08	—	—	14240.26	1840.98
XML	111.46	369.68	165.59	648.57	—	2020.25	330.12
aaaa	0.10	—	0.04	5.59	0.51	1.37	0.46
abab	0.10	—	2044.02	1.49	0.58	1.51	0.57
rand-rep-4	0.07	—	1002.25	1.89	0.72	2.28	0.84
rand-rep-8	0.09	—	837.64	2.06	0.98	3.95	1.36
rand-rep-16	0.17	—	406.66	2.35	1.64	7.32	2.63
rand-rep-32	0.33	—	244.82	2.89	3.14	12.04	4.15

Table 3 shows sorting time of the algorithms. The *skew* could not process the files *DNA*, *ENGLISH* and *XML* because of lack of memory. The *bese* could not process ENGLISH-2, *aaaa*, *abab* and *rand-rep-k* because of lack of memory for stack. The *dc32* could not process *ENGLISH-2* because the size of the suffix array exceeds 2GB.

From the values of the table, *m05* appears predominantly to be the fastest, and *ds* the second fastest program. Note, however, that the worst-case time complexity bound of both of these algorithms is $O(n^2 \log n)$ [25], and the table does indicate their inferior behaviour for a few of the inputs. For the DNA input, *m05* is significantly slower than several of the other programs, including our *LS*. Although the *m05* algorithm avoids many degeneration cases by detecting repetition, it does not completely neutralize the effect of the irregular repetitive patterns of that file. The *bese* program is faster than *LS* for strings with small LCP such as *PROTEINS*, but the performance declines for large LCP files. For *abab*, it degenerates catastrophically to quadratic time, and the speed also decreases for the *rand-rep-k* files.

It is interesting to note that *mcil* is slower than the *LS* programs for all inputs, even though *mcil* implements the Manber–Myers algorithm which is also specialized for suffix sorting and has the same worst-case time complexity as our algorithm.

Ours is faster than the *dc32* program, though it uses more space. The running time of *dc32* could not improve even if more space was used. The *skew* program is the only one with better worst-case time complexity than our algorithm. However its actual performance is worse than ours.

To sum up, our algorithm is the most time- and space-efficient among all algorithms with worst-case $O(n \log n)$ time complexity, and more stable than algorithms which are fast for typical-case inputs.

7. Conclusion

We have proposed a fast and memory-efficient suffix sorting algorithm. Among all suffix sorting algorithms with worst-case $O(n \log n)$ time complexity, ours is the fastest and the most space-efficient. Though there exist faster algorithms than ours for many inputs, they do not have good worst-case time complexity, with the result that their performance will decrease for some inputs. Future works will be to develop algorithms which have good worst-case time complexity, which are fast in practice, and which use little memory.

Acknowledgement

The second author's work was supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

References

[1] Bernhard Balkenhol, Stefan Kurtz, Yuri M. Shtarkov, Modifications of the burrows and wheeler data compression algorithm, in: Proceedings of the IEEE Data Compression Conference, 1999, pp. 188–197.
[2] Jon L. Bentley, M. Douglas McIlroy, Engineering a sort function, Software — Practice and Experience 23 (11) (1993) 1249–1265.
[3] Jon L. Bentley, Robert Sedgewick, Fast algorithms for sorting and searching strings, in: Proceedings of the eighth Annual ACM–SIAM Symposium on Discrete Algorithms, 1997, pp. 360–369.

[4] S. Burkhardt, J. Kärkkäinen, Fast lightweight suffix array construction and checking, in: Proc. CPM, in: LNCS, vol. 2676, 2003, pp. 55–69.
[5] Michael Burrows, David J. Wheeler, A block-sorting lossless data compression algorithm, Research Report. 124, Digital Systems Research Center, Palo Alto, California, May 1994.
[6] M. Farach, Optimal Suffix Tree Construction with Large Alphabets, in: 38th IEEE Symp. on Foundations of Computer Science, 1997, pp. 137–143.
[7] P. Ferragina, G. Manzini, Optimal compression boosting in optimal linear time using the Burrows–Wheeler transform, in: Proceedings of the Fifteenth Annual ACM–SIAM Symposium on Discrete Algorithms, 2004.
[8] P. Ferragina, G. Manzini, Indexing compressed texts, Journal of the ACM 52 (4) (2005) 552–581.
[9] Gaston H. Gonnet, Ricardo A. Baeza-Yates, Handbook of Algorithms and Data Structures, Addison-Wesley, 1991.
[10] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM Journal on Computing 35 (2) (2005) 378–407.
[11] C.A.R. Hoare, Quicksort, Computer Journal 5 (1962) 10–15.
[12] W.K. Hon, T.W. Lam, K. Sadakane, W.K. Sung, Constructing compressed suffix arrays with large alphabets, in: Proc. of ISAAC, in: LNCS, vol. 2906, 2003, pp. 240–249.
[13] W.K. Hon, K. Sadakane, W.K. Sung, Breaking a time-and-space barrier in constructing full-text indices, Proc. IEEE FOCS (2003) 251–260.
[14] H. Itoh, H. Tanaka, An Efficient Method for in Memory Construction of Suffix Arrays, in: Proc. IEEE String Processing and Information Retrieval Symposium (SPIRE'99), 1999, pp. 81–88.
[15] J. Kärkkäinen, P. Sanders, S. Burkhardt, Simple linear work suffix array construction, Journal of the ACM 53 (6) (2006) 918–936.
[16] Richard M. Karp, Raymond E. Miller, Arnold L. Rosenberg, Rapid identification of repeated patterns in strings, trees and arrays, in: Proceedings of the 5th Annual IEEE Symposium on Foundations of Computer Science, 1972, pp. 125–136.
[17] Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, second ed., Prentice Hall, 1988.
[18] D. Kim, J. Sim, H. Park, K. Park, Linear-Time construction of suffix arrays, in: Proc. CPM, in: LNCS, vol. 2676, 2003, pp. 186–199.
[19] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, in: Proc. CPM, in: LNCS, vol. 2676, 2003, pp. 200–210.
[20] Udi Manber, Gene Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.
[21] M. Maniscalco, MSufSort, 2005, http://www.michael-maniscalco.com/ msufsort.htm.
[22] G. Manzini, An analysis of the Burrows–Wheeler transform, Journal of the ACM 48 (3) (2001) 407–430.
[23] G. Manzini, P. Ferragina, Engineering a lightweight suffix array construction algorithm, Algorithmica 40 (1) (2004) 33–50.
[24] Peter M. McIlroy, M. Douglas McIlroy, ssarray.c, Source Code, 1997, http://www.cs.dartmouth.edu/~doug/sarray/.
[25] S. Puglisi, W. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, in: Proc. Prague Stringology Conference, 2005.
[26] S. Puglisi, W. Smyth, A. Turpin, The performance of linear time suffix sorting algorithms (extended abstract), in: Proc. DCC, IEEE, 2005, pp. 358–367.
[27] Kunihiko Sadakane, A fast algorithm for making suffix arrays and for Burrows–Wheeler transformation, in: Proceedings of the IEEE Data Compression Conference, 1998, pp. 129–138.
[28] M. Schindler, szip homepage, 1998, http://www.compressconsult.com/szip/.
[29] A. Schönhage, M. Paterson, N. Pippenger, Finding the median, Journal of Computer and System Sciences 13 (2) (1976) 184–199.
[30] J. Seward, bzip2, 1996, http://www.bzip.org/.
[31] Peter Weiner, Linear pattern matching algorithms, in: Proceedings of the 14th Annual IEEE Symposium on Foundations of Computer Science, 1973, pp. 1–11.