An Efficient, Versatile Approach to Suffix Sorting

MICHAEL A. MANISCALCO
Independent Researcher
and
SIMON J. PUGLISI
Curtin University of Technology

Sorting the suffixes of a string into lexicographical order is a fundamental task in a number of contexts, most notably lossless compression (Burrows–Wheeler transformation) and text indexing (suffix arrays). Most approaches to suffix sorting produce a sorted array of suffixes directly, continually moving suffixes into their final place in the array until the ordering is complete. In this article, we describe a novel and resource-efficient (time and memory) approach to suffix sorting, which works in a complementary way—by assigning each suffix its rank in the final ordering, before converting to a sorted array, if necessary, once all suffixes are ranked. We layer several powerful extensions on this basic idea and show experimentally that our approach is superior to other leading algorithms in a variety of real-world contexts.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms**]: Nonnumerical Algorithms and Problems; E.1 [**Data Structures**]: Arrays; H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; D.1.0 [**Programming Techniques**]: General

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Suffix sorting, Burrows–Wheeler transform, suffix array, suffix tree

ACM Reference Format:

Maniscalco, M. A. and Puglisi, S. J. 2007. An efficient, versatile approach to suffix sorting. ACM J. Exp. Algor. 12, Article 1.2 (2007), 23 pages DOI 10.1145/1227161.1278374 http://doi.acm.org 10.1145/1227161.1278374

The software tested in “An efficient, versatile approach to suffix sorting,” by Maniscalco and Puglisi, is maintained at the following site: www.michael-maniscalco.com: The MSfuSort Algorithm.

Author’s address: Michael A. Maniscalco, email: michael@michael-maniscalco.com; Simon J. Puglisi, Department of Computing, Curtin University of Technology, GPO Box U1987, Perth 6845, Australia; email: sjp@cs.curtin.edu.au.

1. INTRODUCTION

Suffix sorting is the task of lexicographically ordering all the suffixes of a string and is the computational bottleneck in a number of important applications. Perhaps most notable of these is the construction of the suffix array data structure, proposed by Manber and Myers [1993] as a space-efficient alternative to the suffix tree. When combined with relatively small auxiliary information, the suffix array can provide efficient and often optimal solutions to many problems involving pattern matching and pattern discovery in large strings, such as those arising in computational biology [Abouelhoda et al. 2004]. More recently, suffix arrays have become the basis for a variety of succinct full text indexes [Ferragina and Manzini 2000; Sadakane 2002; Grossi and Vitter 2005; Mäkinen and Navarro 2005]. While these structures are still experimental, the resource cost of suffix-sorting will undoubtedly be a hurdle to their wide-scale adoption.

Another important application of suffix sorting is the Burrows–Wheeler transformation (BWT) [Burrows and Wheeler 1994]. The BWT is a reversible permutation of the characters of a string, which enables very powerful lossless compression, and is the basis for practical compression tools such as *bzip2* [Seward 2004] and *szip* [Schindler 2002].

In a suffix-sorting algorithm, we require the following qualities:

1. *Speed*. Obviously, we would like the suffixes to be sorted as quickly as possible (within certain constraints, such as finite memory). A useful suffix–sorting algorithm should be largely resistant to frequent repeated patterns in the input, which would otherwise necessitate many character inspections.
2. *Small working space*. Small space requirements are important because they reduce the burden of programs on the underlying system and allow larger texts to be treated in memory. Manzini and Ferragina [2004] coin the term "lightweight" to refer to suffix-sorting algorithms with (relatively) small space requirements, less than $6n$ bytes. This space is accounted for by the suffix array ($4n$ bytes), the input string (n bytes for ASCII symbols), and some extra working space ($<n$ bytes).
3. *Sensitivity to alphabet*. One of the factors most often neglected by suffix-sorting algorithms is the size of the alphabet, Σ. Ideally, the algorithm should be independent of Σ, but failing that should show a graceful degradation of performance as Σ grows. Many algorithms assume that $|\Sigma| \leq 256$, including some of the fastest known suffix sorters [Manzini and Ferragina 2004; Schürmann and Stoye 2005], which rely on $|\Sigma|^2$ being a manageable size. While this is fine for the common case that the input is ASCII text, or can be treated as byte-wise data, it is infeasible for many other situations. For example, word-based BWT requires suffix–sorting with alphabets between 20,000 to 100,000 symbols [Moffat and Isal 2005] and Asian newspapers and books commonly contain over 10,000 distinct symbols [Vines and Zobel 1998]. In both these contexts, algorithms containing $|\Sigma|^2$ terms would be rendered unusable.

In this paper, we describe a novel algorithm that addresses the above issues. In fact, we introduce several algorithms, putting a basic approach through several versions to arrive at our best algorithm. Our main idea is to break the suffixes into groups, assign ranks to the suffixes in each group in lexicographical order, and then use these ranks to subsequently speed the assignment of ranks to other suffixes. When the algorithm completes, every suffix has been assigned a unique lexicographic rank, enabling the suffix array or the BWT to be computed.

Extensive experiments in a variety of real world contexts show our approach is most often faster than the previous best known algorithms, even when the input contains many repeated patterns. Further, our algorithms require little more than $4n + zn$ bytes of working space, including space for the suffix array and input string, where z is the number of bytes required per input symbol. Thus, in the common case that $|\Sigma| \leq 256$, our approach requires around $5n$ bytes, similar to the most memory-concise algorithms known. We point out that our algorithms are "practical" ones and throughout we avoid any formal analysis of their asymptotic behavior other than stating now (what we believe is) a loose $\Theta(n^2 \log n)$ bound.

Section 2 sets notation and definitions used throughout and Section 3 reviews previous work. We then introduce our algorithms in Section 4 and describe equally important algorithmic engineering issues in Section 5. Our experiments are described in Section 6 and results presented in Section 7, with conclusions and reflections offered in Section 8.

2. DEFINITIONS AND NOTATION

Let Σ be a *constant*, *indexed* alphabet consisting of symbols σ_j, $j = 1, 2, \ldots, |\Sigma|$ ordered $\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}$. Essentially, we assume that Σ can be treated as a sequence of integers whose range is not too large. We will be usually interested in applications with $|\Sigma| \in 0..255$ and sometimes $|\Sigma| \in 0..65536$, where each symbol requires one or two bytes of storage, respectively. We refer to pair of consecutive symbols as a *bigram*.

Throughout, we consider a finite, nonempty string $x = x[0..n] = x[0]x[1] \ldots x[n]$ of $n + 1$ symbols. The first n symbols of x are drawn from Σ and comprise the actual input. The final character $x[n]$ is a special "end of string" character, \$, defined to be lexicographically smaller than any other character in Σ. We assume that $n < 2^{32}$, implying that an integer in the range $0..n$ can be stored in 4 bytes. For $i = 0, \ldots, n$, we write $x[i..n]$ to denote the suffix of x of length $n - i$, that is, $x[i..n] = x[i]\ x[i + 1] \cdots x[n]$. For simplicity, we will frequently refer to suffix $x[i..n]$ simply as "suffix i."

We are interested in computing the *suffix array* of x which we write SA_x or just SA. The suffix array is an array $\text{SA}[0..n]$, which contains a permutation of the integers $0..n$, such that $x[\text{SA}[0]..n] < x[\text{SA}[1]..n] < \cdots < x[\text{SA}[n]..n]$. Stated another way, $\text{SA}[j] = i$ iff $x[i..n]$ is the jth suffix of x in ascending lexicographical order (*lexorder*). We also define the inverse suffix array $\text{ISA}[0..n]$ such that $\text{ISA}[i] = j$, iff $\text{SA}[j] = i$. Thus, $\text{ISA}[i]$ provides the lexicographic rank of $x[i..n]$ in constant time.

Finally, the Burrows–Wheeler transformation bwt$[0..n]$ of $x[0..n]$ is a string defined by SA. Specifically, bwt$[i] = x[\mathrm{SA}[i] - 1]$, unless $\mathrm{SA}[i] = 0$, in which case bwt$[i] = \$$.

3. EXISTING APPROACHES TO SUFFIX SORTING

Over the past 15 years, many suffix-sorting algorithms have been proposed—a recent survey paper counts a dozen distinct approaches [Puglisi et al. 2006].

In its most basic form suffix-sorting is string sorting. The suffixes are treated as independent strings and sorted using one of the many sort routines tailored for strings [McIlroy et al. 1993; Bentley and Sedgewick 1997; Andersson and Nilsson 1998; Sinha and Zobel 2004]. The difficulty of suffix-sorting for a string-sorting routine is captured by the *average longest common prefix (LCP)* of the string, defined by [Sadakane 1998] as follows

$$\text{Average LCP} = \frac{1}{n-1}\sum_{i=1}^{n-1} \mathrm{lcp}(\mathrm{SA}[i],\ \mathrm{SA}[i+1])$$

where $\mathrm{lcp}(j, k)$ gives the longest common prefix of suffixes j and k. When the average LCP of a string is small, direct string sorting gives an acceptable performance [Larsson and Sadakane 1999]. However, as the average LCP grows, custom suffix-sorting algorithms that exploit, in some way, the relationships between suffixes can be many times faster.

Since 2003, several algorithms have been available to sort suffixes in $\Theta(n)$ time, independent of Σ [Ko and Aluru 2003; Kärkkäinen and Sanders 2003; Kim et al. 2003]. While these algorithms are important theoretically, their known implementations are not competitive with supralinear algorithms, in practice, being much slower and requiring two to three times more memory [Puglisi et al. 2005]. The remainder of this section highlights salient features of the more prominent, practical algorithms, particularly, those related to our new approach. For more details and other algorithms we refer the reader to Puglisi et al. [2006].

3.1 Two Stage

Algorithm *two stage* [Itoh and Tanaka 1999] uses a counting sort to logically partition the SA space into $|\Sigma|$ groups, with the ith group containing the suffixes having first character σ_i. Each group is further partitioned, the first portion containing type X suffixes and the second containing type Y suffixes. Type X suffixes have prefix $\sigma_j\sigma_k$ such that $\sigma_j > \sigma_k$. Type Y have $\sigma_j \leq \sigma_k$. When the suffixes of a group are in lexorder, type X suffixes always come before type Y suffixes. The key observation Itoh and Tanaka make is that once all the groups of type Y suffixes are sorted, the order of the type X suffixes is implied. More precisely, with the type Y suffixes sorted (with a string sorting algorithm), algorithm *two stage* makes a single pass over SA and for each suffix i encountered, if suffix $i - 1$ is of type X, then it should be moved the current front of its group in SA and the group front is incremented. In this way, the type X suffixes at least are sorted in *O(n)* time.

3.2 Prefix Doubling

The *prefix-doubling* technique was first applied to suffix-sorting by Manber and Myers [1993], inspired by the earlier work of Karp et al. [1972] in string matching. The most efficient implementation is that of Larsson and Sadakane [1999].

Generally, the approach works in rounds—at the beginning of the round h, the suffixes are sorted on their 2^{h-1} prefix in SA_h with corresponding ranks in ISA_h. It is then observed that a sort using the integer pairs $(\mathrm{ISA}_h[i], \mathrm{ISA}_h[i+h])$ as keys, $i+h \leq n$, computes a $2h$-order of the suffixes i (suffixes $i > n-h$ are necessarily already fully ordered).

The two main implementations of the prefix-doubling approach differ primarily in their application of the above observation. Manber and Myers do an implicit $2h$-sort by performing a left-to-right scan of SA_h that induces the $2h$-rank of $\mathrm{SA}_h[j]h$, $j \in 0..n$. On the other hand, Larsson and Sadakane explicitly sort each h-group using the ternary-split quicksort (TSQS) of Bentley and McIlroy [1993]. Both approaches require $8n$ bytes of working space. Prefix-doubling sorters have the advantage of being alphabet independent and taking $O(n \log n)$ time, in the worst case.

3.3 Copy and Variants

Seward [2000] describes an important heuristic algorithm for suffix-sorting called *copy*. The main idea bears a resemblance to *two stage*. Algorithm *copy* initially sorts the suffixes into 1- and 2-groups, based on their first two characters (using a counting sort). 1-Groups refer to contiguous portions of the suffix array, where suffixes share the same first character and 2-groups ("contained" in 1-groups) refer to contiguous portions sharing the same first two characters. Seward sorts the 1-groups in order of smallest to largest (i.e., those containing least suffixes to those containing the most). Let G_λ denote the 1-group whose member suffixes all start with the letter $\lambda \in \Sigma$. When G_λ is completely sorted, by passing back over the portion of SA containing G_λ (now in order) for each suffix i encountered, the order of the suffixes in 2-group prefixed $x[i-1]\lambda$ can be induced. As sorting of 1-groups proceeds, ever more 2-groups will be already ordered, allowing the sort routine to skip those portions of the 1-group. Seward shows how the sorting of 1-groups can be made still more efficient by avoiding the sorting of suffixes in G_λ prefixed $\lambda\lambda$. If such suffixes are left until after the other members of G_λ are sorted, their order can also be induced. This ability of *copy* to deal with long runs of identical characters efficiently gives it a distinct advantage over *two stage*, which has no such mechanism.

It is worth noting that *copy* was intended for use in a character-based BWT setting, where it is assumed $|\Sigma| \leq 2^8$. This assumption keeps the space required for the $|\Sigma|^2$ buckets reasonable. If, however, $|\Sigma| = 2^{16}$, the memory requirements for the algorithm would increase dramatically, making the algorithm impractical, in some applications. This weakness is inherited by algorithms which extend *copy*. Several very fast suffix sorters are based on *copy*, namely, *cache* [Seward 2000], deep-shallow (*ds*) [Manzini and Ferragina 2004], and bucket pointer refinement (*bpr*) [Schürmann and Stoye 2005]. These algorithms all layer techniques on *copy* to help when the sort depth (or average lcp) of a group of suffixes becomes large.

Algorithm *cache* maintains an array $C[0..n-1]$ of integers, initially all 0. When suffix i is known to be in its final place, k, in SA we know that suffix is ranked k among all suffixes and place the most significant 16 bits (or, alternatively, 8 bits) of k in $C[i]$. Later, if we are comparing suffixes j and k having $x[j] = x[k]$, we then compare $C[j+1]$ and $C[k+1]$, and a difference gives us the correct order. If $C[j+1] = C[k+1]$, we then inspect $x[j+1] = x[k+1]$ and if they are equal then we compare $C[j+2] = C[k+2]$, and so on. Of course, the possibility of equal C values arises, because we are only caching the most significant portion of each rank. This truncation of ranks means only an extra $2n$ bytes is required to store C (or $1n$ if only 8 bits are used).

Algorithm *ds* is probably the fastest suffix sorter available for real-world inputs with $\Sigma \leq 256$. The philosophy of *ds* is to treat suffixes having small lcp differently from those having large lcp. The 1-groups are sorted with multikey quicksort (MKQS) [Bentley and Sedgewick 1997] to depth 256 (small lcp) or until the size of the group is smaller than a predefined constant, at which point it completes the sort using a combination of sophisticated heuristics.

Algorithm *bpr* is a very recently proposed variant of *copy* that blends the ideas of *cache* and *prefix doubling* to sort the 1-groups. Runtimes of *bpr* are competitive with *ds* on real-word data and many times faster on highly repetitive inputs. The drawback of *bpr* is that (like other prefix doubling sorters) it requires at least $8n$ bytes of working memory to hold pointers to suffixes and their evolving ranks.

4. SUFFIX SORTING BY COMPUTING LEXICOGRAPHIC RANKS

We introduce several algorithms in this section, developing the simplest scheme into the most complex. All the algorithms build the ISA, which can then be permuted into the SA or used to construct the BWT text, as required.

4.1 Sorting Strings with Buckets

At the heart of all the algorithms is an efficient bucket-sorting regime. Most of the work is done in an array of n integers, which is eventually the ISA, with extra space required for a small stack. The bucket sorting begins by linking together all the suffixes having the same first character to form *chains* of suffixes. For example, the string

$$\begin{array}{lcccccccc} i & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} \\ x[i] & a & a & b & c & b & c & a & \$ \end{array}$$

would result in the creation of the following four chains

$$\begin{array}{cccc} (7) & (6,1,0) & (4,2) & (5,3) \\ \$ & a & b & c \end{array}$$

We define a *u-chain*, C_u, for prefix u, as a set of suffixes such that for all i and $j \in C$, $x[i..i+|u|-1] = x[j..j+|u|-1]$. In other words, i and j are in the same u-chain iff suffixes i and j share u as a common prefix.

If a u-chain contains only one suffix, like the \$-chain above, we say it is a *singleton*.

The space allocated for the ISA provides a way to efficiently manage chains. Instead of storing the chains explicitly as above, we compute the equivalent array

$$\begin{array}{lcccccccc} i & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} \\ x[i] & a & a & b & c & b & c & a & \$ \\ \mathrm{ISA}[i] & \perp & 0 & \perp & \perp & 2 & 3 & 1 & \perp \end{array}$$

In which ISA[i] is the largest $j < i$ such that $x[j..j+1] = x[i..i+1]$ or a special symbol, $\perp$, if no such j exists. In our example, the chain of all the suffixes prefixed with a contains 6, 1, and 0 and so we have ISA[6] $= 1$, ISA[1] $=$ 0 and ISA[0] $= \perp$, marking the end of the chain. Observe that chains are singly linked and are only traversable right-to-left. We keep track of u-chains to be processed by storing a stack of integer pairs (h, ℓ), where h is the head of the chain (its right-most index), and $\ell = |u|$ is the length of the common prefix shared by the suffixes in the chain. Chains always appear on the stack in ascending lexicographical order, according to $x[h..h+|u|-1]$. Thus, for our example, initially $(7, 1)$ for chain \$ is atop the stack and $(5, 1)$ for chain c at the bottom.

Chains are popped from the stack and progressively refined by looking at further characters. So long as we take care to process the chains in lexicographical order, when we pop a singleton chain, the suffix contained has been differentiated from the rest and can be assigned the next lexicographic rank. Elements in the ISA that are ranks are differentiated from elements in chains by setting the sign bit, that is, if ISA[i] < 0, then the rank for suffix i is $-$ISA[i]. The evolution of the ISA over subsequent sorting rounds for our example string is illustrated below.

Initially, we have the ISA arranged as shown above. The first item to be popped from the stack is (7, 1), representing the \$-chain and, because this is a singleton (ISA[7] $= \perp$), we set ISA[7] $= -1$, indicating suffix 7 has lexrank 1. This yields

$$\begin{array}{lcccccccc} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} \\ \mathrm{ISA} & \perp & 0 & \perp & \perp & 2 & 3 & 1 & -1 \end{array}$$

The next item popped is (6,1) representing the a-chain. We follow the links in the chain, inspecting the second character of each suffix encountered. This splits the chain into three subchains for ab, aa, and $a\$$ and so we push $(1, 2)$, $(0, 2)$, and $(6, 2)$ onto the stack (in that order). The ISA now appears

$$\begin{array}{lcccccccc} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} \\ \mathrm{ISA} & \perp & \perp & \perp & \perp & 2 & 3 & \perp & -1 \end{array}$$

The top item on the stack is now $(6, 2)$, which was just pushed; it is a singleton. We set ISA[6] $= -2$, indicating suffix 6 has lexrank 2.

$$\begin{array}{lcccccccc} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} \\ \mathrm{ISA} & \perp & \perp & \perp & \perp & 2 & 3 & -2 & -1 \end{array}$$

```
formInitialChains()
repeat
  $(h, \ell) \leftarrow$ chainStack.pop()
  if ISA[$h$] $= \perp$ then
    ISA[$h$] $\leftarrow$ nextRank()
  else
    while $h \neq \perp$ do
      sym $\leftarrow$ getSymbol($h + \ell$)
      updateSubChain(sym,$h$)
      $h \leftarrow$ ISA[h]
    sortAndPushSubChains()
until chainstack is empty
```

Fig. 1. Suffix bucket sorting.

This process of popping, splitting chains, and assigning ranks continues until the stack is empty, at which point the ISA is complete

	0	1	2	3	4	5	6	7
ISA	-3	-4	-6	-8	-5	-7	-2	-1

A high-level algorithm embodying these ideas is listed in Figure 1. We reiterate here that when a chain is split, the resulting subchains must be placed on the stack in lexicographical order for the correct assignment of ranks to singletons. This is illustrated above when the a-chain is split and the next chain processed is the singleton $a\$$-chain. Alternatively, we could always place chains on the stack in reverse lexorder and assign ranks from n down to 1.

Results in Seward [2000] would suggest that direct comparison bucket sorting alone would not make a competitive suffix sorter. In coming sections, we show how properties inherent to this basic approach can be exploited to accelerate sorting.

4.2 Exploiting Previously Ranked Suffixes

The processing of chains in lexicographical order allows for the possibility to use previously assigned ranks as sort keys for some of the suffixes in a chain. To elucidate this idea, we first need to make a couple of observations about the way chains are processed.

When processing a u-chain with $|u| = \ell$, we can classify suffixes into two types: suffix i is of type A, if the rank for suffix $i + \ell$ is known and is of type B otherwise. We can classify a suffix this way in constant time by virtue of the fact we are building the ISA—we inspect ISA[$i + \ell$] and a checked sign bit indicates a rank. Now consider the following observation:

LEMMA 4.1. *Lexicographically, type* A *suffixes come before type* B *suffixes.*

To use this observation, when we refine a chain, we place only type B suffixes into subchains and place type A suffixes to one side. Now, the order of the m type A suffixes can be determined via a comparison-based sort, using for suffix i the rank of suffix $i + \ell$ as the sort key. Once sorted, the type A suffixes can be assigned the next m ranks. We refer to sorting suffixes this way as *induction sorting* and say that the type A suffixes are "sorted by induction."

```
formInitialChains() **repeat**
  $(h, \ell) \leftarrow$ chainStack.pop()
  **if** ISA$[h] = \perp$ **then**
    ISA$[h] \leftarrow$ nextRank()
  **else**
    **while** ISA$[h] \neq \perp$ **do**
      **if** isRanked$(h + \ell)$ **then**
        noteSuffix($h$,ISA$[h + \ell]$)
      **else**
        sym $\leftarrow$ getSymbol$(h + \ell)$
        updateSubChain(sym,$h$)
      $h \leftarrow$ ISA$[h]$
    pushSubChains()
    rankNotedSuffixes()
**until** chainStack is empty
```

Fig. 2. The way *induction sorting* integrates with *bucket sorting*.

Pseudo code capturing this idea is given in Figure 2. Loosely speaking, as the number of assigned ranks increases, the probability that a suffix can be sorted by induction also increases. In fact, every chain of suffixes with prefix $\sigma_j\sigma_k$, where $\sigma_j < \sigma_k$ will be sorted entirely by induction. Clearly, induction sorting will lead to a significant reduction in work for many texts.

Induction sorting shares something with both the *two-stage* algorithm of Itoh and Tanaka [1999] and *cache* algorithm of Seward [2000].

Induction sorting can be thought of as not only a very space-efficient version of the *cache* scheme, but also as one that allows use of *full* rank information, not just 16 bits. These improvements are possible primarily because we are manipulating the ISA rather than the SA. Another key difference is the lexicographic order in which ranks are assigned and become available for use in induction sorting.

The idea can also be considered a generalization of the *two-stage* algorithm. As noted above, suffixes in a two-chain with common prefix $\sigma_1\sigma_2$ and $\sigma_1 > \sigma_2$ are sorted entirely by induction (like the type X suffixes of *two stage*). However, the lexicographical processing of suffixes means induction sorting is applied to suffixes in u-chains with $|u| > 2$.

Before moving on, we pause to highlight a virtue of the algorithm so far described. When the ith rank is assigned to ISA$[j]$, we know bwt$[i] = x[j-1]$. Because we are assigning ranks in sequence from 1 to n, as the ith rank is assigned, bwt$[i]$ can be sent straight to the encoder, allowing for parallel implementation of the sorter and the encoder. This same idea can be helpful to suffix array construction, in the event that the SA is being stored to disk for later use. If this is the case, the SA can be output to disk in a contiguous manner as each rank is assigned, minimizing costly seeks.

4.3 Further Reducing Comparison Sorting

In this section, we describe a way of reducing the number of suffixes sorted with comparison sorting (using either characters or ranks) to, at most, $n/2$. The idea is to split the suffixes into two sets and sort only the smaller set using methods described in the previous sections, with the order of the suffixes in the larger set induced as a byproduct.

Suffixes are divided as follows: suffix i is of type U if $x[i..n] < x[i+1..n]$ and is of type V if $x[i..n] > x[i+1..n]$. The type of suffix $x[n]$, is undefined. For example

	0	1	2	3	4	5	6	7
x	c	c	b	a	b	a	c	\$
type	V	V	V	U	V	U	V	–

This classification is a natural extension of the one used by Itoh and Tanaka [1999] first described in Ko and Aluru [2003]. Two important qualities of type U/V suffixes are:

1. All suffixes can be so classified with a simple, linear time algorithm: scanning x left to right, if $x[i] < x[i+1]$, suffix then i is type U, and if $x[i] > x[i+1]$ then suffix i is type V. If $x[i] = x[i+1]$, we then continue inspecting characters until we come to some position j, where $x[j] \neq x[i]$; if $x[i] < x[j]$ then suffixes $i, i+1, \ldots, j$ are all type U, otherwise they are type V.
2. If suffix i is type V, suffix j is type U and $x[i] = x[j]$ then $x[i..n] < x[j..n]$.

We will assume for the moment that the type U suffixes are fewer (shortly we will treat the opposite case). Before sorting of the type U suffixes begins, we mark the positions of the type V suffixes by setting ISA$[i] = \perp$, if suffix i is type V—doing this allows us to later decide the type of a suffix in constant time. Also, for each distinct bigram $\alpha\beta \in \Sigma^2$ that is a prefix of a type V suffix, we maintain a separate list $M_{\alpha\beta}$ (initially empty) in which an ordering on the type V suffixes will be developed. Only type U suffixes are linked together in chains. These suffixes are to be sorted with character comparisons and induction-sorting techniques, as previously described. However, now *whenever* we assign a rank to suffix i, if suffix $i-1$ is of type V, we add suffix $i-1$ to the end of list $M_{x[i-1]x[i]}$. Provided we always do this, the following holds

LEMMA 4.2. *When all suffixes prefixed with bigrams lexicographically less than $\alpha\beta$ have been ranked, the list $M_{\alpha\beta}$ contains the type-V suffixes prefixed with $\alpha\beta$ in lexicographical order.*

PROOF. The result follows from the fact that for all suffixes $i, j \in M_{\alpha\beta}$ we have $x[i] = x[j]$ and i only comes before j in $M_{\alpha\beta}$, if $rank(i+1) < rank(j+1)$. □

We process a list $M_{\alpha\beta}$ containing m suffixes by simply assigning the next m ranks to the suffixes in the order they appear in the list. It is also important here, that when we assign a rank to suffix i, if suffix $i-1$ is of type V we add it to list $M_{x[i-1]x[i]}$. The u-chain on top of the stack indicates when M lists should be processed. More precisely, let (h, ℓ) be the element on top of the chain stack. If $\ell = 1$ and $x[h] \neq \sigma_1$, then we process lists $M_{x[h]\sigma_1}, M_{x[h]\sigma_2}, \ldots, M_{x[h]x[h]}$ before continuing processing of the chain. For example, when we pop the c-chain we know to process M_{ca}, M_{cb}, and M_{cc} before refinement of the c-chain itself.

The $M_{\alpha\beta}$ lists can be stored efficiently in the ISA in the same manner that chains are stored, with extra space required only for list heads and tails.

If the U suffixes are more frequent than the V suffixes, we can reverse their roles in the above process to obtain an algorithm that sorts, at most, $n/2$ suffixes by comparison sorting. The major difference in logic when ranking V suffixes instead of U suffixes is that ranks must be assigned from n down to 1. To this end, we have the bucket-sorting process chains in reverse lexorder and modify induction sorting accordingly.

Finally, we remark that while above the number of $M_{\alpha\beta}$ lists is $O(\Sigma^2)$, it was described this way only for ease of exposition and the technique can still be applied when Σ is large. The basic idea is to maintain two lists for each symbol: M_α for V suffixes prefixed $\alpha\beta$, where $\alpha \neq \beta$ and $M_{\alpha\alpha}$ containing V suffixes prefixed $\alpha\alpha$.

4.4 Detecting and Processing Repetitions

We now describe how the right-to-left chains of suffixes formed during bucket sorting enable us to detect *repetitions* in the input string and sort the suffixes involved in them efficiently. Before giving the details of our method, we first define the structure we are interested in.

Definition 4.3. A **repetition** in a string $x[0..n]$ is a substring $x[i..i+rp]$ for integers $r \geq 2, p \geq 1, i \geq 0$ such that $x[i..i+p] = x[i+p+1..i+2p] = \cdots = x[i+(r-1)p+1..i+rp]$ and $x[i..i+p] \neq x[i-p..i-1]$ and $x[i..i+p] \neq x[i+rp+1..i+rp+p]$. We call $u = x[i..i+p]$ the *generator* of the repetition, $p = |u|$ the *period* of the repetition, and r the *exponent*. A repetition located at position i in a string is succinctly represented by the tuple $u^r = (i, p, r)$.

For example, the string $x[0..17] = abcaaaabcdabcdabcdab$ contains the repetitions: $a^4 = (3, 1, 4)$; $abcd^3 = (6, 4, 3)$; $bcda^3 = (7, 4, 3)$; $cdab^2 = (8, 4, 2)$ and $dabc^2(9, 4, 2)$.

Detecting repetitions in strings is a well-studied problem in stringology (see Smyth [2003]). Strings containing many repetitions are generally difficult cases for suffix-sorting algorithms as the average common prefix is very high. Such strings are known to be particularly catastrophic for direct-comparison algorithms, such as *two-stage*, *copy*, *cache* and *ds*, as illustrated by experiments in Kärkkäinen and Burkhardt [2003], Schürmann and Stoye [2005], and Puglisi et al. [2005].

In a u-chain, a repetition is manifest as a maximal sequence of $|u|$ spaced elements. We call such a sequence a *repetition sequence* denoted $S_{i,u}$ of length $|S_{i,u}| = |S|$, where i is the value of the right-most element in the sequence. We will refer to i as the *terminating position* of the repetition sequence and the other elements in $S_{i,u}$ as *nonterminating positions*. Detecting repetitions by looking for these sequences naturally integrates with the processing of suffix chains. Consider the string

$$\begin{array}{cccccccccc} i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ x[i] & b & c & a & a & a & a & c & b & \$ \end{array}$$

After forming the initial chains, we have

$$\begin{array}{ccc} 5,4,3,2 & 7,0 & 6,1 \\ a & b & c \end{array}$$

For each of these chains $|u| = 1$. We process the chain a first and discover the repetition sequence $S_{5,a} = 5, 4, 3, 2$, because each suffix in the chain is spaced $|u|$ positions apart. Once we have detected the repetition $S_{i,u}$, we can sort the suffixes $i, i - |u|, i - 2|u|.., i - |u|(r - 1)$ relative to each other by making use of the following observation.

LEMMA 4.4. *For the repetition sequence $S_{i,u}$, if suffix i can be sorted by induction, then $x[i..n] < x[i - |u|..n] < \cdots < x[i - |u|(|S| - 1)..n]$, otherwise $x[i..n] > x[i - |u|..n] > \cdots > x[i - |u|(|S| - 1)..n]$.*

The correctness of this observation is because of the processing of the u-chains in lexicographically ascending order.

When a u-chain contains more than one repetition sequence, things become trickier to deal with than the example above. In general, a chain will contain a set of repetition sequences, $\mathcal{M}$, where the repetitions all have the same generator (being u because they have been detected in the same u-chain). Suffixes in $\mathcal{M}$ are processed differently, depending on whether the terminating position can be sorted by induction or not—we divide the set into a set $\mathcal{I}$ containing sequences with terminating position sortable by induction and set $\mathcal{R}$ containing the others.

LEMMA 4.5. *Lexicographically, the suffixes in the repetition sequences in $\mathcal{I}$ come before the suffixes in repetition sequences in $\mathcal{R}$.*

PROOF. Consider $S_{i,u} \in \mathcal{I}$ and $S_{j,u} \in \mathcal{R}$. Every suffix in $S_{i,u}$ is of the form $u^k v, k > 0$. Every suffix in $S_{j,u}$ is of the form $u^\ell w, \ell > 0$. The strings v and w are nonempty and (lexicographically) $w < v$. Because the chains are being processed in lexorder, we must have $u < w$. The result follows. □

Dealing with $\mathcal{I}$ is relatively easy. First, the terminating positions of sequences in $\mathcal{I}$ are sorted along with other suffixes from the u-chain sortable by induction, as usual, and ranks are assigned. Once the order of the terminating positions is known, the order of the nonterminating positions are *interleaved* and put in order. For example, if $\mathcal{I}$ contains two sequences $S_{i,u}$ and $S_{j,u}$ and rank$(i) <$ rank(j) then $rank(i - |u|) < rank(j - |u|) < rank(i - 2|u|) < rank(j - 2|u|) \ldots$, and so on, for each nonterminating position in $S_{i,u}$ and $S_{j,u}$.

The nonterminating positions in sequences in $\mathcal{R}$ can be interleaved in a similar way to those in $\mathcal{I}$ above, however, the sorting of the terminating positions is more delicate. To sort the nonterminating positions $\mathcal{R}$, we need to know the order of the terminating positions, relative to the other suffixes in the u-chain not sortable by induction. We link together all the nonterminating positions from right to left to form a special chain $\mathcal{C}$. Terminating positions are put in subchains as normal, along with other suffixes from the u-chain that are not sortable by induction. Now, we push $\mathcal{C}$ onto the chainstack *before* the other subchains. We continue to process chains from the chainstack, but now when

```
for $i \leftarrow 0$ to n do
  if ISA[$i$] $> 0$
    start $\leftarrow$ suff $\leftarrow$ i
    rank $\leftarrow$ ISA[$i$]-1
    repeat
      tmp $\leftarrow$ ISA[rank]
      ISA[rank] $\leftarrow$ -suff
      suff $\leftarrow$ rank
      rank $\leftarrow$ tmp
    until rank = start
    ISA[rank] $\leftarrow$ -suff
```

Fig. 3. Permute ISA into the SA in place.

we would normally assign a rank to a suffix, we instead add the suffix to the end of a list $\mathcal{Q}$. Later, when we reach $\mathcal{C}$ on the stack, we can use the order of the suffixes in $\mathcal{Q}$ to order the suffixes in $\mathcal{C}$.

To implement the above schemes, we require only constant extra space. The lists $\mathcal{Q}$ and $\mathcal{C}$ can be implemented in the space of the ISA (previously occupied by the suffixes in repetition sequences), with extra space required only for list heads and tails.

If $p \geq 2$, then the detection of (i, p, r) implies the presence of another $p - 1$ repetitions in other u-chains. The sorting of (i, p, r) means these other repetitions will be sorted by induction before they are detected as repetitions.

4.5 ISA to SA Transformation

As noted at the end of Section 4.2, our algorithms are well suited to computing the BWT of the input text: when a suffix i receives its rank, the character in the corresponding final column of the BWT matrix is the ith character in the transformed text. When the transformation completes, the ISA has been computed as a byproduct. If one requires the SA in memory rather than the ISA, the transformation can be done in place (constant extra memory required) and in linear time using the code in Figure 3.

The idea is to cyclically displace ranks in the ISA, moving suffixes into their place in the SA. Say suffix s with rank $q = \text{ISA}[s]$ is the next suffix we need to move into its place in the SA. We displace the rank at ISA[q] to a temporary variable r, and set ISA[q] $= -s$, with the negated sign bit indicating the element is in place. Now, r will be the position of suffix q in the SA. We continue following the cycle in this way until the displaced rank, r equals the index where we started, s. We then move to the right, until we find another element out of place, at which point we cycle again. When we reach the end of the array, the ISA has been transformed to the SA.

5. ENGINEERING AND IMPLEMENTATION DETAILS

In this section, we describe techniques we used to derive efficient implementations of the algorithms described in the previous section. Many of these exploit the case where $|\Sigma| \leq 256$, however, the most significant improvements come from methods that work to make the algorithms more cache friendly and are applicable to all alphabets.

5.1 Input Transformation

Before suffix sorting begins in earnest, we conditionally apply the following transformations to the input string.

1. *Alphabet compaction*. We recode the alphabet so that the symbols are contiguous and have the values $0 \cdots |\Sigma| - 1$. We do not require 0 to be reserved for the $x[n] = \$$ symbol, because we can assign suffix n rank 1 before sorting starts—any other suffix $i < n$ that is compared to depth $n - i$ will be sorted by induction, with $\mathrm{ISA}[n] = 1$ as its sort key.
2. *Boosting type U suffixes*. Instead of having the implementation adapt to sort the smaller of the type U and type V sets, if the type V suffixes are fewer, we transform the string so that the type U suffixes are fewer by replacing $x[i]$ with $|\Sigma| - x[i]$. Ensuring that comparison sorting only deals with U suffixes allowed us to simplify code and reduce development time. In a production environment one would probably implement two versions of the suffix-sorting routine.

In coming sections, Σ should be interpreted as referring to the alphabet after the above transformations have been applied.

5.2 Bucket Sorting

For strings on alphabets $|\Sigma| \leq 256$, we have the bucket sorting consider *two* symbols at a time on the fly. This means forming, at most, $|\Sigma|^2$ chains initially, and on every refinement. Such "dynamic symbol aggregation" is viable, because when $|\Sigma| \leq 256$, $|\Sigma|^2 \leq 65536$ is a manageable size. The combining of two characters into one not only speeds up string sorting, but also allows for induction sorting to be applied to more suffixes (see discussion below).

5.3 Induction Sorting

We collect suffixes for induction sorting by copying both the suffix number and its sort key (the rank of another suffix) to a separate array, which is dynamically grown as required. The suffix and its key are placed in adjacent positions in the array and moved together during comparison sorting. Arranging the items this way attracts a substantial benefit from the cache as it avoids misses that would be incurred by looking up the key in ISA every time it is needed during comparison sorting. The potential disadvantage is the extra space required. If there are m suffixes to be sorted by induction, the above array requires $8m$ bytes of storage. We have found, in practice, that m is usually very small relative to n (especially after alphabet compaction) and believe that pathological cases, where $8m$ bytes is problematic, are very rare. However, if space is a paramount concern there are the following alternatives:

1. We could only copy each suffix index i to the array, lookup the sort key in $\mathrm{ISA}[i + \ell]$, as required, and take the extra cache misses on the chin. This reduces the size of the array to $4m$ bytes.

2. If $4m$ bytes is still too much, a more complicated but space-efficient method can be adopted, which requires $2n + o(n)$ bits. We exploit the fact that the ranks used as sort keys are unique. Let r is the number of ranks assigned so far. We allocate T, a bit array of r bits. As each suffix sortable by induction is found in a u-chain, we set bit ISA$[i + |u|]$ of T. Also, instead of copying induction suffixes out, link them together in the ISA in the same way suffixes in a chain are linked together. Now process T so that the function *rank(k)*, $0 < k \leq r$ can be evaluated in constant time. This requires one pass over T and $r + o(r)$ bits of storage [González et al. 2005]. Finally, pass over the chain of induction suffixes and for each one encountered i, set ISA$[i]$ = $r + rank(\text{ISA}[i + |u|])$[1].

For comparison sorting, we employ our own implementation of the hybrid quicksort/heapsort approach described by Musser [1997]. Our version includes ternary quicksort with insertion sort for small partitions and we found it to be significantly faster than the approach of Bentley and McIlroy [1993].

When $|\Sigma| \leq 256$, comparison of bigrams instead of single symbols allows for a more powerful form of induction sorting. Specifically, when processing a chain with common prefix length ℓ, we can now classify suffixes into three types: Suffix i is of type A, if the rank for suffix $i + \ell - 1$ is known, and is of type B, if the rank for suffix $i + \ell$ is known. If i is not of type A or type B, then it is of type C. Lexicographically, type A suffixes are smaller than type B suffixes, which, in turn, are smaller than type C suffixes. As before, the order of the m type A suffixes can be determined via a comparison-based sort, using for suffix i the rank of suffix $i + \ell - 1$ as the sort key. Once sorted, the type A suffixes are assigned the next m ranks. The type B suffixes are treated similarly, using the rank of $j + \ell$ as the sort key for j. In fact, we can sort the type A and B suffixes in the same sort call by using as a key for a type A suffix i the rank of $i + \ell - 1$ and for a type B suffix the *negated* rank of $i + \ell$.

5.4 Producing the Suffix Array

When computing the suffix array with our approach, the final task is to transform the ISA into the SA. The problem with the algorithm for this task given in Figure 3 is its cache insensitivity: when following a cycle the next place in ISA accessed is essentially random, giving us little help from cache, with correspondingly slow runtime. We put some effort into developing a more cache friendly ISA to SA transform, which uses space of the input string as working space.[2] In the description of it which follows, we assume the string provides us with $2n$ bytes of working space and that n is a power of 2.

The string space, x, is divided into two equal, contiguous portions x_A and x_B, of n bytes (and capable of storing $n/4$ integers) each. Denote by Q_i, $i \in 1..4$, the set of suffixes, which belong in the ith quarter of SA. There are two phases to

[1] We did not actually use this method in any of our programs, but we include it here for the practitioner that is particularly paranoid about worst-case space usage.

[2] This destroys the string, but if we first compute symbol frequencies, it can be reconstructed from the suffix array (see Manzini [2004]).

the transformation. In the first phase, four left-to-right passes are made over the ISA as follows:

- *Pass* 1. Locate the members of Q_1 and move them to x_A. If ISA$[j] \in [0..n/4)$ then suffix $j \in Q_1$, so move it to x_A[ISA$[j]$] and mark ISA$[j]$ as empty. At the end of this pass Q_1 is sorted in x_A.
- *Pass* $k = 2, 3, 4$. If ISA$[j] \in [\frac{n(k-1)}{4}..\frac{nk}{4})$ then suffix $j \in Q_k$. If k is even (respectively odd) move j to $x_B[ISA[j] - \frac{n(k-1)}{4}]$ (respectively, x_A) and mark ISA$[j]$ empty. For each empty place we encounter in the scan (including a place just made empty by a move to x) move the next item in x_A (respectively, x_B) to that place, and *tag* it by setting the most significant two bits to $k - 1$. These tags will later allow us to determine which Q_i a suffix belongs to in constant time.[3] Copying back the suffixes belonging to Q_k this way ensures they are in ascending order of their position in SA (although they may not yet be adjacent), and means their order can later be read in a cache-friendly way.

Now the members of Q_4 are sorted in B and there are $n/4$ empty places at the end of ISA. These elements are copied from B to ISA$[\frac{3n}{4}..n-1]$, where they are in their final position and will not be moved again. The members of Q_1, Q_2, and Q_3 are in ISA$[0..\frac{3n}{4} - 1]$ and are tagged as described above.

The second phase completes the transformation. We scan ISA$[0..\frac{3n}{4} - 1]$ and if ISA$[j]$ belongs to Q_2 (respectively, Q_3) we move it to x_A (respectively, x_B). Membership to a given quarter is determined by inspecting the tags in constant time. When we move an element ISA$[j]$ we mark ISA$[j]$ as empty. During the scan, if we encounter a member of Q_1, we move it to the leftmost unoccupied position (which is kept track of with a separate pointer). When the scan completes, members of Q_1 are in their final positions in ISA$[0..\frac{n}{4} - 1]$. Finally, we copy Q_2 and Q_3 from A and B to ISA$[\frac{n}{4}..\frac{n}{2} - 1]$ and ISA$[\frac{n}{2}..\frac{3n}{4} - 1]$, respectively, and the ISA has been transformed in the SA. Because it mainly performs cache friendly left to right scans of ISA, x_A and x_B, we found the above procedure to be always faster than performing the transformation in place. To adapt the procedure to the case when the string provides us with only n bytes simply requires making us to make more (in fact eight) passes over the ISA and to use 3-bit tags (restricting $n < 2^{29}$).

6. EXPERIMENTS

We implemented two versions of our approach. First, `induce` refers to a program incorporating ideas from Sections 4.1, 4.2, and, 4.4. The second program `split` adds the type U/V splitting idea of Section 4.3, but is otherwise the same as `induce`. We tested these two programs against other leading implementations. These were:

`qsufsort`. A sorter using the *prefix-doubling* approach of Larsson and Sadakane [1999] downloaded from `http://www.larsson.dogma.net/research.html`.

[3] Tagging also restricts the use of this procedure to strings with $n < 2^{30}$

Table I. Description of the Data Set Used in Experiments for Small Alphabet Suffix Array Construction and Character-Based BWT Scenarios[a]

String	Mean LCP	Max LCP	Size (bytes)	Σ	Description
sprot34	89	7,373	109,617,186	66	SwissProt database
rfc	93	3,445	116,421,901	120	IETF RFC files
howto	267	70,720	39,422,105	197	Linux Howto files
reuters	282	26,597	114,711,151	93	Reuters news in XML
linux	479	136,035	116,254,720	256	Linux kernel source
jdk13c	679	37,334	69,728,899	113	JDK 1.3 documentation
etext99	1,108	286,352	105,277,340	146	Project Gutenberg text
chr22	1,979	199,999	34,553,758	4	Human chromosome 22
gcc	8,603	856,970	86,630,400	121	Gnu C Complier source
w3c2	42,300	990,053	104,201,579	255	HTML files from W3C site

[a] LCP refers to the longest common prefix amongst all suffixes in the string.

ds. An implementation of the *deep-shallow* algorithm by Manzini and Ferragina [2004] downloaded from http://www.mfn.unipmn.it/~manzini/lightweight/.

bpr. An implementation of the *bucket-pointer-refinement* algorithm by Schürmann and Stoye [2005] obtained from http://bibiserv.techfak.uni-bielefeld.de/bpr/.

All programs were written in C/C++. We are confident that all implementations tested are of high quality. In preliminary experiments, we tested many more suffix-sorting programs, but do not include measurements here because they were much slower than implementations, such as qsufsort. To assess practical performance we measured runtimes and memory usage of the implementations in three application scenarios, listed below.

- *Character BWT*. The task was to compute the BWT of files with $\Sigma \leq 256$. The other programs tested were qsufsort, ds, and bpr, to which we added a (trivial) final phase to compute the BWT. The files used for testing were from the corpus compiled by Manzini[4] and Ferragina [Manzini and Ferragina 2004] listed in Table I, which have become a defacto benchmark for testing suffix-sorting algorithms. Experiments measured runtimes and memory usage.
- *Small Alphabet Suffix Array Construction*. An important variation on the first scenario is to compute, instead, the suffix array for each file. Other programs tested were qsufsort, ds, and bpr. Test files were those in Table I, as per the BWT scenario. Experiments measured runtimes and memory usage.
- *Large Alphabet Suffix Array Construction*. The purpose was to measure performance on larger (16-bit) alphabets. Test strings were derived from the MF corpus files by combining every second bigram into a single symbol. The resulting strings have one-half as many symbols and $|\Sigma| \in 11,000..65,000$ (each new symbol requires two bytes). Developing the test set this way makes it easy for other researchers to compare future algorithms with ours. Statistics for these recoded strings are in Table II. The only other program we report on is qsufsort, which was the only competitive implementation we could

[4] http://www.mfn.unipmn.it/~manzini/lightweight/corpus/

Table II. Large Alphabet Data Set

String	Mean LCP	Max LCP	Size (symbols)	Σ	Description
sprot34	37	1,667	54,808,593	2,575	SwissProt database
rfc	33	1,626	58,210,950	8,828	IETF RFC files
howto	59	34,674	19,711,052	10,092	Linux Howto files
reuters	114	11,959	57,355,575	4,900	Reuters news in XML
linux	208	68,017	58,127,360	15,578	Linux kernel source
jdk13c	275	17,010	34,864,449	5,858	JDK 1.3 documentation
etext99	235	92,890	52,638,670	5,005	Project Gutenberg text
gcc	4,270	428,464	43,315,200	9,217	Gnu C Complier source
w3c2	15,421	495,026	52,100,789	58,917	HTML files from W3C site

Table III. Runtime (msec) of Peak Memory Usage in Bytes per Input Symbol (in parenthesis) for the Character-Based BWT Experiment

String	split	induce	ds	bpr	qsufsort
sprot34	**55210** (6.31)	79210 (6.16)	79880 (5.01)	104770 (9.01)	133830 (9.00)
rfc	**56240** (6.22)	76730 (6.22)	71010 (5.01)	99200 (9.06)	143180 (9.00)
howto	**16650** (6.25)	23550 (6.25)	20230 (5.01)	23390 (9.78)	38430 (9.00)
reuters	**68080** (6.16)	90530 (6.16)	152990 (5.01)	139180 (9.02)	169250 (9.00)
linux	**51240** (6.15)	77770 (6.15)	60220 (5.01)	74690 (9.58)	121290 (9.00)
jdk13c	**39640** (6.14)	55130 (6.26)	85190 (5.01)	67030 (9.08)	96610 (9.00)
etext99	**53720** (6.17)	70130 (6.25)	75790 (5.01)	97600 (9.12)	135840 (9.00)
chr22	**16650** (6.51)	19490 (6.74)	17260 (5.01)	17720 (9.00)	31100 (9.00)
gcc	57310 (6.20)	**56630** (6.11)	77630 (5.01)	63530 (9.16)	91480 (9.00)
w3c2	**60460** (6.17)	101040 (6.17)	129100 (5.01)	88930 (9.64)	176880 (9.00)

obtain suitable for larger alphabets. Experiments again measured runtimes and memory requirements.

All experiments were conducted on a 2.8 GHz Intel Pentium 4 processor with 2 GB main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2), executed with the −O3 option. All running times given are the average of four runs and do not include time spent reading input files. Times were recorded with the standard Unix `time` function. Memory usage was recorded with the `memusage` command available with most Linux distributions.

7. RESULTS

7.1 Character-Based BWT

Runtimes and peak memory requirements for the sorters are given in Table III and summarized in Figure 4. The `split` sorter is fastest on all files except *gcc*, where `induce` is fractionally quicker. Second place is shared between `induce` (*sprot, reuters, jdk13c, etext99*), `ds` (*rfc, howto, chr22*), and `bpr` (*w3c2*), with `ds` tending to better on the shorter files. The closest competitor to `split`, `ds`, ranges from being 1.1 (*gcc*) to more than two times (*reuters, w3c2*) slower. On the other hand, `ds` requires the least memory and uses 1–2n bytes per input character less than `split` and `induce`, and 4–5n bytes less than `bpr` and `qsufsort`.

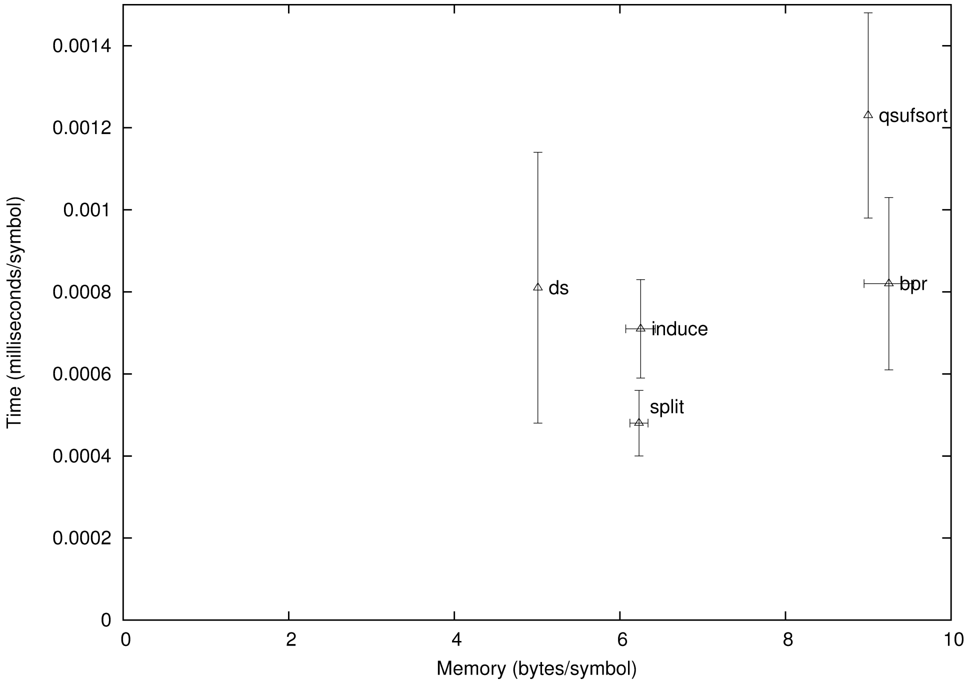

Fig. 4. Resource requirements of the five algorithms when computing the Burrows–Wheeler transform, averaged over the test corpus. Error bars are 1 S.D. Abscissa errors bars for ds and qsufsort are not shown, as they are insignificantly small.

7.2 Suffix Array Construction

In this scenario, split and induce require exactly *n* bytes less than for BWT, and so consume between 5.1 and 5.5 bytes per input symbol. This is competitive with ds, which again is the most space-efficient program. Here, as before, split dominates runtimes, but not as convincingly as it did when computing the BWT. It places second on *howto*, and *linux* (to ds), third on *gcc* (to bpr), and *chr22* (to ds). The result on *chr22* (where split takes 1.2 times as long as ds) indicates the effectiveness of Seward's pointer-copying heuristic (employed by both ds and bpr) on inputs with small Σ. On average, split is still easily the quickest per symbol, as illustrated in Figure 5 and Table IV. Relative to the BWT experiment, times for ds and bpr decrease, because they no longer convert the SA to the BWT text. On the other hand, times for split and induce increase, as they are now required to convert the ISA to the SA. This final step is extremely costly and constitutes 15–20% of split's overall runtime. We remark that when the job was to output the SA to disk, we observed results similar to that for the BWT scenario, with split a clearer winner.

7.3 Large Alphabet (See Later Table V)

In this experiment, both split and induce handsomely outperform qsufsort, which for all files except *jdk13c* is always more than two times slower—a greater gap than in the small Σ scenario. Our sorters also require nearly two bytes per

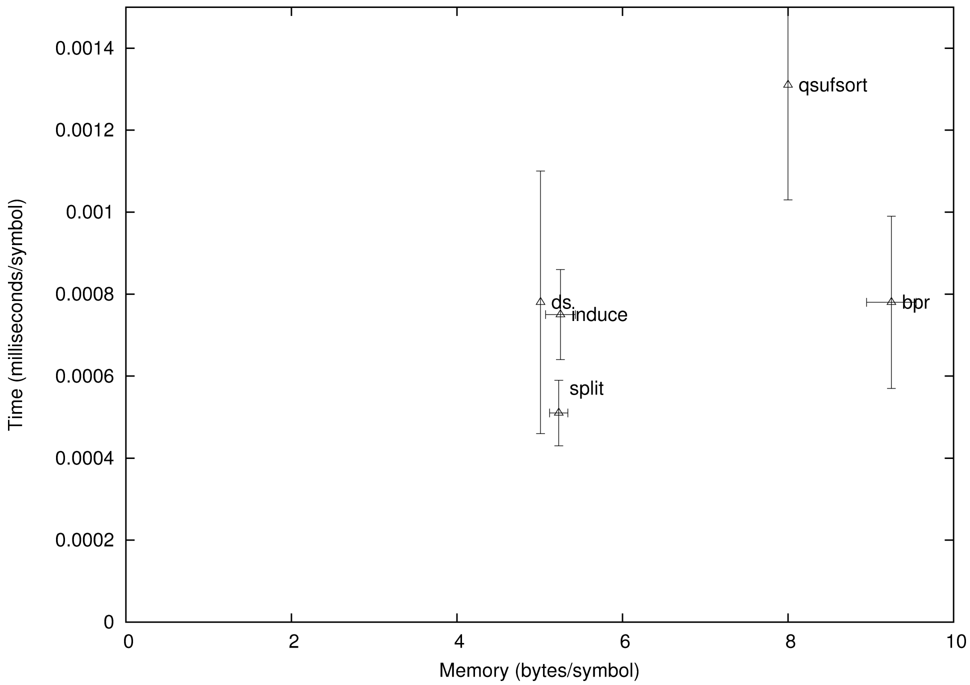

Fig. 5. Resource requirements of the five algorithms when computing the suffix array on the small Σ corpus. Error bars are 1 S.D. Abscissa errors bars for `ds` and `qsufsort` are not shown, as they are insignificantly small.

Table IV. Runtime (msec) of Peak Memory Usage in Bytes Per Input Symbol (in parenthesis) for the Suffix Array Construction, Small Σ Experiment

String	`split`	`induce`	`ds`	`bpr`	`qsufsort`
sprot34	**58990** (5.31)	83650 (5.16)	74090 (5.01)	98980 (9.01)	144070 (8.00)
rfc	**61310** (5.22)	81290 (5.22)	65000 (5.01)	93190 (9.06)	154210 (8.00)
howto	18410 (5.25)	24960 (5.25)	**18160** (5.01)	21520 (9.78)	39970 (8.00)
reuters	**72920** (5.16)	94210 (5.16)	146740 (5.01)	132930 (9.02)	183230 (8.00)
linux	56620 (5.15)	82780 (5.15)	**55110** (5.01)	69570 (9.58)	120430 (8.00)
jdk13c	**43570** (5.14)	58130 (5.26)	81880 (5.01)	63720 (9.08)	105130 (8.00)
etext99	**58160** (5.17)	74420 (5.25)	75790 (5.01)	91760 (9.12)	145520 (8.00)
chr22	18100 (5.51)	20970 (5.74)	**15510** (5.01)	15970 (9.00)	35200 (8.00)
gcc	62120 (5.20)	60080 (5.11)	73970 (5.01)	**59810** (9.16)	92220 (8.00)
w3c2	**65420** (5.17)	104210 (5.17)	118370 (5.01)	84040 (9.64)	192240 (8.00)

input symbol less working space. The difference in runtimes may be partially attributable to the shortening of the average LCP in the recoded test files. Even so, the results demonstrate the efficacy of our general approach to suffix-sorting in applications where Σ is large and sets `split` and `induce` apart from `ds` and `bpr`, which are unusable in such contexts.

Table V. Runtime (msec) of Peak Memory Usage in Bytes Per Input Symbol (in parenthesis) for the Suffix Array Construction, Large Σ Experiment

String	`split`	`induce`	`qsufsort`
sprot34	**38410** (6.31)	43420 (6.16)	85410 (8.00)
rfc	**40130** (6.22)	44440 (6.22)	85990 (8.00)
howto	**11420** (6.25)	13010 (6.25)	23270 (8.00)
reuters	**48190** (6.16)	53360 (6.16)	99980 (8.00)
linux	**36420** (6.15)	41120 (6.15)	79170 (8.00)
jdk13c	**29150** (6.14)	32280 (6.26)	56650 (8.00)
etext99	**33940** (6.17)	39190 (6.25)	78010 (8.00)
gcc	**9820** (6.20)	10970 (6.11)	59270 (8.00)
w3c2	**26330** (6.17)	29720 (6.17)	98610 (8.00)

8. CONCLUSIONS AND FUTURE WORK

We have described a new approach to suffix sorting that is fast, space efficient, and effective for a wide variety of applications. Our experiments have shown the two main variations of our algorithm to be superior, on average, to other schemes, both for suffix array construction and for computing the Burrows–Wheeler transformation, on byte sized and larger alphabets.

Induction sorting reduces the average number of character comparisons required to sort the suffixes to well below the average LCP and means that as more suffixes are sorted, sorting generally becomes easier. The identification of U/V suffixes essentially makes the induction-sorting process more efficient by eliminating the need for much of the comparison sorting. The repetitions heuristic further reduces character comparisons by dealing with long runs of identical substrings (most importantly, runs of identical characters and bigrams) with great ease. This eliminates the useless sorting rounds which plague many of the other suffix-sorting algorithms on very repetitive strings. When taken in combination, we believe these techniques ensure that catastrophic inputs are very rare indeed.

We end by commenting on two potential space-time tradeoffs. When constructing the suffix array, a slowdown comes in the final step, where the ISA is transformed to the SA. We believe the gap between our algorithms and the others would widen still, if $8n + zn$ bytes were budgeted as working space and made available throughout the algorithm. At the other extreme, the observation that once ISA[i] is ranked, ISA[$i + 1$] will never be used for induction sorting alludes to the possibility of a fast version of our approach that uses less than $4n + zn$ bytes of memory if the SA is allowed to be output to secondary storage, or if the BWT is the desired result. Preliminary experiments (see Puglisi [2005]) indicate that, on average, only around 25% of ranks are usuable at any given time. Future work will decide whether these ideas are also of practical interest.

ACKNOWLEDGMENTS

This paper incorporates work previously published as source code at `http://www.michael-maniscalco.com/msufsort.htm`, M. A. Maniscalco, August, 2004,

and in "Exposition and analysis of a suffix-sorting algorithm," S. J. Puglisi, Technical Report No. CAS-05-02-WS, Department of Computing and Software, McMaster University, May, 2005.

REFERENCES

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms 2*, 1, 53–86.

Andersson, A. and Nilsson, S. 1998. Implementing radix sort. *ACM Journal of Experimental Algorithmics 3*.

Bentley, J. L. and McIlroy, M. D. 1993. Engineering a sort function. *Software—Practice and Experience 23*, 11, 1249–1265.

Bentley, J. L. and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, New Orleans, LA. 360–369.

Burrows, M. and Wheeler, D. J. 1994. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, CA.

Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS 00)*. IEEE Computer Society, Redondo Beach, CA. 390–398.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. 2005. Practical implementation of rank and select queries. In *Poster Proceedings Volume of Fourth Workshop on Efficient and Experimental Algorithms (WEA'05)*. CTI Press and Ellinika Grammata, Greece. 27–38.

Grossi, R. and Vitter, J. S. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35*, 2, 378–407.

Itoh, H. and Tanaka, H. 1999. An efficient method for in memory construction of suffix arrays. In *Proceedings of the sixth Symposium on String Processing and Information Retrieval*. IEEE Computer Society, Cancun, Mexico, 81–88.

Kärkkäinen, J. and Burkhardt, S. 2003. Fast lightweight suffix array construction and checking. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003*, R. Baeza-Yates, E. Chávez, and M. Crochemore, Eds. Lecture Notes in Computer Science, vol. 2676. Springer-Verlag, Berlin. 55–69.

Kärkkäinen, J. and Sanders, P. 2003. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloq. Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 2971. Springer-Verlag, Berlin. 943–955.

Karp, R. M., Miller, R. E., and Rosenberg, A. L. 1972. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. Denver, Colorado. 125–136.

Kim, D. K., S., S. J., Park, H., and Park, K. 2003. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003*, R. Baeza-Yates, E. Chávez, and M. Crochemore, Eds. Lecture Notes in Computer Science, vol. 2676. Springer-Verlag, Berlin. 186–199.

Ko, P. and Aluru, S. 2003. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003*, R. Baeza-Yates, E. Chávez, and M. Crochemore, Eds. Lecture Notes in Computer Science, vol. 2676. Springer-Verlag, Berlin. 200–210.

Larsson, N. J. and Sadakane, K. 1999. Faster suffix–sorting. Tech. Rep. LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)/1-20/(1999)], Department of Computer Science, Lund University, Sweden.

Mäkinen, V. and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12*, 2, 40–66.

Manber, U. and Myers, G. W. 1993. Suffix arrays: A new model for on-line string searches. *SIAM Journal of Computing 22*, 5, 935–948.

Manzini, G. 2004. Two space saving tricks for linear time lcp computation. In *SWAT 2004: 9th Scandinavian Workshop on Algorithm Theory*, T. Hagerup and J. Katajainen, Eds. Lecture Notes in Computer Science, vol. 3111. Springer-Verlag, Berlin. 372–383.

MANZINI, G. AND FERRAGINA, P. 2004. Engineering a lightweight suffix array construction algorithm. *Algorithmica 40*, 33–50.

MCILROY, P. M., BOSTIC, K., AND MCILROY, M. D. 1993. Engineering radix sort. *Computing Systems 6*, 1, 5–27.

MOFFAT, A. AND ISAL, R. Y. K. 2005. Word-based compression using the Burrows-Wheeler transform. *Information Processing and Management 41*, 1175–1192.

MUSSER, D. R. 1997. Introspective sorting and selection algorithms. *Software—Practice and Experience 27*, 8, 983–993.

PUGLISI, S. J. 2005. Exposition and analysis of a suffix sorting algorithm. Tech. Rep. CAS-05-02-WS, Department of Computing and Software, McMaster University, Canada.

PUGLISI, S. J., SMYTH, W. F., AND TURPIN, A. H. 2005. The performance of linear time suffix sorting algorithms. In *Proceedings of the Data Compression Conference DCC '05*, M. Cohn and J. Storer, Eds. IEEE Computer Society Press, Los Alamitos, CA. 358–368.

PUGLISI, S. J., SMYTH, W. F., AND TURPIN, A. H. 2007. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*. *39*, 2, Article 1.

SADAKANE, K. 1998. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. In *Proceedings of the Data Compression Conference DCC '98*. IEEE Computer Society, Los Alamitos, CA. 129–138.

SADAKANE, K. 2002. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth ACM-SIAM Symposium on Discrete Algorithms*. 225–232.

SCHINDLER, M. 2002. szip homepage. http://www.compressconsult.com/szip/. Available Online.

SCHÜRMANN, K. AND STOYE, J. 2005. An incomplex algorithm for fast suffix array construction. In *Proceedings of The Seventh Workshop on Algorithm Engineering and Experiments (ALENEX'05)*. SIAM. 77–85.

SEWARD, J. 2000. On the performance of BWT sorting algroithms. In *Proceedings of the Data Compression Conference DCC '00*. IEEE Computer Society, Los Alamitos, CA. 173–182.

SEWARD, J. 2004. The bzip2 and libbzip2 home page. http://sources.redhat.com/bzip2/. Available Online.

SINHA, R. AND ZOBEL, J. 2004. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Journal of Experimental Algorithmics 9*.

SMYTH, W. F. 2003. *Computing Patterns in Strings*. Addison-Wesley-Pearson Education Limited, Essex, England.

VINES, P. AND ZOBEL, J. 1998. Compression techniques for Chinese text. *Software—Practice and Experience 28*, 12, 1299–1314.

Received December 2005; revised June 2005; accepted June 2006