LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)/1–20/(1999)],
Dept of Computer Science, Lund University, Sweden

FASTER SUFFIX SORTING

N. JESPER LARSSON* AND KUNIHIKO SADAKANE†

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 all kinds of input. 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 favourable performance compared to other approaches, and argue that our algorithm is the prime choice for general suffix sorting.

1. INTRODUCTION

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

This has at least two important applications. One is construction of a *suffix array* [11] (also known as *PAT array* [5]), a data structure that supports some of the operations of a *suffix tree* [17], 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 complexities of basic operations closely approach those of the suffix tree.

Another application is in data compression. The *Burrows-Wheeler transform* [4] 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, and with advances in research as well as practical implementations [1, 14, 16], its importance is growing rapidly. 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.

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, 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.

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

* Dept of Computer Science, Lund University, Sweden (jesper@cs.lth.se).
† Dept of Information Science, University of Tokyo, Japan (sada@is.s.u-tokyo.ac.jp).

Manber and Myers [11] presented an elegant radix-sorting based algorithm which takes at most $O(n \log n)$ time. They also suggested augmentations that allowed string matching operations in time bounds close to those of the suffix tree, at the cost of additional space.

Section 2 recapitulates this and other approaches connected with our algorithm. Section 3 presents the basic version of our algorithm. (A preliminary version of this algorithm and some of its refinements has previously been presented by Sadakane [13].) 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 $0 \le i \le n$, 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 $[0, n]$, where i represents S_i. 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 $i \in [1, n]$. 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.

Some previous work on suffix sorting also treat calculation of *longest common prefix* (LCP) information, within the time bounds of the algorithm. We conjecture that this can be efficiently computed as a byproduct of our algorithm as well, but do not consider it further, for the following reasons: The LCP array, as well as other augmentations that allow faster access in the suffix array, increase space requirements to the extent that a compact suffix tree implementation, such as those described by Kurtz [10], would often be a better alternative. Furthermore, LCP information is unnecessary for many applications. It is, for example, of no use in implementing the Burrows-Wheeler transform. Lastly, a linear time LCP calculation algorithm is given by Kasai, Arimura, and Arikawa [8], surpassing our sorting bound as well as previous ones.

Alphabet Size Considerations. Much confusion concerning the time complexity of suffix sorting originates from insufficient consideration of the input alphabet size.

It is well known that general sorting with only pairwise comparisons has time complexity $\Theta(n \log n)$, matching the worst case complexity of the Manber-Myers algorithm as well as ours. However, when the input consists of integers in a restricted range, radix techniques may be used. Indeed, the

Manber-Myers algorithm is radix based, and requires that the input consists of integers bounded by n. To lift this restriction, the algorithm must be preceded by a transform comprising symbol sorting. Our algorithm does not require this augmentation.

Assuming that either the input alphabet size is constant, or that constant-time retrieval is possible through hashing, makes linear-time suffix sorting possible. However, in all work known to us, this is accomplished only by taking the detour over suffix tree construction.

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.** One obvious idea for a suffix sorting algorithm 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 proper suffix of the whole string is also a suffix of *another* suffix.

The key for reducing the number of passes is a doubling technique, originating from Karp, Miller, and Rosenberg [7], 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 its primary key, and the position of $S_i + h$ in the same order as its secondary key, yields the $2h$-order.*

To use this observation, we first sort the suffixes according 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 [11] use 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, the authors 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 [12]. Some properties of McIlroy's implementation are discussed in Section 5.3.

2.2. **Ternary-Split Quicksort.** The well known Quicksort algorithm [6] recursively partitions an array into two parts, one with smaller elements than a *pivot element* and one with larger elements. Then the parts are processed 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.

This approach is analyzed and implemented by Bentley and McIlroy [2]. The comparison-based sorting 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 with partitioning starting from 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 implementation *Bzip2* [16].

Our proposed algorithm does not explicitly make use of this string sorting method, but the techniques are related. This is apparent from our time complexity analysis in Section 4: Bentley and Sedgewick consider 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 in suffix sorting, the the final positions of most of the suffixes are determined by only the first few symbols of each suffix. This is true for common real-life data (see Section 6.2) as well as random strings. As a result, a specialized suffix sorting method, such as the Manber-Myers algorithm, is often 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 unnecessary scanning and idle reorganizing of already sorted suffixes. 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 [13] 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 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 in positions corresponding to their leftmost elements. To distinguish between these, we store positive numbers for unsorted groups and negative numbers (the negated length) for combined sorted. 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 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.*

1. Place the suffixes, represented by the numbers $0, \dots, n$, 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 $I[\,f \dots g\,]$, 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.

FIGURE 1. The basic version of our proposed algorithm.

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.

Figure 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. 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.

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 $n-h+1, \dots, n$ have length at most h, and the \$ symbol is therefore included in the considered part 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.

Figure 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 ($I[\,1 \dots 2\,]$), 4 ($I[\,3 \dots 4\,]$), 9 ($I[\,6 \dots 9\,]$) and 13 ($I[\,11 \dots 13\,]$) 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

h															
	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
h	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	7	0		2	10	12	2		7	12	7
	$I[\,i\,]$		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

Figure 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.

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 obtain 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 Figure 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 asymptotically 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 shows that a worst case bound of $O(n \log n)$ is possible.

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 [15], as part of the partitioning routine. In practice, this is rarely desirable, due to increased constant factors, and hardly necessary. There exists a range of pivot-choice methods which balances guaranteed worst-case versus expected performance [2].

For simplicity, we assume in the following analysis that the same method is used for the initial sorting in Step 1 as in later passes. Employing a different sorting algorithm for initial sorting (considered in Section 5) may improve 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: 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 nonempty 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. Figure 3 shows an example ternary tree that corresponds to the same input and sorting process as Figure 2.

The following lemma bounds the height of the ternary tree:

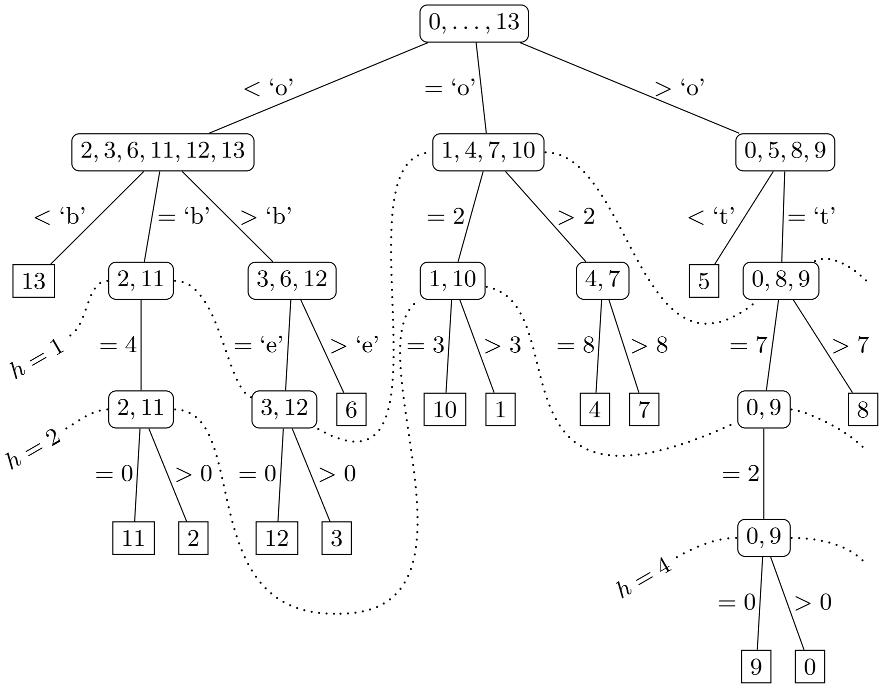
Lemma 5. *The length of a path from the root to any leaf in the ternary tree is at most* $2 \log n + 3$.

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.



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	\$

FIGURE 3. An implicit ternary tree that emerges from sorting with the input string 'tobeornottobe'. The sorting process corresponds to that of Figure 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 mark transitions between sorting passes of the algorithm.

Lemma 6. *Partitioning operations corresponding to all the nodes of any given depth of the tree takes 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 Figure 1 can be done in $O(n \log n)$ worst case time.*

Proof. Partitioning asymptotically dominates sorting time; splitting and combining groups is 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 in 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 Figure 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, and can be directly obtained from V.

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. 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 Figure 1 scans the processed parts twice, in order to update the information in V and L. This is true both for the initial sorting step 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 concatenating adjacent sorted groups, to obtain the maximal combined sorted, 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 other updating of group numbers and lengths can be incorporated in 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 in 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 involves *decreasing* group numbers – this keeps the sorting keys consistent.

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 effect the worst case time complexity, it causes a nontrivial 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 nonnegative 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, which to some degree all originate from McIlroy [12]:

- 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 suffixes is maintained.

McIlroy's implementation requires an alphabet transformation that represents the unique \$ symbol with 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 to 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 [11, 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 alphabet 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 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 later 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 can not be readily renumbered, we can not 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 can not 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 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 [9]. Since the details for implementation of alphabet transformation (described in Section 5.3) and bucket sorting (described in Section 5.4) are 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*, which can be obtained from *www.cs.lth.se/Research/Algorithms/Source/* .

The main suffix sorting routine is shown in Figure 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 two paragraphs of Section 5.3.

The suffixsort function first sets global variables that allow the arrays to be accessed by other functions, then enters the alphabet transformation and initial sorting phase.

The *transform* function called in this phase 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 $K^r \leq q$, K is the smallest power of two such that $K > k - l$, and q is the last parameter in the call to *transform*. The value of r is kept as a global variable.

```
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;
}
```

FIGURE 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 nonnegative 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 Figure 5. V, I, h, and r are global variables in the program.

The transformed alphabet is $\{0,\ldots,j-1\}$ for some alphabet size $j \le q+1$, 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

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

   if (n<7) {
      select_sort_split(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);
}
```

FIGURE 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 a subarray and the number of elements. Function *select_sort_split* is an alternative sorting function, used for small subarrays. Function *update_group* is shown in Figure 6. Function *choose_pivot* returns the key for one element in the subarray.

```
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);
}
```

FIGURE 6. Function *update_group*. Asserts that a subarray of I that was 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.

circumstances *compacts* the alphabet after symbol aggregation, so that all integers less than j occur at least once in V.)

We adapt the use of *transform* to the sizes of the input and the input alphabet. If n is large enough for I to be used as a bucket array for the given alphabet range, i.e., if $n \geq k - l$, we call transform with the q parameter set to n. This guarantees that bucketing is still possible for the transformed alphabet. We then use bucket sorting for initialization of I through a call to a separate function *bucketsort*.

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 the call to sort_split, 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 scheme 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 Burrows-Wheeler transformation, this step can be replaced by an analogous one that computes the transformed string instead.

Figure 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. Choice of pivot element is in a separate function, choose_pivot, for flexibility. Our implementation uses the same *ninther* strategy as Bentley and McIlroy. Other possibilities are, for instance, using the true median (as we assumed for guaranteed worst case performance in Section 4) or a random choice.

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 Figure 6.

For fast handling of very small subarrays, we use a nonrecursive 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

file	contents	size
maini	All 1995 articles of the Japanese newspaper *Mainichi*.	109 442 894
patent	A collection of Japanese patent claims.	89 229 120
reuters	The Reuters corpus.	27 636 766
html	A collection of html files from servers in Japan.	125 595 037
calg	Concatenation of the original Calgary corpus files except *pic* (13 files).	2 628 406
cant	Concatenation of the Canterbury corpus files except *ptt5*.	2 297 568
pic	A Calgary corpus file (the same as *ptt5* of the Canterbury Corpus).	513 216
ecoli	The file E.coli of the large Canterbury corpus.	4 638 690
bible	The file bible.txt of the large Canterbury corpus.	4 047 392
world	The file world192.txt of the large Canterbury corpus.	2 473 400
aaaa64k	The letter ‘a’ repeated 64×1024 times.	65 536
aaaa2M	The letter ‘a’ repeated two million times.	2 000 000
—2M	The first two million bytes of the corresponding file.	2 000 000
—8M	The first 8191 kB of the corresponding file.	8 387 584

TABLE 1. Input data set used for algorithm comparison.

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 report suffix sorting time for various inputs. We use a SUN Ultra 60 workstation (UltraSPARC-II 360 MHz CPU and 2 GB primary storage) running Solaris 2.6. The programs were compiled with the Gnu C compiler version 2.7.2.3, with option *–O3* for maximum optimization. The reported times are user times, measured with the *rusage* command.

As example input, we use a set of large files, listed in Table 1. The files are chosen to demonstrate the behaviour of the programs for different kinds of natural data as well as degeneration cases.

The programs included in the comparison are listed in Table 2. The *htr2ar*, *tr2ar*, and *bese* programs were kindly supplied by Stefan Kurtz of Bielefeld University. The first two of these use suffix trees implemented using Kurtz's space reduction techniques [10]. The *htr2ar* code originates from an application with limited input size; it is unable to handle our largest input files.

The *mcil* program is the implementation by McIlroy [12], referred to in Section 2.1 and Section 5.3. It uses a variant of the Manber-Myers algorithm [11], with improvements that yield better performance than a direct implementation of that algorithm. McIlroy's original implementation contains error checks and calculation of parameters that we regard as inputs. These computations, which would lead to unjustly large execution times, have been removed in our experiments. Because of McIlroy's input requirements, the same input alphabet computation as for *qss2* is incorporated.

program	algorithm
htr2ar	Kurtz's suffix tree implementation with hash table representation (IHTI).
tr2ar	Kurtz's suffix tree implementation with linked list representation (ILLI).
mcil	McIlroy's suffix sorting implementation using an improved version of the Manber-Myers algorithm.
bese	The string sorting algorithm of Bentley and Sedgewick (see Section 2.3) with an initial bucket sorting step. Implementation by Kurtz.
qss0	Our algorithm with input alphabet size 256.
qss1	Our algorithm with input alphabet limits k and l set according to the input (see Section 5.3).
qss2	Our algorithm with compacted input alphabet (see Section 5.3).

TABLE 2. Algorithm implementations participating in the comparison.

file	LCP (avg,	max)	*htr2ar*	*tr2ar*	*mcil*	*bese*	*qss0*	*qss1*	*qss2*
cant	9.0	738	8.4	15.7	24.1	**3.7**	4.0	4.0	4.2
bible	14.0	551	20.4	13.8	72.6	**9.1**	12.0	10.7	10.7
calg	14.6	1 706	12.5	11.8	43.2	**5.0**	5.7	5.7	5.8
ecoli	17.4	2 815	29.2	17.6	101.1	**8.5**	17.3	13.5	9.8
maini	20.1	5 918	—	1 109.2	5 499.9	**415.8**	537.1	539.4	536.7
world	23.0	559	11.1	7.6	39.1	8.0	6.7	**6.0**	6.1
patent	41.4	8 923	—	545.7	3 663.7	398.6	**390.1**	385.9	392.2
reuters	50.9	4 975	—	120.3	713.4	161.6	115.0	103.6	**103.3**
html	606.4	99 125	—	953.2	6 450.5	3 521.3	**585.0**	586.1	585.9
pic	2 353.4	36 316	1.6	**0.8**	3.3	53.3	0.9	0.9	0.9

TABLE 3. Sorting times in seconds. Average and maximum LCP, *longest common prefix* for adjacent suffixes in sorted order, is listed at the left for each file. Lowest time is in bold face.

file	LCP (avg,	max)	*htr2ar*	*tr2ar*	*mcil*	*bese*	*qss0*	*qss1*	*qss2*
maini8M	19.3	4 701	40.2	50.9	205.4	**21.3**	24.0	23.8	21.4
patent8M	38.1	2 027	39.9	33.9	160.6	29.5	**25.5**	25.6	26.1
reuters8M	50.3	4 967	36.7	31.0	199.0	41.7	29.0	**25.9**	26.5
html8M	849.6	73 344	38.8	40.7	238.2	301.6	25.4	**25.4**	25.9
cant2M	8.3	228	7.4	15.3	14.6	**3.1**	3.2	3.2	3.3
maini2M	10.0	1 032	9.3	10.7	33.0	**3.5**	4.1	4.1	4.2
calg2M	11.0	1 029	9.9	9.0	32.4	**3.6**	4.3	4.3	4.4
ecoli2M	12.9	1 345	11.7	7.1	34.2	**3.1**	6.0	4.7	3.5
bible2M	14.7	551	9.4	6.3	30.6	**4.1**	5.0	4.5	4.4
world2M	22.9	559	8.8	6.3	30.2	6.5	5.1	**4.7**	4.8
patent2M	31.6	1 439	9.2	7.0	29.6	5.4	4.5	**4.5**	4.6
reuters2M	47.1	4 967	8.6	6.3	36.4	8.1	5.0	**4.6**	4.7
html2M	252.1	27 110	9.0	8.9	36.9	21.0	4.0	**4.0**	4.1
aaaa2M	999 999.5	1 999 999	4.4	**1.8**	11.4	—	5.8	5.1	5.2
aaaa64k	32 767.5	65 535	0.1	**0.1**	0.2	92.8	0.1	0.1	0.1

TABLE 4. Sorting times reported analogously to Table 3, but with input files truncated to equal lengths.

Table 3 and Table 4 show sorting time of the algorithms, listed with average and maximum LCP length for each file, which gives a good estimate of the repetitiveness of the files. The files are listed in order of average LCP. (Maximum LCP is equivalent to the longest repeated string.) Table 3 lists the results for the full sized natural data files; Table 4 lists results for generated and truncated files of equal length, which give normalized timing results.

The tables show that the simple, non-specialized, string sorting implementation *bese* is the fastest when average LCP is small, but not much faster than the *qss* programs that implement our algorithm. When repeated strings are longer, the *qss* programs are more efficient, and for extremely repetitive input, the suffix tree implementations have an advantage. For the most repetitive files, *bese* degenerates to quadratic time complexity. Since the *bese* program can not handle the *aaaa2M* file, we include the smaller file *aaaa64k* to illustrate the extremely poor behaviour of *bese* for this kind of data.

It is interesting to note that *mcil* is slower than the *qss* programs for all input, 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. Indeed, these experiments indicate that the Manber-Myers algorithm performs very badly for large files, even for natural, non-degenerate, input data. When maximum LCP is large, *mcil* becomes slow, since the number of passes in this algorithm is the logarithm of maximum LCP length, and each pass has to process the full input string. In our algorithm, the speed is not much influenced by maximum LCP, because in later passes most suffixes are already sorted and skipped.

Note that the difference between *qss* and *mcil* is fairly small for *aaaa2M*, whose average and maximum LCP are both large, which causes the unsorted parts to shrink slowly. For *ecoli* on the other hand, the difference between these algorithms is large, since average LCP is small but maximum LCP is large.

Although *htr2ar* is the only program that uses an algorithm with linear worst case performance, it is not the fastest for any of the inputs. The other suffix tree implementation, *tr2ar*, uses linked lists for storing edges, which means that the input alphabet is a factor in its time complexity. This program is slightly faster than those using our algorithm for the most repetitive natural data file *pic*, and the fastest without comparison for the generated file *aaaa2M*, whose input alphabet size is one.

Input alphabet compaction clearly helps when the input alphabet is small. This is noticeable particularly for *ecoli*, which is in the four symbol alphabet of DNA sequences, causing *qss2* to be much faster than *qss0* and *qss1*.

Our algorithm is the fastest for files whose average LCP is neither terribly small nor large. Moreover, it exhibits robust behaviour over all the inputs: the difference in speed between our algorithms and the fastest one is small for all files.

7. Conclusion

Although the proposed algorithm is strongly related to that of Manber and Myers – it requires the same amount of space, has the same asymptotic worst case time complexity, and relies on the same suffix ordering observations –

our experiments clearly show that our ideas for how to reduce superfluous processing are successful, and yield a substantially faster algorithm.

Furthermore, we have found our algorithm to outperform suffix tree implementations for natural data, even for very large files, even though suffix trees theoretically have superior asymptotic time complexity. In addition, our algorithm requires less space than a suffix tree.

Finally, our algorithm exhibits an excellent robustness when processing large or repetitive inputs, matched only by suffix trees. Thus, although a general string sorting algorithm optimized for short strings may have a slight advantage for inputs with little repetition, we assert that our algorithm is clearly a better choice in general, since ordinary string sorting degenerates catastrophically for some input distributions.

References

1. Bernhard Balkenhol, Stefan Kurtz, and Yuri M. Shtarkov, *Modifications of the burrows and wheeler data compression algorithm*, Proceedings of the IEEE Data Compression Conference, March 1999, pp. 188–197.
2. Jon L. Bentley and M. Douglas McIlroy, *Engineering a sort function*, Software – Practice and Experience **23** (1993), no. 11, 1249–1265.
3. Jon L. Bentley and Robert Sedgewick, *Fast algorithms for sorting and searching strings*, Proceedings of the eighth Annual ACM–SIAM Symposium on Discrete Algorithms, January 1997, pp. 360–369.
4. Michael Burrows and David J. Wheeler, *A block-sorting lossless data compression algorithm*, Research Report. 124, Digital Systems Research Center, Palo Alto, California, May 1994.
5. Gaston H. Gonnet and Ricardo A. Baeza-Yates, *Handbook of algorithms and data structures*, Addison-Wesley, 1991.
6. C. A. R. Hoare, *Quicksort*, Computer Journal **5** (1962), 10–15.
7. Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg, *Rapid identification of repeated patterns in strings, trees and arrays*, Proceedings of the 5th Annual IEEE Symposium on Foundations of Computer Science, May 1972, pp. 125–136.
8. Toru Kasai, Hiroki Arimura, and Setsou Arikawa, *Virtual suffix trees: Fast computation of subword frequency using suffix arrays*, Proceedings of the 1999 Winter LA Symposium, February 1999, in Japanese.
9. Brian W. Kernighan and Dennis M. Ritchie, *The C programming language*, second ed., Prentice Hall, 1988.
10. Stefan Kurtz, *Reducing the space requirement of suffix trees*, Tech. Report 98-03, Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik, 1998.
11. Udi Manber and Gene Myers, *Suffix arrays: A new method for on-line string searches*, SIAM Journal on Computing **22** (1993), no. 5, 935–948.
12. M. Douglas McIlroy, *ssort.c*, Source Code, 1997, http://cm.bell-labs.com/cm/cs/who/doug/source.html.
13. Kunihiko Sadakane, *A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation*, Proceedings of the IEEE Data Compression Conference, March–April 1998, pp. 129–138.
14. Michael Schindler, *bzip2 program*, HTTP site www.compressconsult.com, 1998.
15. A. Schönhage, M. Paterson, and N. Pippenger, *Finding the median*, Journal of Computer and System Sciences **13** (1976), no. 2, 184–199.
16. Julian Seward, *bzip2 program*, HTTP site www.muraroa.demon.co.uk, 1997–1999.
17. Peter Weiner, *Linear pattern matching algorithms*, Proceedings of the 14th Annual IEEE Symposium on Foundations of Computer Science, 1973, pp. 1–11.