SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:309–329
Published online 26 September 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.768

An incomplex algorithm for fast suffix array construction

Klaus-Bernd Schürmann and Jens Stoye*,†

Universität Bielefeld, Technische Fakultät, AG Genominformatik, Germany

SUMMARY

The suffix array of a string is a permutation of all starting positions of the string's suffixes that are lexicographically sorted. We present a practical algorithm for suffix array construction that consists of two easy-to-implement components. First it sorts the suffixes with respect to a fixed length prefix; then it refines each bucket of suffixes sharing the same prefix using the order of already sorted suffixes. Other suffix array construction algorithms follow more complex strategies. Moreover, we achieve a very fast construction for common strings as well as for worst case strings by enhancing our algorithm with further techniques. Copyright © 2006 John Wiley & Sons, Ltd.

Received 17 October 2005; Revised 15 May 2006; Accepted 21 May 2006

KEY WORDS: strings; indexing; suffix arrays; permutations

1. INTRODUCTION

Full text indices are used to process different kinds of sequences for diverse applications. The suffix tree is the best known full text index. It has been studied for decades and is used in many algorithmic applications. Nevertheless, in practice the suffix tree is used less than one would expect. We believe that this lack of practical usage is due to the high space requirements of the data structure. Moreover, while conceptually simple, the efficient implementation of suffix trees is difficult. Presently the construction of suffix trees in linear time is non-trivial.

Research on suffix arrays has increased since Manber and Myers [1] introduced this data structure as an alternative to suffix trees in the early 1990s. They presented an $O(n \log n)$ time algorithm to directly construct the suffix array of a text of length n. The algorithm is based on the doubling technique introduced by Karp *et al.* [2]. The theoretically best algorithms to construct suffix arrays run in $\Theta(n)$ time. However, although Farach *et al.* [3] correlated suffix sorting and linear-time suffix tree

*Correspondence to: Jens Stoye, Universität Bielefeld, Technische Fakultät, AG Genominformatik, Postfach 10 01 31, 33501 Bielefeld, Germany.
†E-mail: stoye@techfak.uni-bielefeld.de

construction in 2000, up until 2003 all known algorithms reaching this bound took a detour over suffix tree construction and afterwards obtained the ordered suffixes by traversing the suffix tree instead of directly constructing suffix arrays. The drawback of this approach is the space demand of linear-time suffix tree construction algorithms. The most space efficient implementation by Kurtz [4] uses between $8n$ and $14n$ bytes of space in total. Moreover, the linear-time suffix tree construction algorithms do not explicitly consider the memory hierarchy, which leads to unfavourable effects on current computer architectures. When the suffix tree grows over a certain size, the ratio of cache misses rises.

In 2003, the problem of direct linear-time construction of suffix arrays was solved independently by Kärkkäinen and Sanders [5], Kim *et al.* [6], and Ko and Aluru [7]. Shortly after, Hon *et al.* [8] gave a linear-time algorithm that needs $O(n)$ bits of working space.

Apart from these more theoretical results, there has also been much progress in practical suffix array construction. Larsson and Sadakane [9] presented a fast algorithm, called *qsufsort*, running in $O(n \log n)$ worst-case time using $8n$ bytes. In common with Manber and Myers [1] they utilize the doubling technique of Karp *et al.* [2]. Recently, Kim *et al.* [10] introduced a divide and conquer algorithm based on [6] with $O(n \log \log n)$ worst-case time complexity, but with faster practical running times than the previously mentioned linear-time algorithms. Both algorithms use the odd–even scheme introduced by Farach [11] for suffix tree construction.

Other viable algorithms mainly consider space requirements. They are called *lightweight algorithms* due to their small space requirements. Itoh and Tanaka [12] as well as Seward [13] proposed algorithms using only $5n$ bytes. In theory their worst-case time complexity is $\Omega(n^2)$. However, in practice they are very fast if the average Longest Common Prefix (LCP) is small. (By LCP we refer to the length of the LCP of two consecutive suffixes in the suffix array.) More recently, Manzini and Ferragina [14] engineered an algorithm called *deep shallow* suffix sorting. They combine different methods to sort suffixes depending on the LCP lengths and did in-depth work on finding suitable settings to achieve fast construction. The algorithm's space demands are small, and it is applicable for strings with high average LCP.

The most recent lightweight algorithm was developed by Burkhardt and Kärkkäinen [15]. It is called the *difference-cover* algorithm. Its worst-case running time is $O(n \log n)$ and it uses $O(n/\sqrt{\log n})$ extra space. For common real-life data, though, the algorithm is on average slower than *deep shallow* suffix sorting.

The above mentioned suffix array construction algorithms meet some of the following requirements for practical suffix array construction:

- fast construction for common real-life strings (small average LCP)—Larsson and Sadakane [9], Itoh and Tanaka [12], Seward [13], Manzini and Ferragina [14], Kim *et al.* [10];
- fast construction for degenerate strings (high average LCP)—Larsson and Sadakane [9], Manzini and Ferragina [14], Kim *et al.* [10], Burkhardt and Kärkkäinen [15], and others [1,5–7];
- small space demands—Itoh and Tanaka [12], Seward [13], Manzini and Ferragina [14], Burkhardt and Kärkkäinen [15].

Based on our experience with biological sequence data, we believe that further properties are required. There are many applications where very long sequences with mainly small LCPs, interrupted by occasional very large LCPs, are investigated. In genome comparison, for example, concatenations of similar sequences are indexed to find common subsequences, repeats, and unique regions.

DOI: 10.1002/spe

Thus, to compare genomes of closely related species, one has to build suffix arrays for strings with highly variable LCPs.

We believe that the characteristics as observed in a bioinformatics context can be found in other application areas as well. Also, in Burrows–Wheeler text compression, the problem of computing the Burrows–Wheeler Transform [16] by block-sorting the input string is equivalent to suffix array construction. These facts stress the importance of efficient ubiquitous suffix array construction algorithms.

In Section 2 we give the basic definitions and notations concerning suffix arrays and suffix sorting. Section 3 is devoted to our approach, the *bucket-pointer refinement* (*bpr*) algorithm. Experimental results are presented in Section 4. Section 5 concludes and discusses open questions.

2. SUFFIX ARRAYS AND SORTING—DEFINITIONS AND TERMINOLOGY

Let Σ be a finite ordered alphabet of size $|\Sigma|$ and $t = t_1 t_2 \dots t_n \in \Sigma^n$ a text over Σ of length n. Let \$ be a character not contained in Σ, and assume $\$ < c$ for all $c \in \Sigma$. For illustration purposes, we will often consider a \$-padded extension of string t, denoted $t^+ = t\$^n$. For $1 \leq i \leq n$, let $s_i(t) = t_i \dots t_n$ indicate the ith (non-empty) suffix of string t. The starting position i is called its *suffix number*.

The *suffix array* $sa(t)$ of t is a permutation of the suffix numbers $\{1 \dots n\}$, according to the lexicographic ordering of the n suffixes of t. More precisely, for all pairs of indices (k, l), $1 \leq k < l \leq n$, the suffix $s_{sa[k]}(t)$ at position k in the suffix array is lexicographically smaller than the suffix $s_{sa[l]}(t)$ at position l in the suffix array.

A *bucket* $b = [l, r]$ is an interval of the suffix array sa, determined by its left and right end in sa. A bucket $b_p = [l, r]$ is called a *level-m bucket*, if all contained suffixes $s_{sa[l]}(t), s_{sa[l+1]}(t), \dots, s_{sa[r]}(t)$ share a common prefix p of length m.

A *radix step* denotes the part of an algorithm in which strings are sorted according to the characters at a certain *offset* in the string. The *offset* is called *radix level*. A radix step is like a single iteration of *radix sort*.

Range reduction performs a monotone, bijective function, $rank : \Sigma \to \{0, \dots, |\Sigma| - 1\}$, of the alphabet to the first $|\Sigma|$ natural numbers. More precisely, for two characters $c_1 < c_2$ of alphabet Σ, $rank(c_1)$ is smaller than $rank(c_2)$. Applied to a string t, *range reduction* maps each character c of t to its *rank*, $rank(c)$. Note that the suffix array of a range reduced string equals the suffix array of the original string.

A *multiple character encoding* for strings of length d is a monotone bijective function $code_d : \Sigma^d \to \{0, \dots, |\Sigma|^d - 1\}$ such that for strings u, v of length d, $code_d(u) < code_d(v)$ if and only if u is lexicographically smaller than v. For a given rank function, such an encoding can easily be defined as $code_d(u) = \sum_{k=1}^{d} |\Sigma|^{d-k} rank(u[i + k - 1])$. The encoding can be generalized to strings of length greater than d, by just encoding the first d characters. Given the encoding $code_d(i)$ for suffix $s_i(t)$, $1 \leq i < n$, the encoding for suffix $s_{i+1}(t)$ can be derived by shifting away the first character of $s_i(t)$ and adding the *rank* of character $t^+[i + d]$:

$$code_d(i + 1) = |\Sigma|(code_d(i) \bmod |\Sigma|^{d-1}) + rank(t^+[i + d]) \tag{1}$$

DOI: 10.1002/spe

3. THE BUCKET-POINTER REFINEMENT ALGORITHM

Most of the previously mentioned practical algorithms order suffixes with respect to their leading characters into buckets, which are then recursively refined. Before describing our new algorithm that uses a similar though somewhat enhanced technique, we classify the specific techniques used.

3.1. Classification of techniques

The first type of bucket refinement techniques found in the literature is formed by string sorting methods without using the dependencies among suffixes. Most representatives of this class sort the suffixes with respect to their leading characters and then refine the groups of suffixes with equal prefixes by recursively performing radix steps until unique prefixes are obtained [17–19].

The second type of algorithms use the order of previously computed suffixes in the refinement phase. If suffixes $s_i(t)$ and $s_j(t)$ share a common prefix of length *offset*, the order of $s_i(t)$ and $s_j(t)$ can be derived from the ordering of suffixes $s_{i+offset}(t)$ and $s_{j+offset}(t)$. Many practical algorithms that use this technique, also apply methods of the first type to fall back upon if the order of suffixes at *offset* is not yet available [12–14].

We further divide the second type into two subgroups: the *push* method, and the *pull* method. The *push* method uses the ordering of previously determined groups that share a leading character and pass this ordering on to undetermined buckets. Some representatives that use this technique are: Itoh and Tanaka's *two-stage* algorithm [12], Seward's *copy* algorithm [13], and the *deep shallow* algorithm of Manzini and Ferragina [14].

Pull methods look up the order of suffixes $s_{i+offset}(t)$ and $s_{j+offset}(t)$ to determine the order of $s_i(t)$ and $s_j(t)$. This technique is used in many algorithms. Larsson and Sadakane's *qsufsort* [9], Seward's *cache* algorithm [13], and the *deep shallow* sorting of Manzini and Ferragina [14] are examples of practical algorithms that use this method. The *difference-cover* algorithm by Burkhardt and Kärkkäinen [15] first sorts a certain subset of suffixes to ensure the existence of a bounded offset to this subset of previously sorted suffixes.

The linear-time algorithms of Kärkkäinen and Sanders [5], Kim *et al.* [6], and Ko and Aluru [7], as well as the $O(n \log \log n)$ time algorithm of Kim *et al.* [10] follow different divide and conquer schemes, but share the basic framework. They divide the suffixes into two groups, recursively sort one of the groups, use the ordering of suffixes in the sorted group to determine the ordering of suffixes in the other group, and finally merge the two sorted groups to receive the total ordering of all suffixes, namely the suffix array. These are not bucket refinement algorithms. Nevertheless, since they all pass the sorted order of suffixes of one group on to determine the ordering of the other group, they could be classified as *push* algorithms (although their overall strategy is more advanced).

3.2. The new algorithm

The new *bpr* algorithm that we present in this paper combines the approaches of refining groups with equal prefix by recursively performing radix steps and the *pull* technique. It mainly consists of two simple phases. Given a parameter d (usually less than $\log n$), the suffixes are lexicographically sorted in the first phase, so that suffixes with the same d-length prefix are grouped together. Before entering the second phase, a pointer to its bucket $bptr[i]$ is computed for each suffix with suffix number i,

DOI: 10.1002/spe

String to build suffix array for: $t =$ e f e f e f a a (1 2 3 4 5 6 7 8)

	a$	aa		ef		fa		fe	
sa after initial sorting ($d = 2$):	8	7	1	3	5	6	2	4	
	1	2	3	4	5	6	7	8	
bptr after initial sorting:	5	8	**5**	8	**5**	6	**2**	1	
sa after sorting bucket [3, 5]:	8	7	5	1	3	6	2	4	
	1	2	3	4	5	6	7	8	
bptr after updating positions 1, 3, 5:	**5**	8	**5**	8	**3**	6	2	1	

Figure 1. Example of the refinement procedure after the initial sorting of suffixes regarding prefixes of length $d = 2$. The suffix array *sa* and the table of bucket pointers *bptr* is shown before and after applying the refinement steps to the bucket [3, 5] containing suffixes 1, 3, 5 with respect to $offset = 2$. The sort keys (drawn in bold face) are $sortkey(1) = bptr[1 + 2] = 5$, $sortkey(3) = bptr[3 + 2] = 5$, and $sortkey(3) = bptr[5 + 2] = 2$. After the sorting, the bucket pointers for the suffixes 1, 3, 5 in bucket [3, 5] are updated to $bptr[1] = 5$, $bptr[3] = 5$, and $bptr[5] = 3$.

such that suffixes with the same d-length prefix share the same bucket pointer. Lexicographically smaller suffixes have smaller bucket pointers. In our descriptions and implementation we use the position of the rightmost suffix in each bucket as bucket pointer.

In the second phase, the buckets containing suffixes with equal prefix are recursively refined. Let $[l, r]$ be the segment in the suffix array *sa* forming a bucket B of suffixes $sa[l]$, $sa[l + 1]$, ..., $sa[r]$ with equal prefix of length *offset*. Then, the refinement procedure works in the following way. The suffixes in B are sorted according to the bucket pointer at offset *offset*. That is, for each suffix $sa[k]$ in B, $l \leq k \leq r$, $bptr[sa[k] + offset]$ is used as the sort key.

After sorting the suffixes of B, the sub-buckets are determined that contain suffixes sharing the same sort key. Then, for each suffix $sa[k]$, $l \leq k \leq r$, its bucket pointer $bptr[sa[k]]$ is updated to point to the new sub-bucket containing $sa[k]$. Finally, all sub-buckets are refined recursively by calling the refinement procedure with increased offset, $offset_{\text{new}} = offset_{\text{old}} + d$. After termination of the algorithm, *sa* is the suffix array and the array *bptr* reflects the inverse suffix array. An example of the refinement procedure is given in Figure 1.

Properties. The main improvement of our algorithm, compared to earlier algorithms performing bucket refinements, is that it benefits from the immediate use of subdivided bucket pointers after each refinement step. With increasing number of subdivided buckets, it becomes more and more likely that different bucket pointers can be used as sort keys during a refinement step, such that the expected recursion depth decreases for the buckets refined later. The final position of a suffix i in the current bucket is reached at the latest when $bptr[i + offset]$ is unique for the current offset. That is, when the suffix $i + offset$ has reached its final position.

Another improvement of our algorithm is that in each recursion step *offset* can be increased by d. Hence, the recursion depth decreases by a factor of d, compared to algorithms performing characterwise radix steps.

DOI: 10.1002/spe

Note that the algorithm can be applied to arbitrary ordered alphabets, since it just uses comparisons to perform suffix sorting.

Analysis. So far we were not able to determine tight time bounds for our algorithm. The problem is that the algorithm quite arbitrarily uses the dependence among suffixes. Hence, we can only state a straight forward quadratic time complexity for the general worst case, while a subquadratic upper time bound can be found for certain periodic strings.

The simple $O(n^2)$ upper time bound can be seen as follows. The first phase of the algorithm can simply be performed in linear time (see Section 3.3 for more details). For the second phase, we assume that an $O(n \log n)$ time sorting procedure is applied. In each recursion level there are at most n suffixes to be sorted in $O(n \log n)$ time. The maximal offset when the end of the string is reached is n, and *offset* is incremented by d in each recursive call. Hence, the maximal recursion depth is n/d. Therefore, the worst-case time complexity of the algorithm is limited by $O(n^2 \log n/d)$. By setting $d = \log n$, we obtain the upper bound of $O(n^2)$ without taking into account the dependencies among suffixes.

Now, we focus on especially bad instances for our algorithm; in particular, strings maximizing the recursion depth. Since the recursion depth is limited by the LCPs of suffixes to be sorted, periodic strings maximizing the average LCP are especially hard strings for our algorithm.

A string $\mathtt{a}^n$ consisting of one repeated character maximizes the average LCP and is therefore analysed as a particularly difficult input string. In the first phase of our algorithm the last $d-1$ suffixes $s_{n+2-d}(\mathtt{a}^n), s_{n+3-d}(\mathtt{a}^n), \ldots, s_n(\mathtt{a}^n)$ are mapped to singleton buckets. One large bucket containing all the other suffixes with leading prefix $\mathtt{a}^d$ remains to be refined. In each recursive refinement step, the remaining large bucket is again subdivided into *offset* singleton buckets and one larger bucket, while *offset* is incremented by d, starting with $\mathit{offset} = d$ in step 1. Hence, in the ith refinement step, $d \cdot i$ suffixes are subdivided into singleton buckets. The recursion proceeds until all buckets are singleton, that is, until a recursion depth *recdepth* is reached, such that $n \leq d - 1 + \sum_{i=1}^{\mathit{recdepth}} d \cdot i = d - 1 + d(\mathit{recdepth}(\mathit{recdepth}+1)/2)$. Therefore, for the string $\mathtt{a}^n$ the recursion depth *recdepth* of our algorithm is in $\Theta(\sqrt{n/d})$. Less than n suffixes have to be sorted in each recursive step. Hence, we multiply sorting complexity and recursion depth *recdepth* to get the time bound $O(n \log n\sqrt{n/d})$ of our algorithm for the string $\mathtt{a}^n$. By setting $d = \log n$ we achieve a running time of $O(n \log n\sqrt{n/\log n}) = O(n^{3/2}\sqrt{\log n})$. By taking into account the decreasing number of suffixes to be sorted with increasing recursion depth, a more sophisticated analysis shows the same time bound. Therefore, this worst-case time bound seems to be tight for this string.

3.3. Engineering and implementation

In this section, we present more detailed descriptions of the two phases of the algorithm and briefly explain enhancements to achieve faster construction times in practice.

Phase 1

We perform the initial sorting regarding the d-length prefixes of the suffixes by bucket sort, using $\mathit{code}_d(i)$ as the sort key for suffix i, $1 \leq i \leq n$.

The bucket sorting is performed using two scans of the sequence, thereby successively computing $\mathit{code}_d(i)$ for each suffix using Equation (1). There are $|\Sigma|^d$ buckets, one for each possible code_d.

DOI: 10.1002/spe

In the first scan, the size of each bucket is determined by counting the number of suffixes for each possible $code_d$. The outcome of this is used to compute the starting position for each bucket. These positions are stored in the table *bkt*, which is of size $|\Sigma|^d$. During the second scan, the suffix numbers are mapped to the buckets, where suffix number i is mapped to bucket number $code_d(i)$.

After the bucket sort, the computation of the bucket pointer table *bptr* can be performed by another scan of the sequence. For suffix i, the bucket pointer $bptr[i]$ is simply the rightmost position of the bucket containing i, $bptr[i] = bkt[code_d(i) + 1] - 1$.

Phase 2

We now give a more in-depth description of the three steps of the refinement procedure and present improvements to the basic approach.

Sorting. In the refinement procedure, the suffixes are first sorted with respect to a certain offset using the bucket pointer $bptr[i + offset]$ as the sort key for suffix i. The sorting algorithms used to perform this are well known. *Insertion Sort* is used for buckets of size up to 15. For the larger buckets, we apply *Quicksort* with Lomuto's partitioning scheme [20, Problem 8-2]. The pivot is chosen to be the median of three elements due to Singleton [21]. In addition, we apply a heuristic for the case that we have many equal bucket pointers. After a partitioning step, we just extend the partitioning position as long as the sort key equals the pivot, such that there is an already sorted region around this position and the size of the remaining unsorted partitions decreases. This heuristic works especially well for periodic strings.

Updating bucket pointers. The update of bucket pointers can simply be performed by a right-to-left scan of the current bucket. As long as the sort keys of consecutive suffixes are equal, they are located in the same refined bucket, and the bucket pointer is set to the rightmost position of the refined bucket. Note that the refined bucket positions are implicitly contained in the bucket pointer table *bptr*. The left pointer l of a bucket is the right pointer of the bucket directly to the left increased by one, and the right pointer r is simply the bucket pointer for the suffix $sa[l]$ at position l, $r = bptr[sa[l]]$, since for each suffix i its bucket pointer $bptr[i]$ points to the rightmost position of its bucket.

Recursive Refinement. The recursive refinement procedure is usually called with an incremented offset, $offset + d$. Note that for a sub-bucket b_{sub} of b containing each suffix $s_i(t)$, for which the suffix $s_{i+offset}(t)$ with respect to *offset* is also contained in b, the offset can be doubled. This is so because all suffixes contained in b share a common prefix of length *offset*, and for each suffix $s_i(t)$ in b_{sub}, there is also the suffix $s_{i+offset}(t)$ with respect to *offset* in b. Hence, all suffixes contained in b_{sub} share a prefix of length $2offset$.

Further on, we add an additional heuristic to avoid the unnecessary repeated sorting of buckets. For a bucket consisting only of suffixes that all share a common prefix much larger than the current offset, many refinement steps may be performed without actually refining the bucket. This may continue until *offset* reaches the length of the common prefix. Therefore, if a bucket is not refined during a recursion step, we search for the lowest offset dividing the bucket. This is performed by just iteratively scanning the bucket pointers of the contained suffixes with respect to *offset* and incrementing *offset* by d after each run. As soon as a bucket pointer different from the others is met, the current *offset* is used to call the refinement procedure.

DOI: 10.1002/spe

Table I. Worst-case time complexities of the investigated suffix array construction algorithms.

bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
$O(n^2)$	$O(n^2 \log n)$	$O(n^2 \log n)$	$O(n^2 \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log \log n)$	$\Theta(n)$

Further improvements. We enhanced our algorithm by the *copy* method of Seward [13] that was earlier mentioned by Burrows and Wheeler [16]. If the buckets consisting of suffixes with the leading character p are determined, they form a level-1 (L1) bucket B_p. Let $b_{c_1 p}, b_{c_2 p}, \ldots, b_{c_{|\Sigma|} p}, c_i \in \Sigma$, be level-2 (L2) buckets with the second character p. The ordering of suffixes in B_p can be used to forward the ordering to the specified L2 buckets by performing a single pass over B_p. If i is the current suffix number in B_p and $c = t[i-1]$ is the previous character, then the suffix number $i-1$ is assigned to the first non-determined position of bucket b_{cp}. Seward also showed how to derive the positions of suffixes in b_{pp} using the buckets $b_{c_i p}$, $p \neq c_i \in \Sigma$. Hence, the usage of Seward's *copy* technique avoids the comparison-based sorting of more than half of the buckets.

Our program sorts the suffixes in L1 buckets B_c, $c \in \Sigma$, comparison-based in ascending order with respect to the number of suffixes, $|B_c| - |b_{cc}|$.

4. EXPERIMENTAL RESULTS

In this section we investigate the practical construction times of our algorithm for DNA sequences, common texts, and artificially generated strings with a high average LCP.

We compared our *bpr* implementation [22] to seven other practical implementations: *deep shallow* by Manzini and Ferragina [14], *cache* and *copy* by Seward [13], *qsufsort* by Larsson and Sadakane [9], *difference-cover* by Burkhardt and Kärkkäinen [15], *divide* and *conquer* by Kim *et al.* [10], and *skew* by Kärkkäinen and Sanders [5]. The worst-case time complexities of the algorithms are shown in Table I. Since the original *skew* implementation is limited to integer alphabets, in all instances we mapped the character string to an integer array.

The experiments were performed on a computer with 1.3 GHz Intel Pentium^{TM}M (Klamath) processor, running the Linux operating system. The memory hierarchy is composed of separate L1 instruction and data cache, each of size 32 Kbyte and 3 cycles latency, a 1024 Kbyte L2 cache with 10 cycles latency, and 512 Mbytes of main memory. Each cache is 8-way associative with 64 byte line size. All programs were compiled with the *gcc* compiler, respectively *g++* compiler, with optimization options '-O3 -fomit-frame-pointer -funroll-loops'.

The investigated data files are listed in Table II and are ordered by average LCP. For the DNA sequences, we selected genomic DNA from different species: the whole genome of the bacteria *Escherichia coli* (*E. coli*), the fourth chromosome of the flowering plant *Arabidopsis thaliana* (*A. thaliana*), the first chromosome of the nematode *Caenorhabditis elegans* (*C. elegans*), and the human (*H. sapiens*) chromosome 22. Moreover, we investigated the construction times for different

DOI: 10.1002/spe

Table II. Description of the data set.

Data set	LCP average	LCP maximum	String length	Alphabet size	Description
E. coli genome	17	2815	4 638 690	4	*Escherichia coli* genome
A. thaliana chr. 4	58	30 319	12 061 490	7	*A. thaliana* chromosome 4
H. sapiens chr. 22	1979	199 999	34 553 758	5	*H. sapiens* chromosome 22
C. elegans chr. 1	3181	110 283	14 188 020	5	*C. elegans* chromosome 1
6 *Streptococci*	131	8091	11 635 882	5	6 *Streptococcus* genomes
4 *Chlamydophila*	1555	23 625	4 856 123	6	4 *Chlamydophila* genomes
3 *E. coli*	68 061	1 316 097	14 776 363	5	3 *E. coli* genomes
bible	13	551	4 047 392	63	King James bible
world192	23	559	2 473 400	94	CIA world fact book
rfc	87	3445	50 000 000	110	Texts from the RFC project
sprot34	91	2665	50 000 000	66	SwissProt database
howto	267	70 720	39 422 105	197	Linux Howto files
reuters	280	24 449	50 000 000	91	Reuters news in XML
w3c	478	29 752	50 000 000	255	html files of w3c homepage
jdk13	654	34 557	50 000 000	110	JDK 1.3 doc files
linux	766	136 035	50 000 000	256	linux kernel source files
etext99	1845	286 352	50 000 000	120	Project Gutenberg texts
gcc	14 745	856 970	50 000 000	121	*gcc* 3.0 source files
random	4	9	20 000 000	26	Bernoulli string
period 500 000	9 506 251	19 500 000	20 000 000	26	Repeated Bernoulli string
period 1 000	9 999 001	19 999 000	20 000 000	26	Repeated Bernoulli string
period 20	9 999 981	19 999 980	20 000 000	17	Repeated Bernoulli string
Fibonacci	5 029 840	10 772 535	20 000 000	2	*Fibonacci string*

concatenated DNA sequences of certain families. For this we used six *Streptococcus* genomes, four genomes of the *Chlamydophila* family, and three different *E. coli* genomes.

For the evaluation of common real-world strings, we used the suite of test files given by Manzini and Ferragina in [14]. The strings are usually concatenations of text files, or alternatively, *tar* archives. Due to the memory constraints of our test computer, we just took the last 50 million characters of those text files that exceeded the 50 million character limit.

The artificial files were generated as described by Burkhardt and Kärkkäinen [15]: a random string made out of Bernoulli distributed characters and periodic strings composed of an initial random string that is repeated until a length of 20 million characters is reached. We used initial random strings of length 20, 1000 and 500 000 to generate the periodic strings. Also, we investigated a Fibonacci string of length 20 million characters.

The suffix array construction times of the different algorithms are given in Tables III–V. Table III contains the construction times for the DNA sequences. Our *bpr* algorithm is the fastest suffix array construction algorithm for most DNA sequences. Only *deep shallow* is about 5% faster for the fourth

DOI: 10.1002/spe

Table III. Suffix array construction times for different DNA sequences and generalized DNA sequences by different algorithms, with $d = 7$ for *bpr*.

	Construction time (s)							
DNA sequences	*bpr*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
E. coli genome	**1.46**	1.71	3.69	2.89	2.87	4.32	5.81	13.48
A. thaliana chr. 4	5.24	**5.01**	12.24	9.94	8.42	13.29	16.94	38.30
H. sapiens chr. 22	**15.92**	16.64	40.08	30.04	26.52	44.93	51.31	112.38
C. elegans chr. 1	**5.70**	6.03	20.79	17.48	13.09	16.94	18.64	41.28
6 *Streptococci*	**5.27**	5.97	14.43	10.38	13.16	14.50	16.40	36.24
4 *Chlamydophila*	**2.31**	3.43	17.46	14.45	7.49	5.59	6.13	14.13
3 *E. coli*	**8.01**	13.75	437.18	1294.30	32.72	20.57	21.58	47.32

Table IV. Suffix array construction times for various texts by different algorithms, with $d = 3$ for *bpr*.

	Construction time (s)							
Text	*bpr*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
bible	1.90	**1.41**	2.91	2.24	3.17	3.74	6.39	11.59
world192	1.05	**0.73**	1.47	1.24	1.91	2.28	3.57	6.45
rfc	31.16	**26.37**	57.97	55.21	58.10	71.10	101.57	169.03
sprot34	35.75	**29.77**	71.95	71.96	60.24	81.76	104.71	169.16
howto	22.10	**19.63**	39.92	47.27	41.14	48.45	83.32	141.50
reuters	**47.32**	52.74	111.80	157.63	73.19	108.85	108.84	169.18
w3c2	**41.04**	61.37	82.46	167.76	69.40	96.02	105.89	163.15
jdk13	**40.35**	47.23	101.58	263.86	73.75	97.12	98.13	162.39
linux	**23.72**	23.95	50.93	99.47	61.01	65.66	98.06	173.05
etext99	**32.60**	33.25	68.84	267.48	61.19	65.31	110.95	190.33
gcc	**33.19**	76.23	2894.81	21 836.56	59.44	73.54	83.96	162.06

chromosome of *A. thaliana*. However, for sequences with higher average LCP, *bpr* outperforms all other existing algorithms. For the concatenated sequence of three *E. coli* genomes with average LCP 68 061, *deep shallow*, the closest competitor of *bpr*, is 1.72 times slower.

For the real-world strings the running times of the investigated algorithms are shown in Table IV. For the texts with small average LCP, *deep shallow* is the fastest suffix array construction algorithm. *Bpr* shows the second best running times. However, when the average LCP exceeds a limit of about 300, our algorithm is the fastest among all investigated algorithms. For *gcc* with an average LCP of 14 745, it is clearly faster than the others.

DOI: 10.1002/spe

Table V. Suffix array construction times for artificial strings by different algorithms, with $d = 3$ for *bpr*.

	Construction time (s)							
Artificial strings	*bpr*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
random	8.95	**8.31**	15.17	10.83	14.83	20.19	37.52	47.25
period 500 000	**12.33**	710.52	—	—	89.52	47.28	31.21	61.04
period 1000	**16.76**	1040.45	—	—	86.45	78.87	21.96	52.34
period 20	41.61	—	—	—	74.74	59.38	**10.33**	43.24
Fibonacci string	28.00	680.43	—	—	82.62	69.63	**22.21**	38.17

For degenerated strings the construction times are given in Table V. Wherever an algorithm used more than 12 h of computation time, we stopped the computation. This is indicated by a dash in the table. Here, *bpr* is much quicker than *deep shallow*, *cache*, and *copy*, which are very fast algorithms for strings with lower average LCP. Even compared to the algorithms *qsufsort*, *difference-cover*, *divide & conquer*, and *skew* with good worst-case time complexities, our algorithm performs very well. For strings with period 1000 and 500 000 it is by far the fastest algorithm. For strings with period 20 and for the Fibonacci string, just the *divide & conquer* algorithm with its $O(n \log \log n)$ worst-case time complexity is faster.

In summary, one can say that *bpr* is always among the two fastest of the investigated algorithms. In most cases, and specifically for all but one DNA sequences, it is the fastest algorithm. For strings with very small average LCP its running time is comparable to *deep shallow*, *cache*, and *copy*. With an increasing average LCP, it is clearly the fastest algorithm. Even for worst-case strings with very high average LCP, *bpr* performs well compared to the algorithms *qsufsort*, *difference-cover*, and *divide & conquer* with good worst-case time complexity, whereas the construction times for *deep shallow*, *cache*, and *copy* escalate.

4.1. Performance on very large scale data sets

In a separate experiment, we took the construction times for the human and mouse genome on a Sun Fire V1280 server running twelve 900 MHz UltraSparc-III processors. Its memory hierarchy is composed of 32 Kbyte level-1 instruction and 64 Kbyte level-1 data cache, 8 Mbyte level-2 cache, and 96 Gbyte main memory. The genomes are concatenated DNA sequences of all their chromosomes where the human genome consists of about 3.08 billion nucleotides and the mouse genome of about 2.62 billion, in total. We compiled the implementations of suffix array construction algorithms with further 64-bit options '-m64 -mptr64'.

Bpr with $d = 9$ needs 7 h 9 min for the human genome and 5 h 37 min for the mouse genome. The other algorithms abort unexpectedly. It seems that their particular implementations are limited to 32 bit address space. Note that, at the time we were performing the experiments, the server ran multiple concurrent processes, such that the times may vary in different runs.

DOI: 10.1002/spe

Table VI. Average virtual memory space consumption per input character for the different suffix array construction algorithms.

Bytes per input character							
bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
9.22	5.08	6.06	5.06	8.05	5.96	15.88	31.45

4.2. Space consumption

Besides the running times, we measured the space consumptions of the different suffix array construction algorithms over all data files. We used *memtime* [23] to get the peak virtual memory consumption traced by the linux operating system. Table VI shows the results in average number of bytes per character of the used input sequences.

With 5.06n to 6.06n bytes, the lightweight algorithms *copy*, *deep shallow*, *difference-cover*, and *cache* use slightly more space than the theoretical minimum of 5n bytes, consisting of 4n bytes for the suffix array and n bytes for the input string. *qsufsort's* 8.05n and *bpr's* 9.22n bytes are still under the limit of 10n bytes, while *divide & conquer* and *skew* using 15.88n and 31.45n bytes, respectively, consume significantly more space.

4.3. Detailed runtime analysis

For a more detailed performance analysis of the suffix array construction algorithms, we used the profiler and cache simulator *valgrind* [24] to count the number of executed instructions and to simulate the caching behaviour.

The number of executed instructions of the different algorithms is shown in Table VII, the L1 data references in Table VIII, the L1 misses, or alternatively, L2 references in Table IX, and the number of L2 misses in Table X. We stopped the computation whenever a simulation used more than 24 h, which is indicated by a dash in the tables. In addition, Figures 2 and 3 exemplarily show bar charts for *H. sapiens chromosome 22* and the *linux* source code. Note that besides the instructions and cache references of the pure suffix array construction algorithms, *valgrind* also counts those of the different IO routines for reading the input strings from the disk.

It is impressive that the instruction counts for *bpr* clearly outperform all other algorithms for all but one string, the artificial string with period length 20 for which *divide & conquer* and *skew* take fewer instructions. For real-world strings, the second best algorithm, *deep shallow*, executes on average more than twice as many instructions, and it is more sensitive with respect to higher average LCP. For periodic strings, *deep shallow* takes an escalating number of instructions. In contrast, *bpr* is stable with respect to high average LCP. Even for periodic strings, the average instruction count of *bpr* is comparable with the linear-time algorithm *skew* and the quasi-linear *divide & conquer* algorithm.

The caching behaviour of *bpr* is not as optimal as we expected. Although it takes the smallest number of L1 cache references for all but the *bible* and *period 20* strings, its inferior miss ratio often leads to

DOI: 10.1002/spe

Table VII. Number of executed instructions.

		Executed instructions (thousand)							
Sequence type	Sequence	*bpr*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
DNA sequence	*E. coli* genome	590 968	1 116 001	3 048 715	2 924 944	1 447 669	4 169 253	2 007 812	1 793 770
	A. thaliana chr. 4	1 577 214	2 975 394	10 840 397	10 391 687	4 076 698	11 488 842	5 224 565	4 764 062
	H. sapiens chr. 22	5 054 919	9 077 493	28 489 800	26 490 978	11 816 047	37 186 250	15 266 866	13 760 985
	C. elegans chr. 1	1 766 219	4 082 728	28 938 600	22 816 333	5 746 697	16 212 057	6 327 318	5 615 001
	6 *Streptococci*	1 605 429	4 264 965	14 712 656	10 621 542	4 905 627	12 411 770	5 067 023	4 543 454
	4 *Chlamydophila*	840 548	3 678 525	34 218 426	22 160 559	2 585 444	5 344 131	2 113 043	1 911 102
	3 *E. coli*	2 484 036	14 478 491	1 016 365 699	2 191 651 989	9 904 217	16 992 486	6 469 187	5 876 152
Text	*bible*	774 557	1 018 210	2 520 999	2 477 103	1 481 335	4 056 576	1 868 161	1 491 339
	world192	444 263	622 208	1 524 604	1 582 120	874 543	2 815 562	1 138 542	910 405
	rfc	11 349 518	18 651 462	50 950 884	66 425 234	22 048 783	63 936 958	24 977 285	19 291 184
	sprot34	11 332 660	23 514 808	76 297 396	92 711 305	21 206 686	70 779 027	23 969 935	19 247 803
	howto	7 941 501	13 166 719	29 340 824	59 265 649	16 517 644	43 743 216	18 918 419	15 758 944
	reuters	14 099 318	48 711 948	146 182 468	229 252 426	23 136 951	79 088 119	25 489 119	19 593 772
	w3c2	13 472 295	82 101 938	99 424 334	258 178 856	23 430 229	85 262 620	25 681 277	19 533 352
	jdk13	14 038 110	49 319 246	134 990 795	423 915 692	23 558 497	85 567 980	25 675 083	19 500 321
	linux	10 162 913	18 854 491	50 234 557	149 806 756	23 903 435	64 257 123	24 439 748	19 906 506
	etext99	11 063 320	22 007 979	47 465 418	416 485 942	22 307 806	53 537 497	23 962 778	20 053 024
	gcc	15 993 418	126 675 467	7 095 650 010	—	23 090 215	72 258 976	24 885 116	19 939 060
Artificial	random	2 975 030	5 495 448	10 497 086	10 333 982	5 270 092	16 373 218	7 594 978	5 693 778
	period 500 000	4 893 746	1 307 962 759	—	—	15 000 996	41 807 848	7 649 106	7 781 054
	period 1 000	3 803 035	2 064 251 713	—	—	15 206 346	44 399 867	7 888 927	7 872 762
	period 20	11 395 604	—	—	—	16 802 830	49 033 908	9 212 229	7 618 039
	Fibonacci string	7 091 709	1 083 038 179	—	—	15 706 027	48 193 424	11 590 507	7 657 301

DOI: 10.1002/spe

Table VIII. Number of L1 cache references.

			L1 data cache references (thousand)						
Sequence type	Sequence	*bpr*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
DNA sequence	*E. coli* genome	315 960	390 145	1 417 732	1 514 313	722 751	2 161 443	1 200 158	1 061 180
	A. thaliana chr. 4	811 668	1 037 317	5 385 296	5 977 490	2 005 570	5 955 907	3 107 695	2 818 054
	H. sapiens chr. 22	2 621 604	3 110 712	13 401 675	14 173 009	5 830 116	19 541 493	8 945 029	8 141 712
	C. elegans chr. 1	914 101	1 583 436	16 289 055	15 172 626	2 834 950	8 576 998	3 684 766	3 323 156
	6 *Streptococci*	831 339	1 684 693	7 799 907	6 256 366	2 422 510	6 218 415	3 012 958	2 689 070
	4 *Chlamydophila*	421 644	1 646 859	20 210 568	15 802 123	1 282 902	2 613 074	1 256 153	1 130 888
	3 *E. coli*	1 308 183	7 114 216	611 666 511	1 627 469 385	4 915 532	8 341 563	3 840 080	3 475 480
Text	*bible*	406 461	376 163	1 163 298	1 268 490	731 942	2 027 115	1 038 776	882 752
	world192	232 912	236 954	731 716	873 982	437 246	1 434 013	635 910	538 872
	rfc	6 055 629	7 290 350	26 000 246	42 206 854	10 734 274	32 815 923	13 250 700	11 420 932
	sprot34	6 031 435	8 798 704	41 016 605	61 492 481	10 423 755	37 730 780	13 053 571	11 398 901
	howto	4 185 612	5 048 757	13 893 174	37 915 444	8 048 669	21 770 064	10 297 684	9 314 364
	reuters	7 593 193	19 954 465	81 970 742	161 985 424	11 378 689	42 758 079	13 351 247	11 608 824
	w3c2	7 252 083	41 998 735	54 302 218	184 293 769	11 598 038	45 809 587	13 379 028	11 577 033
	jdk13	7 555 931	20 207 358	75 418 755	307 107 639	11 701 574	46 736 290	13 378 402	11 562 449
	linux	5 521 524	7 471 093	25 663 673	103 994 819	11 621 838	32 517 373	13 155 616	11 773 550
	etext99	5 853 319	8 447 063	22 866 943	300 615 222	10 819 095	26 899 839	13 045 597	11 853 018
	gcc	8 909 280	64 505 787	4 275 774 325	—	11 351 015	38 131 061	13 234 281	11 801 330
Artificial	random	1 530 409	1 909 155	4 696 771	4 618 954	2 733 712	8 358 796	4 659 477	3 385 913
	period 500 000	2 549 647	778 806 712	—	—	7 753 145	23 845 004	4 747 209	4 624 488
	period 1 000	1 997 493	1 058 027 759	—	—	7 678 600	25 638 336	4 816 024	4 706 577
	period 20	5 824 351	—	—	—	8 239 241	28 628 901	5 508 884	4 515 564
	Fibonacci string	3 804 577	644 630 685	—	—	8 032 388	28 346 724	6 501 889	4 544 172

DOI: 10.1002/spe

Table IX. Number of L1 cache misses, or alternatively, L2 cache references.

		L1 cache misses (thousand)							
Sequence type	Sequence	*bpr*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
DNA sequence	*E. coli* genome	25 552	22 750	34 669	27 001	32 974	231 628	72 440	146 963
	A. thaliana chr. 4	82 813	64 487	102 685	77 563	104 643	231 628	190 843	391 901
	H. sapiens chr. 22	263 348	208 974	324 054	252 211	302 949	736 584	536 723	1 101 259
	C. elegans chr. 1	85 120	73 273	200 063	162 155	143 677	264 567	208 204	426 806
	6 *Streptococci*	83 021	73 343	122 604	88 793	146 658	234 025	185 476	370 137
	4 *Chlamydophila*	37 789	39 424	157 369	111 068	87 457	92 233	76 464	154 788
	3 *E. coli*	133 774	171 457	4 934 361	12 364 222	343 281	316 081	237 797	481 498
Text	*bible*	30 023	19 355	29 400	22 464	39 778	67 386	79 079	124 044
	world192	15 614	10 141	15 673	11 472	23 478	37 772	47 921	74 697
	rfc	530 937	354 540	482 414	411 912	746 458	1 190 125	1 057 183	1 578 645
	sprot34	650 955	451 872	575 495	453 905	768 979	1 458 680	1 079 116	1 591 473
	howto	352 315	251 729	353 580	448 174	530 641	801 628	863 326	1 353 305
	reuters	851 567	770 883	833 980	803 472	916 619	1 875 046	1 123 812	1 592 107
	w3c2	780 447	605 451	688 276	1 024 135	915 366	1 748 600	1 151 603	1 553 724
	jdk13	812 871	808 250	783 493	1 419 869	938 076	1 855 852	1 089 554	1 531 613
	linux	447 413	301 789	437 131	860 558	810 693	1 038 691	1 028 326	1 635 774
	etext99	544 000	405 860	560 245	2 260 457	766 160	1 076 752	1 096 127	1 756 199
	gcc	963 115	1 499 266	25 889 834	—	803 007	1 206 890	948 605	1 555 627
Artificial	random	142 861	121 016	153 138	131 819	136 937	368 134	400 908	499 383
	period 500 000	192 045	12 110 399	—	—	931 467	554 347	353 680	623 189
	period 1000	296 405	14 425 652	—	—	1 060 907	1 341 299	298 064	535 734
	period 20	1 208 572	—	—	—	1 143 283	740 147	102 591	488 128
	Fibonacci string	511 027	11 032 995	—	—	951 990	905 678	217 466	428 954

DOI: 10.1002/spe

Table X. Number of L2 cache misses.

Sequence type	Sequence	L2 cache misses (thousand)								
		bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
DNA sequence	*E. coli* genome	11 457	11 355	18 571	12 205	22 652	38 041	53 940	127 920
	A. thaliana chr. 4	31 659	36 285	59 162	41 093	62 491	129 410	156 698	359 369
	H. sapiens chr. 22	90 296	126 741	204 391	151 006	211 124	447 961	467 784	1 038 501
	C. elegans chr. 1	37 212	43 176	113 762	69 365	112 818	153 238	172 002	389 561
	6 *Streptococci*	35 932	44 546	80 741	51 382	106 906	135 290	153 082	341 193
	4 *Chlamydophila*	18 906	22 453	100 552	57 958	72 749	47 106	56 778	132 813
	3 *E. coli*	70 309	124 041	4 425 379	10 701 120	313 477	193 982	200 688	444 152
Text	*bible*	10 991	7 334	12 579	7 595	22 952	30 800	55 001	108 376
	world192	5 847	2 870	5 320	2 851	14 586	13 791	30 034	59 444
	rfc	223 540	171 034	265 918	199 120	473 103	681 694	822 048	1 479 736
	sprot34	295 917	179 474	296 792	202 541	483 257	768 487	841 154	1 468 657
	howto	148 096	129 111	193 106	164 491	320 056	476 646	691 055	1 261 786
	reuters	523 073	341 164	453 604	356 515	642 506	1 183 388	872 733	1 487 177
	w3c2	360 861	252 132	337 474	293 359	626 185	875 117	812 437	1 438 407
	jdk13	400 996	247 713	398 723	401 730	665 131	885 281	801 261	1 431 649
	linux	190 276	145 165	216 520	211 462	517 424	600 023	785 128	1 517 618
	etext99	213 528	235 346	335 026	856 837	473 414	682 519	920 192	1 655 538
	gcc	289 923	272 645	9 755 584	—	535 168	676 715	681 651	1 447 402
Artificial	random	52 416	45 960	69 702	49 224	114 700	225 137	351 088	433 712
	period 500 000	97 270	6 375 039	—	—	895 214	404 486	296 241	568 378
	period 1000	179 257	13 882 918	—	—	903 421	990 147	220 091	493 428
	period 20	962 914	—	—	—	1 034 323	614 043	101 918	482 019
	Fibonacci string	386 494	10 880 356	—	—	890 734	737 962	199 586	416 624

DOI: 10.1002/spe

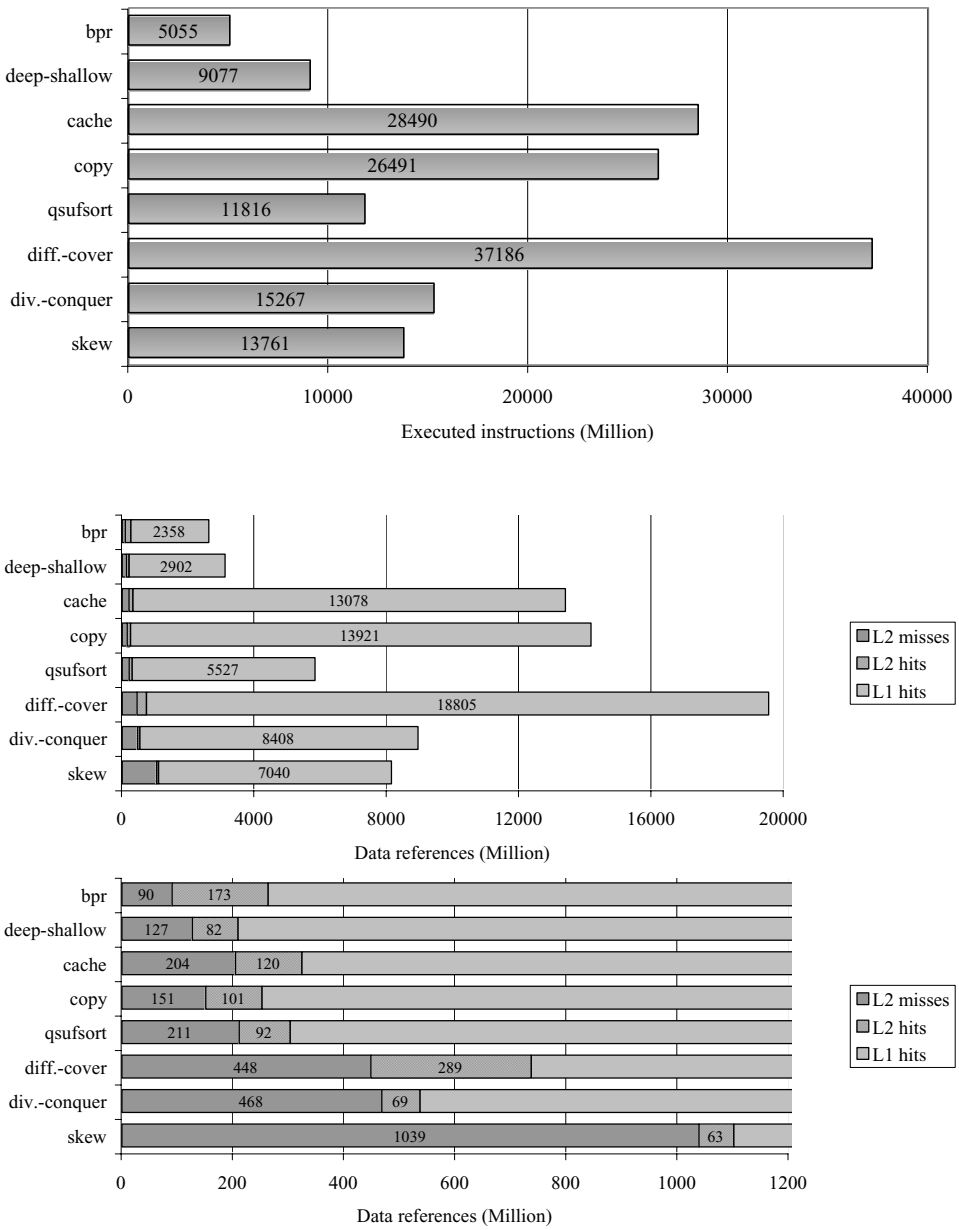

Figure 2. Instruction counts and cache references for *H. sapiens chr. 22*, with $d = 7$ for *bpr*.

DOI: 10.1002/spe

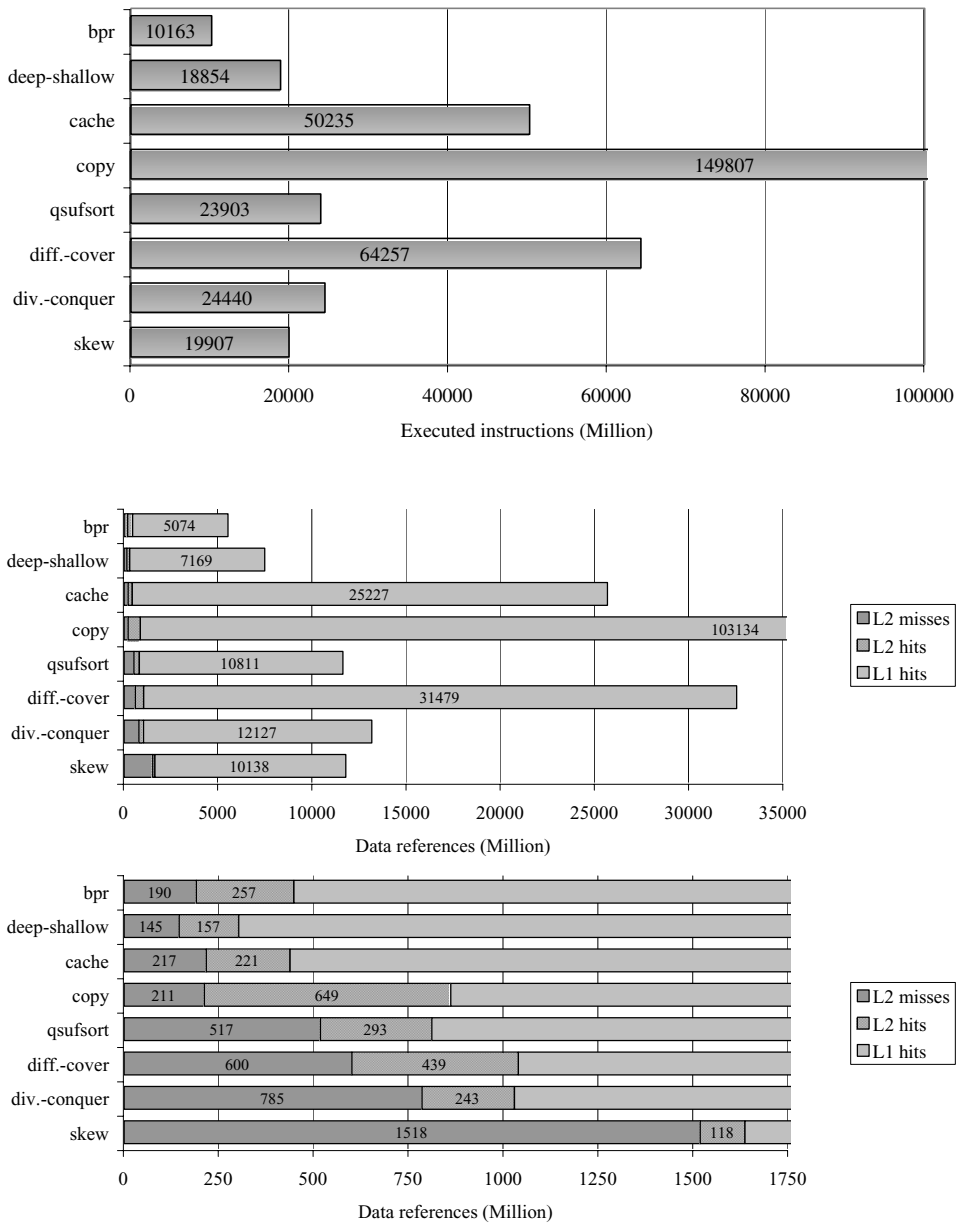

Figure 3. Instruction counts and cache references for the *linux* file, with $d = 3$ for *bpr*.

DOI: 10.1002/spe

more cache misses. For DNA sequences, *bpr* still has the fewest L2 misses, but for other real world strings *deep shallow*, *cache*, and *copy* often have less L2 misses. For strings with periods of length 500 000 and 1000, *bpr* takes the fewest cache misses. Only for the string with a period shorter than 20 and for the Fibonacci string, *divide & conquer* and *skew* have fewer cache misses.

4.4. Discussion

We first believed that the practical speed of our algorithm was mainly due to the combination of different techniques with good locality behaviour. However, the simulations showed that, compared to the other suffix array construction algorithms, *bpr* mainly gains its fast running time from the fewer executed instructions rather than from its good locality behaviour. Hence, with respect to the number of executed instructions, *bpr* is the algorithmically best algorithm.

The few executed instructions are apparently due to the different strategies of the two phases of the *bpr* algorithm. First of all, if the d-length substrings are uniformly distributed, phase 1 equally divides all suffixes into small buckets by just scanning the input string twice. However, this does not explain its speed for the periodic strings. Here, the suffixes are just partitioned into a few large buckets. For such strings, our algorithm basically benefits from the employment of relations among the suffixes in phase 2. By using the bucket pointers as sort keys, the method incorporates information about the subdivided buckets into the bucket refinement process as soon as this information becomes available. In the bucket-refinement process each bucket is refined recursively until it consists of singleton sub-buckets. This technique of dividing suffixes from small to smaller buckets is similar to *Quicksort* for original sorting, which is known to be fast in practice. The combination of these techniques, further heuristics in the refinement procedure (Section 3.3), and Seward's *copy* method [13] result in the final low instruction count.

In our first assumptions that the good locality behaviour was mainly responsible for the speed of *bpr*, we were misled by some elements of the algorithm that have good locality behaviour with respect to the data structure, but this is not always the case. The data structure can be divided into four parts: the input string, the suffix array, the *bptr* array, and the bucket array storing the boundaries for all buckets. Phase 1, for example, just scans the sequence twice. It has a good locality of memory access with respect to the input string and the *bptr* array, whereas the bucket array and the suffix array are arbitrarily accessed. In contrast, phase 2 has a good locality of memory access with respect to the bucket array and the suffix array. The bucket array is accessed from left to right and the suffix array is divided into increasingly smaller buckets. The *bptr* array is again arbitrarily accessed.

Therefore, *bpr*'s cache miss ratio is generally worse than that of *deep shallow*, *cache*, and *copy*. Even the linear-time *skew* and the quasi-linear *divide & conquer*, from which one could expect that they trade locality of memory access against good worst-case time complexity, show comparable cache miss ratios. Nevertheless, thanks to its fewer total cache accesses and its fewer executed instructions, *bpr* is generally faster than the other algorithms.

The instruction counts for the different real world strings of length 50 million reveal further interesting facts. The linear-time *skew*, the quasi-linear *divide & conquer*, and the $O(n \log n)$ time *qsufsort* algorithms show little variance of instruction counts indicating little dependence on the sequence structure. In contrast, *deep shallow*, *cache*, and *copy's* instruction counts vary greatly. *Deep-shallow*, for example, executes less than 19 billion instructions for the *rfc* and *linux* files, but more than 82 billion instructions for *w3c2* and *gcc*. For *gcc*, the very high average and maximum LCP

DOI: 10.1002/spe

values account for the high instruction count, whereas for *w3c2* this is not so. The string has even lower LCP values than *linux*, nevertheless *deep shallow* needs more than four times the number of executed instructions. Therefore, other structural properties of the text also seem to be important for the instruction count, and thus for the performance of these algorithms.

Moreover, for the strings of length 50 million, *deep shallow's* instruction count is often related to *cache's*. The fact that *deep shallow* uses the method of *cache* as a subprocedure suggests that its performance highly depends on the *cache* method.

Comparing the instruction counts for the 50 million character strings shows that *deep shallow* often executes many more instructions than *qsufsort*, *divide* & *conquer*, or *skew*, even though its execution time is always significantly faster. The higher number of L2 cache misses for *qsufsort*, *divide* & *conquer*, and in particular *skew* reveal that the fragmented memory access slows down their suffix array construction. Therefore, the practically fastest algorithm does not need to have the lowest instruction count or the lowest number of cache misses, but as with *bpr*, it must possess the optimal combination of both properties.

However, the space requirements of *bpr* are higher than the space requirements for *deep shallow*, *cache*, and *copy*. In practice, *bpr* takes between $9n$ and $10n$ bytes, the suffix array and the bucket pointer table each consume $4n$ bytes, and the input string n bytes. Additional space is used for the bucket pointers of the initial bucket sort and for the recursion stack, even though the recursion depth decreases by a factor of d. However, for certain applications, such as the computation of the Burrows–Wheeler Transform [16], the construction of the suffix array is just a byproduct, and the complete suffix array does not need to remain in memory.

Therefore, if one is concerned about space, the *deep shallow* algorithm might be the best choice. If there are no major space limitations, we believe that the *bpr* algorithm is an attractive alternative.

5. CONCLUSION AND FURTHER WORK

We have presented a fast suffix array construction algorithm that performs very well even for worst-case strings. Due to its simple structure, it is easy to implement. Therefore, we believe that our algorithm can be widely used in all kinds of suffix array applications.

An open question remains. We were so far unable to prove a better worst-case time complexity than $O(n^2)$ while at the same time we are not aware of an example showing that this bound is tight. For certain periodic strings, we verified an $O(n^{3/2}\sqrt{\log n})$ time bound, but for general strings finding a non-trivial upper bound seems to be hard since our algorithm quite arbitrarily uses the dependence among suffixes.

Of further interest will be the parallelization of suffix array construction, since the suffix array construction for very large DNA sequences is usually performed on servers with more than one CPU.

ACKNOWLEDGEMENTS

We wish to thank Dong Kyue Kim for providing the code of the *divide* & *conquer* algorithm, Peter Husemann for implementing the timing for the *skew* algorithm, Hans-Michael Kaltenbach for advice on the analysis of the *bpr* algorithm, and Sita Lange for carefully proofreading this manuscript.

DOI: 10.1002/spe

REFERENCES

1. Manber U, Myers EW. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* 1993; **22**(5):935–948.
2. Karp RM, Miller RE, Rosenberg AL. Rapid identification of repeated patterns in strings, trees and arrays. *Proceedings of the 4th ACM Symposium on Theory of Computing (STOC 1972)*. ACM Press: New York, 1972; 125–136.
3. Farach-Colton M, Ferragina P, Muthukrishnan S. On the sorting-complexity of suffix tree construction. *Journal of the ACM* 2000; **47**(6):987–1011.
4. Kurtz S. Reducing the space requirements of suffix trees. *Software: Practice and Experience* 1999; **29**(13):1149–1171.
5. Kärkkäinen J, Sanders P. Simple linear work suffix array construction. *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)* (*Lecture Notes in Computer Science*, vol. 2719). Springer: Berlin, 2003; 943–955.
6. Kim DK, Sim JS, Park H, Park K. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms* 2005; **3**(2–4):126–142.
7. Ko P, Aluru S. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms* 2005; **3**(2–4): 143–156.
8. Hon W-K, Sadakane K, Sung W-K. Breaking a time-and-space barrier in constructing full-text indices. *Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003)*. IEEE Computer Society: Los Alamitos, CA, 2003; 251–260.
9. Larsson NJ, Sadakane K. Faster suffix sorting. *Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999)*, Department of Computer Science, Lund University, May 1999.
10. Kim DK, Jo J, Park H. A fast algorithm for constructing suffix arrays for fixed-size alphabets. *Proceedings of the 3rd International Workshop on Experimental and Efficient Algorithms* (*Lecture Notes in Computer Science*, vol. 3059), Ribeiro CC, Martins SL (eds.). Springer: Berlin, 2004; 301–314.
11. Farach M. Optimal suffix tree construction with large alphabets. *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science (FOCS 1997)*, October 1997; 137–143.
12. Itoh H, Tanaka H. An efficient method for in memory construction of suffix arrays. *Proceedings of String Processing and Information Retrieval Symposium and International Workshop on Groupware (SPIRE/CRIWG 1999)*. IEEE Computer Society Press: Los Alamitos, CA, 1999; 81–88.
13. Seward J. On the performance of BWT sorting algorithms. *Proceedings of the Data Compression Conference (DCC 2000)*. IEEE Computer Society: Los Alamitos, CA, 2000; 173–182.
14. Manzini G, Ferragina P. Engineering a lightweight suffix array construction algorithm. *Algorithmica* 2004; **40**(1):33–50.
15. Burkhardt S, Kärkkäinen J. Fast lightweight suffix array construction and checking. *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)* (*Lecture Notes in Computer Science*, vol. 2676). Springer: Berlin, 2003; 55–69.
16. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. *Technical Report Research Report 124*, Digital System Research Center, May 1994.
17. Bentley JL, McIlroy MD. Engineering a sort function. *Software: Practice and Experience* 1993; **23**(11):1249–1265.
18. Bentley JL, Sedgewick R. Fast algorithms for sorting and searching strings. *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1997)*. Society for Industrial and Applied Mathematics: Philadelphia, PA, 1997; 360–369.
19. McIlroy PM, Bostic K, McIlroy MD. Engineering radix sort. *Computing Systems* 1993; **6**(1):5–27.
20. Cormen TH, Leiserson CE, Rivest RL. *Introduction to Algorithms* (1st edn). MIT Press: Cambridge, CA, 1989.
21. Singleton RC. ACM Algorithm 347: An efficient algorithm for sorting with minimal storage. *Communications of the ACM* 1969; **12**(3):185–187.
22. Schürmann K-B. Bpr Home. http://bibiserv.techfak.uni-bielefeld.de/bpr/ [20 June 2006].
23. Bengtsson J. Project details for memtime. http://freshmeat.net/projects/memtime [20 June 2006].
24. Seward J, Nethercote N, Fitzhardinge J *et al*. Valgrind Home. http://valgrind.org [20 June 2006].

DOI: 10.1002/spe