Fast Lightweight Suffix Array Construction and Checking

Stefan Burkhardt[1⋆] and Juha Kärkkäinen[1⋆]

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
[stburk,juha]@mpi-sb.mpg.de

Abstract. We describe an algorithm that, for any $v \in [2, n]$, constructs the suffix array of a string of length n in $\mathcal{O}(vn + n \log n)$ time using $\mathcal{O}(v + n/\sqrt{v})$ space in addition to the input (the string) and the output (the suffix array). By setting $v = \log n$, we obtain an $\mathcal{O}(n \log n)$ time algorithm using $\mathcal{O}\big(n/\sqrt{\log n}\big)$ extra space. This solves the open problem stated by Manzini and Ferragina [ESA '02] of whether there exists a lightweight (sublinear extra space) $\mathcal{O}(n \log n)$ time algorithm. The key idea of the algorithm is to first sort a sample of suffixes chosen using mathematical constructs called difference covers. The algorithm is not only lightweight but also fast in practice as demonstrated by experiments. Additionally, we describe fast and lightweight suffix array checkers, i.e., algorithms that check the correctness of a suffix array.

1 Introduction

The suffix array [21, 9], a lexicographically sorted array of the suffixes of a string, has numerous applications, e.g., in string matching [21, 9], genome analysis [1] and text compression [5]. In many cases, the construction of the suffix array is a bottleneck. As suffix arrays are often generated for very long strings, both space and time requirement matter.

In a typical situation, a string of length n occupies n bytes and its suffix array $4n$ bytes of space. The suffix array can be constructed using little extra space, but then the worst-case running time is $\Omega(n^2)$. All fast construction algorithms with runtime guarantee $\mathcal{O}(n \log n)$ or better require at least $4n$ bytes of extra space, which almost doubles the space requirement. Manzini and Ferragina [22] asked whether it is possible to achieve $\mathcal{O}(n \log n)$ runtime using sublinear extra space. In this paper, we answer the question positively describing an algorithm with $\mathcal{O}(n \log n)$ worst-case runtime that uses only $\mathcal{O}\big(n/\sqrt{\log n}\big)$ extra space.

Previous Work. In addition to time and space requirements, suffix array construction algorithms differ in their alphabet model. The models of interest, in order from the most restrictive to the most general, are constant alphabet (the

⋆ Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

size of the alphabet is bounded by a constant), integer alphabet (characters are integers in a range of size $n^{\mathcal{O}(1)}$), and general alphabet (only character comparisons are allowed).

Previous suffix array construction algorithms can be classified into four main categories.

The algorithms in the first category compute the suffix array from the suffix tree in linear time. The classical suffix tree construction algorithms [28, 23, 26] work in linear time for constant alphabets. Farach's algorithm [8] achieves linear time for integer alphabets. The drawback of these algorithms is their space requirement. The most space efficient implementation by Kurtz [18] uses $8n$–$14n$ bytes of space in total, but this comes at the cost of limiting the maximum string length to 135 million characters.

The second category, direct linear time construction algorithms, has appeared very recently [16, 17, 11]. It is not yet clear what the space requirements of practical implementations are but all of them appear to require at least $4n$ bytes of extra space and likely more. One of these algorithms [11] is, in fact, closely related to the present algorithm as described in Section 9. All of these algorithms support integer alphabets.

The algorithms in the third category are based on the doubling algorithm of Karp, Miller and Rosenberg [13]. These algorithms sort the suffixes initially by their first characters and then double the significant prefix length in each further pass. The first algorithm in this category was by Manber and Myers [21], but the Larsson-Sadakane algorithm [19] is considered to be the best in practice. Both algorithms run in $\mathcal{O}(n \log n)$ time, need $4n$ bytes of extra space, and support general alphabets.

The final category consists of algorithms based on sorting the suffixes as independent strings. For most real world inputs, string sorting is very fast [19] and it needs little extra space. Furthermore, Itoh and Tanaka [10] as well as Seward [24] reduce the number of suffixes to be sorted to about half by taking advantage of the correlation between suffixes starting at consecutive positions. However, the worst case running time of these algorithms is $\Omega(n^2)$ and they can also be very slow on real world inputs with long repeats [22]. Basic string sorting supports general alphabets but the Itoh–Tanaka, Seward, and Manzini–Ferragina (below) heuristics assume a constant alphabet.

The string sorting based algorithms are attractive in practice since they are both the fastest in most cases and use little space, but the possibility of quadratic running time is unacceptable for many applications. The solution adopted by the `bzip2` compression package [25] implementing Seward's algorithm uses the Larsson–Sadakane algorithm as a fallback when the primary algorithm takes too much time. However, in such cases, the algorithm is slow and, more importantly, needs $4n$ bytes of extra space. The algorithms of Manzini and Ferragina [22] sort the suffixes using only their first ℓ characters and apply fallback heuristics to the groups of suffixes that remain unsorted. The algorithms use little extra space (less than $0.03n$ bytes), are fast even for many difficult cases, but the worst case running times remain $\Omega(n^2)$.

Our Contribution We describe an algorithm that runs in $\mathcal{O}(vn + n \log n)$ time using $\mathcal{O}(v + n/\sqrt{v})$ extra space for any $v \in [2, n]$. Thus, the choice of v offers a space–time tradeoff. Moreover, setting $v = \log n$ gives an $\mathcal{O}(n \log n)$ time, $\mathcal{O}(n/\sqrt{\log n})$ extra space algorithm, the first $\mathcal{O}(n \log n)$ time algorithm using sublinear extra space. The algorithm is alphabet-independent, i.e., it supports general alphabets. Note that $\mathcal{O}(n \log n)$ time is optimal under general alphabets.

The key to the result are mathematical constructs called *difference covers* (see Section 2). Difference covers are a powerful tool that can be used to obtain a number of other results, too. We give a brief glimpse to such results in Section 9. Difference covers have also been used for VLSI design [15] and distributed mutual exclusion [20, 7].

Experiments show that the algorithm is also practical. A straightforward implementation using less than n bytes of extra space is competitive with the Larsson-Sadakane algorithm [19], which is considered the best among the good worst case algorithms. The algorithm of Manzini and Ferragina [22], probably the fastest string sorting based implementation available, is significantly faster and slightly more space efficient on real world data but is unusably slow on worst case data. We believe that an improved implementation of our algorithm can bring its running time close to that of Manzini–Ferragina even on real world data (see Section 6).

Verifying the correctness of an implementation of an algorithm is a difficult, often impossible task. A more modest but still useful guard against incorrect implementation is a *result checker* [4, 27] that verifies the output of a computation. A *suffix array checker* verifies the correctness of a suffix array. The trivial suffix array checker has an $\mathcal{O}(n^2)$ worst case running time. In Section 8, we describe some simple, fast and lightweight checkers. These checkers are not directly related to our main result, but we found them useful during the implementation of the difference cover algorithm.

2 Basic Idea

Our algorithm is based on the following simple observation. Suppose we want to compare two suffixes S_i and S_j (i.e., suffixes starting at positions i and j) to each other. If we can find another pair S_{i+k} and S_{j+k}, called an *anchor pair*, of suffixes whose relative order is already known, then at most the first k characters of S_i and S_j need to be compared. Some of the fallback heuristics of Manzini and Ferragina [22] rely on this observation. Unlike their methods, our algorithm gives a guarantee of finding an anchor pair with a small offset k for any pair of suffixes. This is achieved by sorting first a *sample* of suffixes, and then finding the anchor pairs among the sample.

The question is how to choose such a sample.

An initial attempt might be to take every vth suffix for some v. However, for any pair of suffixes S_i and S_j such that $i \not\equiv j \pmod{v}$, there would be no anchor pairs at all. A second try could be a *random* sample of size $n/\sqrt{v}$, which makes expected distance to an anchor pair $\mathcal{O}(v)$. The problem is that there are no time and space efficient algorithms for sorting an arbitrary set of suffixes. The alternatives are basically general string sorting and full suffix array construction, neither of which is acceptable in our case.

Our solution is based on using difference covers. A *difference cover* D modulo v is a set of integers in the range $[0, v)$ such that for all $i \in [0, v)$, there exist $j, k \in D$ such that $i \equiv k - j \pmod v$. For example, $D = \{1, 2, 4\}$ is a difference cover modulo 7:

i	0	1	2	3	4	5	6
$k-j$	$1-1$	$2-1$	$4-2$	$4-1$	$1-4$	$2-4$	$1-2$

The sample we use contains all suffixes whose starting positions are in D modulo v. The key properties of the sample are:

- The size of the sample is $\mathcal{O}(n/\sqrt{v})$ (see Section 3).
- The sample guarantees that an anchor pair is found within distance v (see Section 4).
- The periodicity of the sample (with period v) makes it possible to sort the sample efficiently (see Section 5).

3 Tools

Let $s[0, n)$ be a string of length n over a general alphabet, i.e., we assume that the characters can be compared in constant time but make no other assumptions on the alphabet. Let S_i, $i \in [0, n)$, denote the suffix $s[i, n)$. A set of suffixes is *v-ordered* if they are ordered by their first v characters, i.e., as if the suffixes longer than v were truncated to length v.

Lemma 1. *A set of m suffixes (represented by an array of their starting positions) can be v-ordered in $\mathcal{O}(vm + m \log m)$ time using $\mathcal{O}(\log m)$ extra space.*

Proof. Use the multikey quicksort algorithm of Bentley and Sedgewick [3] that can sort m strings of total length M in time $\mathcal{O}(M + m \log m)$. Only the recursion stack requires non-constant extra space.

The suffix array $\mathrm{SA}[0, n)$ of s is a permutation of $[0, n)$ specifying the lexicographic order of the suffixes of s, i.e., for all $0 \le i < j < n$, $S_{\mathrm{SA}[i]} < S_{\mathrm{SA}[j]}$. The inverse suffix array $\mathrm{ISA}[0, n)$ is the inverse permutation of SA, i.e., $\mathrm{ISA}[i] = j$ if and only if $\mathrm{SA}[j] = i$. The inverse suffix array constitutes a *lexicographic naming* of the suffixes allowing constant time comparisons: $S_i \le S_j$ if and only if $\mathrm{ISA}[i] \le \mathrm{ISA}[j]$. The suffix array and its inverse can be computed quickly with the algorithms of Manber and Myers [21] or Larsson and Sadakane [19].

Lemma 2 ([21, 19]). *The suffix array* SA *and the inverse suffix array* ISA *of a string s of length n can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.*

A difference cover D modulo v is a set of integers in the range $[0, v)$ such that for all $i \in [0, v)$, there exists $j, k \in D$ such that $i \equiv k - j \pmod{v}$. It is easy to see that a difference cover modulo v must be of size at least $\sqrt{v}$. A simple method for generating small difference covers is described by Colbourn and Ling [7]:

Lemma 3 ([7]). *For any v, a difference cover modulo v of size $\leq \sqrt{1.5v} + 6$ can be computed in $\mathcal{O}(\sqrt{v})$ time.*

For any integers i, j, let $\delta(i, j)$ be an integer $k \in [0, v)$ such that $(i+k) \bmod v$ and $(j+k) \bmod v$ are both in D.

Lemma 4. *Given a difference cover D modulo v, a data structure allowing constant time evaluation of $\delta(i, j)$ for any i and j can be computed in $\mathcal{O}(v)$ time and space.*

Proof. Build a lookup table d such that, for all $h \in [0, v)$, both $d[h]$ and $(d[h] + h) \bmod v$ are in D, and implement δ as $\delta(i, j) = (d[(j-i) \bmod v] - i) \bmod v$. Then $i + \delta(i, j) = d[(j-i) \bmod v] \in D$ and $j + \delta(i, j) = d[(j-i) \bmod v] + (j-1) \in D \pmod{v}$.

For the difference cover $D = \{1, 2, 4\}$ modulo 7, we have

h	0	1	2	3	4	5	6
$d[h]$	1	1	2	1	4	4	2

For example, let $i = 2$ and $j = 12$. Then we have $k = \delta(2, 12) = (d[10 \bmod 7] - 2) \bmod 7 = (1-2) \bmod 7 = 6$, and $(i + \delta(i, j)) \bmod 7 = (2+6) \bmod 7 = 1 \in D$ and $(j + \delta(i, j)) \bmod 7 = (12+6) \bmod 7 = 4 \in D$.

4 The Algorithm

The algorithm consists of the following phases:

Phase 0. Choose $v \geq 2$, a difference cover D modulo v with $|D| = \Theta(\sqrt{v})$, and compute the associated function δ.

Phase 1. Sort the suffixes whose starting position modulo v is in D.

Phase 2. Construct SA by exploiting the fact that, for any $i, j \in [0, n-v]$, the relative order of the suffixes starting at $i + \delta(i, j)$ and $j + \delta(i, j)$ is already known.

The implementation of Phase 0 was already explained in the previous section. Phase 1 is described in detail in the next section. In this section, we concentrate on Phase 2.

Let D be a difference cover modulo v. A D-sample D_n is the set $\{i \in [0, n] \mid i \bmod v \in D\}$. Let $m = |D_n| \leq (n/v + 1)|D|$. A lexicographic naming of the D-sample suffixes is a function $\ell : D_n \mapsto [0, m)$ satisfying $\ell(i) \leq \ell(j)$ if and only if $S_i \leq S_j$ (see Fig. 1). Phase 1 produces the function ℓ as its output. In the next section, we prove the following lemma.

Lemma 5. *A data structure allowing constant time evaluation of $\ell(i)$ for any $i \in D_n$ can be computed in $\mathcal{O}(\sqrt{v}n + (n/\sqrt{v})\log(n/\sqrt{v}))$ time using $\mathcal{O}(v + n/\sqrt{v})$ space in addition to the string s.*

Input:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$s[i]$	a		r	o	s	e		i	s		a		r	o	s	e		i	s		a		r	o	s	e

Result of phase 1:

$i \in D_{26}$	1	2	4	8	9	11	15	16	18	22	23	25
$\ell(i)$	3	8	11	10	0	2	5	1	9	7	6	4

Result of step 2.1:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$SA^7[i]$	9	19	6	16	21	1	11	20	0	10	25	5	15	7	17	23	3	13	22	2	12	8	18	24	4	14

Output:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$SA[i]$	19	9	16	6	21	11	1	20	10	0	25	15	5	17	7	23	13	3	22	12	2	18	8	24	14	4

Fig. 1. The difference cover algorithm with the difference cover $D = \{1, 2, 4\}$ modulo 7 applied to the string "`a rose is a rose is a rose`"

The essence of Phase 2 is the observation that the following comparisons are equivalent for all $i, j \in [0, n - v]$:

$$S_i \le S_j$$
$$\langle s[i, i+v), S_{i+\delta(i,j)} \rangle \le \langle s[j, j+v), S_{j+\delta(i,j)} \rangle$$
$$\langle s[i, i+v), \ell(i+\delta(i,j)) \rangle \le \langle s[j, j+v), \ell(j+\delta(i,j)) \rangle$$

Thus we can implement Phase 2 as follows.

Step 2.1. v-order the suffixes using multikey quicksort.

Step 2.2. For each group of suffixes that remains unsorted, i.e., shares a prefix of length v, complete the sorting with a comparison based sorting algorithm using $\ell(i + \delta(i, j))$ and $\ell(j + \delta(i, j))$ as keys when comparing suffixes S_i and S_j.

The operation of the algorithm is illustrated in Fig. 1. As an example of Step 2.2, consider the ordering of suffixes S_2 and S_{12}, which were not sorted in Step 2.1 because they share the prefix "`rose is`". We compute $\delta(2, 12) = 6$ and compare $\ell(2+6) = 10$ and $\ell(12+6) = 9$ to find out that S_{12} is smaller than S_2.

Lemma 6. *If δ and ℓ can be evaluated in constant time, the suffix array* $\mathrm{SA}[0, n)$ *of a string $s[0, n)$ can be constructed in $\mathcal{O}(vn + n \log n)$ time using $\mathcal{O}(\log n)$ space in addition to s,* SA*, and the space needed for δ and ℓ.*

Proof. The time complexity follows from Lemma 1. To keep the space requirement within $\mathcal{O}(\log n)$, Step 2.2 is performed for an unsorted group immediately when it is formed during Step 2.1. Then the only extra space is needed for the stack.

Combining Lemmas 3, 4, 5, and 6 we obtain the main result.

Theorem 1. *For any positive integer v, the suffix array of a string of length n can be constructed in $\mathcal{O}(vn + n \log n)$ time using $\mathcal{O}(v + n/\sqrt{v})$ space in addition to the string and the suffix array.*

By choosing $v = \log n$, we get:

Corollary 1. *The suffix array of a string of length n can be constructed in $\mathcal{O}(n \log n)$ time using $\mathcal{O}\big(n/\sqrt{\log n}\big)$ extra space.*

5 Sorting the Sample

The remaining part is to show how to sort the D-sample suffixes. More precisely, we want to compute the lexicographic naming function ℓ. Let $D = \{d_0, \ldots, d_{k-1}\}$ with $d_0 < d_1 < \cdots < d_{k-1} < d_k = v$, and let h be such that $d_h \leq n \bmod v < d_{h+1}$. Then $D_n = \{d_0, \ldots, d_{k-1}, d_0 + v, \ldots, d_{k-1} + v, d_0 + 2v, \ldots, d_0 + \lfloor n/v \rfloor v, \ldots, d_h + \lfloor n/v \rfloor v\}$. Let μ be the mapping $D_n \mapsto [0, m) : \mu(d_i + jv) = \lfloor n/v \rfloor i + \min(i, h) + j$, i.e., it maps each sequence $d_i, d_i + v, d_i + 2v, \ldots$ to consecutive positions (see Fig. 2). To compute $\mu(k)$, for $k \in D_n$, we need to find i and j such that $k = d_i + jv$. This is done by computing $j = \lfloor k/v \rfloor$, $d_i = k \bmod v$, and using a lookup table to compute i from d_i. Thus, the function μ can be evaluated in constant time after an $\mathcal{O}(v)$ time and space preprocessing.

Let ℓ^v be a lexicographic naming of the D-sample suffixes based on their first v characters, i.e., $\ell^v(i) \leq \ell^v(j)$ if and only if $s[i, \min(i+v, n)) \leq s[j, \min(j+v, n))$. Let $s'[0, m)$ be the string defined by $s'[\mu(i)] = \ell^v(i)$ for $i \in D_n$. Note that, for all $i \in (n - v, n] \cap D_n$, the character $s'[\mu(i)] = \ell^v(i)$ is unique and acts as a separator.

Let S'_i, $i \in [0, m)$, denote the suffixes of s', and let SA' and ISA' be the suffix array of s' and its inverse. The following inequalities are all equivalent for all $i, j \in D_n$.

$$
\begin{aligned}
\ell(i) &\leq \ell(j) \\
S_i &\leq S_j \\
s[i, n) &\leq s[j, n) \\
s[i, i+v) \cdot s[i+v, i+2v) \cdots &\leq s[j, j+v) \cdot s[j+v, j+2v) \cdots \\
\ell^v(i) \cdot \ell^v(i+v) \cdots &\leq \ell^v(j) \cdot \ell^v(j+v) \cdots \\
s'[\mu(i)] \cdot s'[\mu(i)+1] \cdots &\leq s'[\mu(j)] \cdot s'[\mu(j)+1] \cdots \\
S'_{\mu(i)} &\leq S'_{\mu(j)} \qquad \text{(Note: separators in } s'\text{)} \\
\mathrm{ISA}'[\mu(i)] &\leq \mathrm{ISA}'[\mu(j)]
\end{aligned}
$$

Input:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$s[i]$	a		r	o	s	e		i	s		a		r	o	s	e		i	s		a		r	o	s	e
$i \in D_{26}$		1	2		4				8	9		11				15	16		18				22	23		25
$\ell^v(i)$		2	7		9				8	0		2				4	1		8				6	5		3
$\mu(i)$		0	4		8				1	5		9				2	6		10				3	7		11

i	0	1	2	3	4	5	6	7	8	9	10	11
$s'[i]$	2	8	4	6	7	0	1	5	9	2	8	3
$ISA'[i]$	3	10	5	7	8	0	1	6	11	2	9	4

Output of Phase 1:

$i \in D_{26}$	1	2	4	8	9	11	15	16	18	22	23	25
$\ell(i)$	3	8	11	10	0	2	5	1	9	7	6	4

Fig. 2. Phase 1 of the algorithm with the difference cover $D = \{1, 2, 4\}$ modulo 7 applied to the string "`a rose is a rose is a rose`"

Thus we can implement $\ell(i)$ as $\text{ISA}'[\mu(i)]$.

The inverse suffix array ISA$'$ is computed as follows:

Step 1.1. v-order the D-sample suffixes using multikey quicksort.
Step 1.2. Compute $\ell^v(i)$ for all $i \in D_n$ by traversing the D-sample suffixes in lexicographic order and construct s' by setting $s'[\mu(i)] = \ell^v(i)$.
Step 1.3. Compute ISA$'$ using Manber-Myers or Larsson-Sadakane algorithm.

Proof (of Lemma 5). For D of size $\mathcal{O}(\sqrt{v})$, $m = \mathcal{O}(n/\sqrt{v})$. Thus, the first step takes $\mathcal{O}(vm + m \log m) = \mathcal{O}(\sqrt{v}n + (n/\sqrt{v}) \log(n/\sqrt{v}))$ time (Lemma 1). The second step requires $\mathcal{O}(m)$ and the third $\mathcal{O}(m \log m)$ time (Lemma 2). The space requirements are $\mathcal{O}(v)$ for implementing μ, $\mathcal{O}(m) = \mathcal{O}(n/\sqrt{v})$ for s', SA$'$, and ISA$'$, and $\mathcal{O}(\log m)$ for a stack.

6 Implementation

We implemented the difference cover algorithm in C++ using difference covers modulo powers of two for fast division (available through `http://www.mpi-sb.mpg.de/~juha/publications.html`). The difference covers up to modulo 64 are from [20]. The difference covers modulo 128 and 256 we found using an exhaustive search algorithm. For larger v, we computed the difference covers using the algorithm in [7] (see Lemma 3). In addition to the string and the suffix array, the only non-constant data structures in the implementation are an array storing the difference cover, two lookup tables of size v to implement δ and μ, and an array of size about $4n|D|/v$ bytes for the inverse suffix array ISA$'$. During Phase 1, another array of size $4n|D|/v$ is needed for SA$'$ but we use (a part of) the suffix array for this. The sizes of the difference cover D and the ISA$'$ array (the latter in bytes) for different values of v are:

v	4	8	16	32	64	128	256	512	1024	2048
$\|D\|$	3	4	5	7	9	13	20	28	40	58
$\|\mathrm{ISA}'\|$	$3n$	$2n$	$1.25n$	$0.88n$	$0.56n$	$0.41n$	$0.31n$	$0.22n$	$0.16n$	$0.11n$

We used our own implementations of multikey quicksort [3] for v-ordering, and Larsson-Sadakane [19] for ISA$'$ construction.

There are several possibilities for further improving the running time. For real world data, the majority of the construction time is spent in Step 2.1 for v-ordering the suffixes. The Manzini–Ferragina algorithm [22] also spends it most of its time on v-ordering the suffixes but, as shown by the experiments in the next section, they are much faster. Their implementation of v-ordering uses several additional optimization techniques that could also be used in our algorithm. The most important of these is probably Seward's "pointer copying" heuristic [24].

7 Experiments

We evaluated the performance of our implementation for typical inputs and for bad cases, texts with very high average LCP (longest common prefix). We compared our implementation using a difference cover modulo 32 to Larsson's and Sadakane's implementation of their algorithm (`http://www.cs.lth.se/~jesper/qsufsort.c`) and to that of Manzini and Ferragina (`http://www.mfn.unipmn.it/~manzini/papers/ds.tgz`).

We ran the experiments on a single processor of a Sun-Fire-15000 and compiled with `g++ -O3` respectively `gcc -O3`. The Solaris operating system provides a virtual file, `/proc/self/as`, the size of which equals the total amount of memory a process occupies (including code, stack and data). We monitored this file during our runs and report the largest value.

For the real-world experiments we selected text files from the Canterbury Corpus (`http://corpus.canterbury.ac.nz/index.html`), the tarfile of `gcc 3.0`, the Swissprot protein database (Version 34) and a random (symmetric Bernoulli) string. Table 1 shows alphabet size $|\Sigma|$, average LCP and maximum LCP for these files. The results can be found in Table 2. The algorithm by Manzini and Ferragina outperforms the other two (which are of comparable speeds) by roughly a factor of 3. It also uses the least memory, followed closely by our implementation which requires about 17% more space. The algorithm of Larsson and Sadakane requires between 52 and 59% more space than that of Manzini and Ferragina.

To evaluate performance for difficult cases we created four artificial strings with 50 million characters each. The first contains solely the letter A. This is a frequently used worst-case text, but as our results indicate, it is far from the worst case for some algorithms. Therefore we also created three others consisting of a single random (symmetric Bernoulli) 'seed' string which is repeated until 50 million characters are reached. We used seed strings of length 20, 1000 and 500 000. Statistics for these texts are presented in Table 3 followed by results in Table 4. Apart from the not-so worst case string consisting only of the letter

Table 1. Real-world text files sorted by average LCP

Text	$\|\Sigma\|$	Characters	average LCP	maximum LCP
random string	26	50 000 000	4.84	10
King James Bible (*bible.txt*)	63	4 047 392	13.97	551
E. coli genome (*E.coli*)	4	4 638 690	17.38	2 815
CIA World Fact Book (*world192.txt*)	94	2 473 400	23.01	559
Swissprot V34 Protein Database	66	109 617 186	89.08	7 373
gcc 3.0 source code tarfile	150	86 630 400	8 603.21	856 970

Table 2. Runtimes and space consumption of Larsson-Sadakane(LS), Manzini-Ferragina(MF) and our difference cover algorithm(DC32) for real world files

Text	LS		MF		DC32	
	Time[sec]	Space[MB]	Time[sec]	Space[MB]	Time[sec]	Space[MB]
random	227.46	401.0	106.06	251.9	275.60	295.4
Bible	11.98	33.3	2.12	21.6	5.91	25.5
E. coli	13.56	38.1	2.05	24.6	7.20	28.9
World	4.11	20.8	0.99	13.7	3.81	16.2
Swissprot	996.10	877.9	292.66	551.2	1126.80	645.7
gcc 3.0	528.63	694.0	298.89	439.1	577.60	510.6

Table 3. Text files with long common prefixes

Text	$\|\Sigma\|$	Characters	average LCP	maximum LCP
length 500000 random, repeated	26	50 000 000	24 502 500.5	49 500 000
length 1000 random, repeated	26	50 000 000	24 999 000.5	49 999 000
length 20 random, repeated	15	50 000 000	24 999 980.5	49 999 980
$\{A\}^{5\cdot 10^7}$	1	50 000 000	24 999 999.5	49 999 999

Table 4. Runtimes and space consumption of Larsson-Sadakane(LS), Manzini-Ferragina(MF) and our difference cover algorithm(DC32) for worst-case strings. For the first three databases the algorithm of Manzini and Ferragina did not finish within 24 hours, so we broke off the tests (marked with a -)

Text	LS		MF		DC32	
	Time[sec]	Space[MB]	Time[sec]	Space[MB]	Time[sec]	Space[MB]
500000 char repeats	1310.82	401.0	-	-	618.91	295.4
1000 char repeats	1221.71	401.0	-	-	1460.26	295.4
20 char repeats	1118.49	401.0	-	-	808.36	295.4
$\{A\}^{5\cdot 10^7}$	160.13	401.0	8.30	251.9	343.44	295.4

A, the speed of our implementation (DC32) is comparable to that of Larsson and Sadakane (between 53% faster and 19% slower) but it saves 26% space. For these instances, the algorithm of Manzini and Ferragina had runtimes of more than one day (an estimate based on experiments with smaller datasets resulted in a time of 2 weeks).

We also studied the effect of the choice of the difference cover on the performance of the algorithm. For texts with low or moderate average LCP, large covers result in lower runtimes and space requirements. However, for texts with high average LCP, there exists a tradeoff between space and time consumption. Figure 3 illustrates this with the results of experiments using difference covers modulo 4 to 1024 applied to a random (symmetric Bernoulli) string with 50 million characters and a high-LCP string of the same length (length 1000 random, repeated). It clearly displays the different behaviour of our algorithm for low and high average LCP texts. In addition to the better performance of large covers for texts with low or moderate LCP, it shows that moderately sized covers are relatively good overall performers.

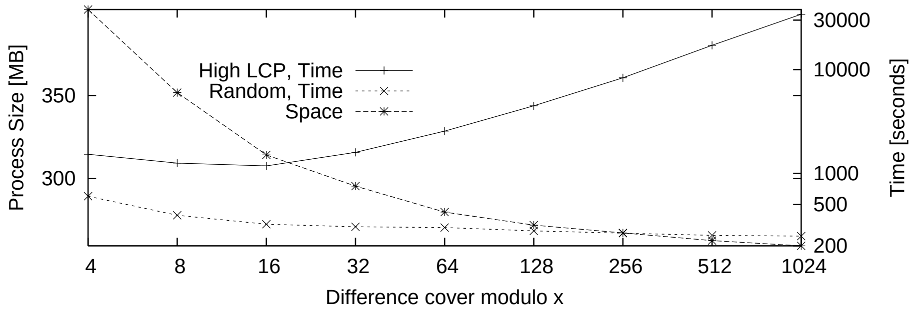

Fig. 3. Influence of the difference cover on performance

8 Suffix Array Checkers

A suffix array checker is an algorithm that takes a string and its suffix array and verifies the correctness of the suffix array. In this section, we describe suffix array checkers based on the following fact.

Theorem 2. *An array* $\mathrm{SA}[0,n)$ *of integers is the suffix array of a string* $s[0,n)$ *if and only if the following conditions are satisfied:*

1. *For all* $i \in [0,n)$, $\mathrm{SA}[i] \in [0,n)$.
2. *For all* $i \in [1,n)$, $s[\mathrm{SA}[i-1]] \leq s[\mathrm{SA}[i]]$.
3. *For all* $i \in [1,n)$ *such that* $s[\mathrm{SA}[i-1]] = s[\mathrm{SA}[i]]$ *and* $\mathrm{SA}[i-1] \neq n-1$, *there exists* $j,k \in [0,n)$ *such that* $\mathrm{SA}[j] = \mathrm{SA}[i-1]+1$, $\mathrm{SA}[k] = \mathrm{SA}[i]+1$ *and* $j < k$.

We omit the proof here.

The conditions 1 and 2 are easy to check in linear time. For checking condition 3, we can find the values j and k by $j = \mathrm{ISA}[\mathrm{SA}[i-1]+1]$ and $k = \mathrm{ISA}[\mathrm{SA}[i]+1]$ if the inverse suffix array ISA is available. Since ISA can be computed from SA (or checked against SA) in linear time, we obtain a simple linear time checker.

The drawback of the above algorithm is the extra space required for the inverse suffix array. Therefore, we offer lightweight alternatives based on the following alternative formulation of Condition 3:

> For all characters $c \in \Sigma$: If $\mathrm{SA}[a, b]$ contains the suffixes starting with the character c, then $\mathrm{SA}[a]+1, \mathrm{SA}[a+1]+1, \ldots, \mathrm{SA}[b]+1$ occur in SA in this order (but not consecutively in general), except that the first entry $\mathrm{SA}[a]+1$ is missing when $c = s[n-1]$.

For a given character, checking the condition in linear time is trivial. This leads to an $\mathcal{O}(\sigma_2 n)$ time, $\mathcal{O}(1)$ extra space algorithm, where σ_2 is the number of distinct characters that occur at least twice in s (no checking is needed for characters that occur only once).

The following algorithm does the checking for all the characters in one pass.

```
1  for i := n − 1, ..., 0 do A(s[SA[i]]) = i
2  A(s[n − 1])++                                   // skip SA[i] = n − 1
3  for k := 0, ..., n − 1 do
4      if SA[k] > 1 then
5          c := s[SA[k] − 1]
6          check that SA[A(c)] + 1 = SA[k]
7          A(c)++
```

The time and space complexity of the algorithm depends on the implementation of A. The algorithm performs $\mathcal{O}(n)$ accesses to A and everything else runs in linear time and constant extra space. Simple implementations include a lookup table, a hash table and a sorted array.

For large alphabets, the space requirement can approach the simpler inverse suffix array checker. A simple optimization, already mentioned above, is to include only characters that occur at least twice. Furthermore, the σ_2 characters can be split into smaller groups and the lines 3–7 of the above algorithm performed separately for each group. Limiting group size to $d < \sigma_2$, the sorted array implementation runs in $\mathcal{O}((\sigma_2 n/d)\log d)$ time and $\mathcal{O}(d)$ extra space. Setting $d = n/\sqrt{\log n}$ to match the space requirement of our construction algorithm, we obtain an $\mathcal{O}\big(n \log \sigma_2 + \sigma_2 \log^{1.5} n\big)$ time checker.

A straightforward sorted array implementation (to make it comparison based as the construction algorithm is) never took more than half the construction time and usually much less.

9 Discussion

We have described a suffix array construction algorithm that combines fast worst case running times with small space consumption. The key idea behind the algorithm is using a sample of suffixes based on a difference cover. There are several further possibilities for extensions and variations of the algorithm. For example, the algorithm can be extended to compute the longest common prefix (LCP) of each pair of adjacent suffixes with the same time and extra space limits. There is a linear time LCP algorithm [14] but it requires $4n$ bytes of extra space.

Another application is sorting a *subset* of suffixes. Some special kinds of subsets can be sorted efficiently [12, 2, 6, 17]. Also, the technique of Section 5 can be used when the set of starting positions is periodic. However, the previous alternatives for sorting an *arbitrary* subset of m suffixes of a string of length n are string sorting, with $\Omega(nm)$ worst case running time, and full suffix array construction, with $\Omega(n)$ extra space requirement. Phase 2 of the difference cover algorithm works just as well for an arbitrary subset of suffixes, giving an algorithm that runs in $\mathcal{O}(\sqrt{v}n + (n/\sqrt{v})\log(n/\sqrt{v}) + vm + m\log m)$ time and $\mathcal{O}(v + n/\sqrt{v})$ extra space. For example, if $m = n/\log n$, by choosing $v = \log^2 n$ we obtain an $\mathcal{O}(n\log n)$ time and $\mathcal{O}(n/\log n) = \mathcal{O}(m)$ extra space algorithm.

The basic idea of using difference covers can also be realized in ways that are quite different from the present one. This is demonstrated by the *linear time* (but not lightweight) algorithm in [11]. To illustrate the similarities and differencies of the two algorithms, we give a brief description (which is quite different from the one in [11]) of the linear time algorithm:

Phase 0. Choose $v = 3$ and $D = \{1, 2\}$. Then the D-sample contains two thirds of the suffixes.

Phase 1. Perform the same steps as in Phase 1 of the present algorithm but do Step 1.1 by radix sort and Step 1.3 by a recursive call.

Phase 2(a). Perform Phase 2 on the remaining one third of the suffixes (not on all suffixes) using radix sort for both steps. Note that using radix sort in Step 2.2 is possible because $\delta(i, j) = 1$ for all $i, j \equiv 0 \pmod 3$.

Phase 2(b). Merge the two sorted groups of suffixes. Comparisons of suffixes are done as in Phase 2 of the present algorithm.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proc. 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of *LNCS*, pages 449–463. Springer, 2002.
2. A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.
3. J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th Annual Symposium on Discrete Algorithms*, pages 360–369. ACM, 1997.
4. M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, Jan. 1995.

5. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC (digital, Palo Alto), May 1994.
6. R. Clifford. Distributed and paged suffix trees for large genetic databases. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, 2003. This volume.
7. C. J. Colbourn and A. C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1–2):9–12, July 2000.
8. M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.
9. G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.
10. H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proc. 6th Symposium on String Processing and Information Retrieval*, pages 125–136. IEEE, 1999.
11. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*. Springer, 2003. To appear.
12. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics*, volume 1090 of *LNCS*, pages 219–230. Springer, 1996.
13. R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. 4th Annual Symposium on Theory of Computing*, pages 125–136. ACM, 1972.
14. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.
15. J. Kilian, S. Kipnis, and C. E. Leiserson. The organization of permutation architectures with bused interconnections. *IEEE Transactions on Computers*, 39(11):1346–1358, Nov. 1990.
16. D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, 2003. This volume.
17. P. Ko and S. Aluru. Linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, 2003. This volume.
18. S. Kurtz. Reducing the space requirement of suffix trees. *Software — Practice and Experience*, 29(13):1149–1171, 1999.
19. N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical report LU-CS-TR:99-214, Dept. of Computer Science, Lund University, Sweden, 1999.
20. W.-S. Luk and T.-T. Wong. Two new quorum based algorithms for distributed mutual exclusion. In *Proc. 17th International Conference on Distributed Computing Systems*, pages 100–106. IEEE, 1997.
21. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, Oct. 1993.
22. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *LNCS*, pages 698–710. Springer, 2002.
23. E. M. McCreight. A space-economic suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

24. J. Seward. On the performance of BWT sorting algorithms. In *Proc. Data Compression Conference*, pages 173–182. IEEE, 2000.
25. J. Seward. The bzip2 and libbzip2 official home page, 2002. `http://sources.redhat.com/bzip2/`.
26. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
27. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, Nov. 1997.
28. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.