Alcom-FT Technical Report Series
ALCOMFT-TR-03-29

Simple Linear Work Suffix Array Construction*

Juha Kärkkäinen, Peter Sanders

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

Abstract. A suffix array represents the suffixes of a string in sorted order. Being a simpler and more compact alternative to suffix trees, it is an important tool for full text indexing and other string processing tasks. We introduce the *skew algorithm* for suffix array construction over integer alphabets that can be implemented to run in linear time using integer sorting as its only nontrivial subroutine:

1. recursively sort suffixes beginning at positions $i \bmod 3 \neq 0$.
2. sort the remaining suffixes using the information obtained in step one.
3. merge the two sorted sequences obtained in steps one and two.

The algorithm is much simpler than previous linear time algorithms that are all based on the more complicated suffix tree data structure. Since sorting is a well studied problem, we obtain optimal algorithms for several other models of computation, e.g. external memory with parallel disks, cache oblivious, and parallel. The adaptations for BSP and EREW-PRAM are asymptotically faster than the best previously known algorithms.

1 Introduction

The suffix *tree* [39] of a string is a compact trie of all the suffixes of the string. It is a powerful data structure with numerous applications in computational biology [21] and elsewhere [20]. One of the important properties of the suffix tree is that it can be constructed in linear time in the length of the string. The classical linear time algorithms [32, 36, 39] require a constant alphabet size, but Farach's algorithm [11, 14] works also for integer alphabets, i.e., when characters are polynomially bounded integers. There are also efficient construction algorithms for many advanced models of computation (see Table 1).

The suffix *array* [18, 31] is a lexicographically sorted array of the suffixes of a string. For several applications, the suffix array is a simpler and more compact alternative to the suffix tree [2, 6, 18, 31]. The suffix array can be constructed in linear time by a lexicographic traversal of the suffix tree, but such a construction loses some of the advantage that the suffix array has over the suffix tree. The fastest *direct* suffix array construction algorithms that do not use suffix trees require $\mathcal{O}(n \log n)$ time [5, 30, 31]. Also under other models of computation, direct algorithms cannot match suffix tree based algorithms [9, 16]. The existence of an I/O-optimal direct algorithm is mentioned as an important open problem in [9].

We introduce the *skew algorithm*, the first linear-time direct suffix array construction algorithm for integer alphabets. The skew algorithm is simpler than any suffix tree construction algorithm. (In the appendix, we give a 50 line C++ implementation.) In particular, it is much simpler than linear time suffix tree construction for integer alphabets.

Independently of and in parallel with the present work, two other direct linear time suffix array construction algorithms have been introduced by Kim et al. [28], and Ko and Aluru [29]. The two algorithms are quite different from ours (and each other).

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

The skew algorithm. Farach's linear-time suffix tree construction algorithm [11] as well as some parallel and external algorithms [12–14] are based on the following divide-and-conquer approach:

1. Construct the suffix tree of the suffixes starting at odd positions. This is done by reduction to the suffix tree construction of a string of half the length, which is solved recursively.
2. Construct the suffix tree of the remaining suffixes using the result of the first step.
3. Merge the two suffix trees into one.

The crux of the algorithm is the last step, merging, which is a complicated procedure and relies on structural properties of suffix trees that are not available in suffix arrays. In their recent direct linear time suffix array construction algorithm, Kim et al. [28] managed to perform the merging using suffix arrays, but the procedure is still very complicated.

The skew algorithm has a similar structure:

1. Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.
2. Construct the suffix array of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Surprisingly, the use of two thirds instead of half of the suffixes in the first step makes the last step almost trivial: a simple comparison-based merging is sufficient. For example, to compare suffixes starting at i and j with $i \bmod 3 = 0$ and $j \bmod 3 = 1$, we first compare the initial characters, and if they are the same, we compare the suffixes starting at $i+1$ and $j+1$ whose relative order is already known from the first step.

Results. The simplicity of the skew algorithm makes it easy to adapt to other models of computation. Table 1 summarizes our results together with the best previously known algorithms for a number of important models of computation. The column "alphabet" in Table 1 identifies the model for the alphabet Σ. In a *constant* alphabet, we have $|\Sigma| = \mathcal{O}(1)$, an *integer* alphabet means that characters are integers in a range of size $n^{\mathcal{O}(1)}$, and *general* alphabet only assumes that characters can be compared in constant time.

The skew algorithm for RAM, external memory and cache oblivious models is the first optimal *direct* algorithm. For BSP and EREW-PRAM models, we obtain an improvement over *all* previous results, including the first linear work BSP algorithm. On all the models, the skew algorithm is much simpler than the best previous algorithm.

In many applications, the suffix array needs to be augmented with additional data, the most important being the longest common prefix (lcp) array [1, 2, 26, 27, 31]. In particular, the suffix tree can be constructed easily from the suffix and lcp arrays [11, 13, 14]. There is a linear time algorithm for computing the lcp array from the suffix array [27], but it does not appear to be suitable for parallel or external computation. We extend our algorithm to compute also the lcp array while retaining the complexities of Table 1. Hence, we also obtain improved suffix tree construction algorithms for the BSP and EREW-PRAM models.

The paper is organized as follows. In Section 2, we describe the basic skew algorithm, which is then adapted to different models of computation in Section 3. The algorithm is extended to compute the longest common prefixes in Section 4.

2 The Skew Algorithm

For compatibility with C and because we use many modulo operations we start arrays at position 0. We use the abbreviations $[a, b] = \{a, \dots, b\}$ and $s[a, b] = [s[a], \dots, s[b]]$ for a string or array s.

Table 1. Suffix array construction algorithms. The algorithms in [11–14] are *indirect*, i.e., they actually construct a suffix *tree*, which can be then be transformed into a suffix array

model of computation	complexity	alphabet	source
RAM	$\mathcal{O}(n \log n)$ time	general	[31, 30, 5]
	$\mathcal{O}(n)$ time	integer	[11, 28, 29],skew
External Memory [38] D disks, block size B, fast memory of size M	$\mathcal{O}\left(\frac{n}{DB} \log_{\frac{M}{B}} \frac{n}{B} \log_2 n\right)$ I/Os $\mathcal{O}\left(n \log_{\frac{M}{B}} \frac{n}{B} \log_2 n\right)$ internal work	integer	[9]
	$\mathcal{O}\left(\frac{n}{DB} \log_{\frac{M}{B}} \frac{n}{B}\right)$ I/Os $\mathcal{O}\left(n \log_{\frac{M}{B}} \frac{n}{B}\right)$ internal work	integer	[14],skew
Cache Oblivious [15] M/B cache blocks of size B	$\mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B} \log_2 n\right)$ cache faults	general	[9]
	$\mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right)$ cache faults	general	[14],skew
BSP [37]	$\mathcal{O}\left(\frac{n \log n}{P} + (L + \frac{gn}{P})\frac{\log^3 n \log P}{\log(n/P)}\right)$ time	general	[12]
P processors h-relation in time $L + gh$	$\mathcal{O}\left(\frac{n \log n}{P} + L \log^2 P + \frac{gn \log n}{P \log(n/P)}\right)$ time	general	skew
$P = \mathcal{O}(n^{1-\epsilon})$ processors	$\mathcal{O}(n/P + L \log^2 P + gn/P)$ time	integer	skew
EREW-PRAM [25]	$\mathcal{O}(\log^4 n)$ time, $\mathcal{O}(n \log n)$ work	general	[12]
	$\mathcal{O}(\log^2 n)$ time, $\mathcal{O}(n \log n)$ work	general	skew
arbitrary-CRCW-PRAM [25]	$\mathcal{O}(\log n)$ time, $\mathcal{O}(n)$ work (rand.)	constant	[13]
priority-CRCW-PRAM [25]	$\mathcal{O}(\log^2 n)$ time, $\mathcal{O}(n)$ work (rand.)	constant	skew

Similarly, $[a, b) = [a, b-1]$ and $s[a, b) = s[a, b-1]$. The operator $\circ$ is used for the concatenation of strings. Consider a string $s = s[0, n)$ over the alphabet $\Sigma = [1, n]$. The suffix array SA contains the suffixes $S_i = s[i, n)$ in sorted order, i.e., if $\mathrm{SA}[i] = j$ then suffix S_j has rank $i + 1$ among the set of strings $\{S_0, \ldots, S_{n-1}\}$. To avoid tedious special case treatments, we describe the algorithm for the case that n is a multiple of 3 and adopt the convention that all strings α considered have $\alpha[|\alpha|] = \alpha[|\alpha|+1] = 0$. The implementation in the Appendix fills in the remaining details. Figure 1 gives an example.

The first and most time consuming step of the skew algorithm sorts the suffixes S_i with $i \bmod 3 \neq 0$ among themselves. To this end, it first finds *lexicographic names* $s'_i \in [1, 2n/3]$ for the triples $s[i, i+2]$ with $i \bmod 3 \neq 0$, i.e., numbers with the property that $s'_i \leq s'_j$ if and only if $s[i, i+2] \leq s[j, j+2]$. This can be done in linear time by radix sort and scanning the sorted sequence of triples — if triple $s[i, i+2]$ is the k-th different triple appearing in the sorted sequence, we set $s'_i = k$.

If all triples get different lexicographic names, we are done with step one. Otherwise, the suffix array SA^{12} of the string

$$s^{12} = [s'_i : i \bmod 3 = 1] \circ [s'_i : i \bmod 3 = 2]$$

is computed recursively. Note that there can be no more lexicographic names than characters in s^{12} so that the alphabet size in a recursive call never exceeds the size of the string. The recursively computed suffix array SA^{12} represents the desired order of the suffixes S_i with $i \bmod 3 \neq 0$. To see this, note that $s^{12}[\frac{i-1}{3}, \frac{n}{3})$ for $i \bmod 3 = 1$ represents the suffix $S_i = s[i, n) \circ [0]$ via lexicographic naming. The 0 characters at the end of s make sure that $s^{12}[n/3-1]$ is unique in s^{12} so that it does

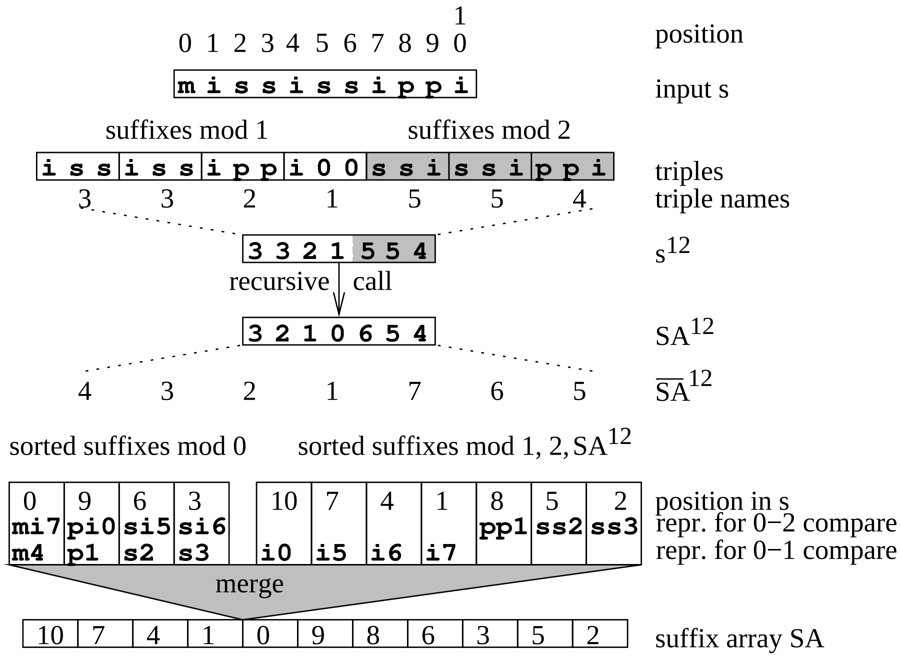

Fig. 1. The skew algorithm applied to $s = \texttt{mississippi}$.

not matter that s^{12} has additional characters. Similarly, $s^{12}[\frac{n+i-2}{3}, \frac{2n}{3})$ for $i \bmod 3 = 2$ represents the suffix $S_i = s[i, n) \circ [0, 0]$.

The second step is easy. The suffixes S_i with $i \bmod 3 = 0$ are sorted by sorting the pairs $(s[i], S_{i+1})$. Since the order of the suffixes S_{i+1} is already implicit in SA^{12}, it suffices to stably sort those entries $\mathrm{SA}^{12}[j]$ that represent suffixes S_{i+1}, $i \bmod 3 = 0$, with respect to $s[i]$. This is possible in linear time by a single pass of radix sort.

The skew algorithm is so simple because also the third step is quite easy. We have to merge the two suffix arrays to obtain the complete suffix array SA. To compare a suffix S_j with $j \bmod 3 = 0$ with a suffix S_i with $i \bmod 3 \neq 0$, we distinguish two cases:
If $i \bmod 3 = 1$, we write S_i as $(s[i], S_{i+1})$ and S_j as $(s[j], S_{j+1})$. Since $i + 1 \bmod 3 = 2$ and $j + 1 \bmod 3 = 1$, the relative order of S_{j+1} and S_{i+1} can be determinded from their position in SA^{12}. This position can be determined in constant time by precomputing an array $\overline{\mathrm{SA}}^{12}$ with $\overline{\mathrm{SA}}^{12}[i] = j + 1$ if $\mathrm{SA}^{12}[j] = i$. This is nothing but a special case of lexicographic naming.[1]

Similarly, if $i \bmod 3 = 2$, we compare the triples $(s[i], s[i+1], S_{i+2})$ and $(s[j], s[j+1], S_{j+2})$ replacing S_{i+2} and S_{j+2} by their lexicographic names in $\overline{\mathrm{SA}}^{12}$.

The running time of the skew algorithm is easy to establish.

Theorem 1. *The skew algorithm can be implemented to run in time $\mathcal{O}(n)$.*

Proof. The execution time obeys the recurrence $T(n) = \mathcal{O}(n) + T(\lceil 2n/3 \rceil)$, $T(n) = \mathcal{O}(1)$ for $n < 3$. This recurrence has the solution $T(n) = \mathcal{O}(n)$. □

[1] $\overline{\mathrm{SA}}^{12} - 1$ is also known as the *inverse suffix array* of SA^{12}.

3 Other Models of Computation

Theorem 2. *The skew algorithm can be implemented to achieve the following performance guarantees on advanced models of computation:*

model of computation	complexity	alphabet
External Memory [38] D disks, block size B, fast memory of size M	$\mathcal{O}\left(\frac{n}{DB}\log_{\frac{M}{B}}\frac{n}{B}\right)$ I/Os $\mathcal{O}\left(n\log_{\frac{M}{B}}\frac{n}{B}\right)$ internal work	integer
Cache Oblivious [15]	$\mathcal{O}\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$ cache faults	general
BSP [37] P processors h-relation in time $L+gh$	$\mathcal{O}\left(\frac{n\log n}{P}+L\log^2 P+\frac{gn\log n}{P\log(n/P)}\right)$ time	general
$P=\mathcal{O}(n^{1-\epsilon})$ processors	$\mathcal{O}(n/P+L\log^2 P+gn/P)$ time	integer
EREW-PRAM [25]	$\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n\log n)$ work	general
priority-CRCW-PRAM [25]	$\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n)$ work (rand.)	constant

Proof. **External Memory:** Sorting tuples and lexicographic naming is easily reduced to external memory integer sorting. I/O optimal deterministic[2] parallel disk sorting algorithms are well known [34, 33]. We have to make a few remarks regarding internal work however. To achieve optimal internal work for all values of n, M, and B, we can use radix sort where the most significant digit has $\lfloor\log M\rfloor-1$ bits and the remaining digits have $\lfloor\log M/B\rfloor$ bits. Sorting then starts with $\mathcal{O}\left(\log_{M/B} n/M\right)$ data distribution phases that need linear work each and can be implemented using $\mathcal{O}(n/DB)$ I/Os using the same I/O strategy as in [33]. It remains to stably sort the elements by their $\lfloor\log M\rfloor-1$ most significant bits. For this we can use the distribution based algorithm from [33] directly. In the distribution phases, elements can be put into a bucket using a full lookup table mapping keys to buckets. Sorting buckets of size M can be done in linear time using a linear time internal algorithm.

Cache Oblivious: We use the comparison based model here since it is not known how to do cache oblivious integer sorting with $\mathcal{O}(\frac{n}{B}\log_{M/B}\frac{n}{B})$ cache faults and $o(n\log n)$ work. The result is an immediate corollary of the optimal comparison based sorting algorithm [15].

EREW PRAM: We can use Cole's merge sort [8] for sorting and merging. Lexicographic naming can be implemented using linear work and $\mathcal{O}(\log P)$ time using prefix sums. After $\Theta(\log P)$ levels of recursion, the problem size has reduced so far that the remaining subproblem can be solved on a single processor. We get an overall execution time of $\mathcal{O}(n\log n/P+\log^2 P)$.

BSP: For the case of many processors, we proceed as for the EREW-PRAM algorithm using the optimal comparison based sorting algorithm [19] that takes time $\mathcal{O}(n\log n/P+(gn/P+L)\frac{\log n}{\log(n/P)})$.

For the case of few processors, we can use a linear work sorting algorithm based on radix sort [7] and a linear work merging algorithm [17]. The integer sorting algorithm remains applicable at least during the first $\Theta(\log\log n)$ levels of recursion of the skew algorithm. Then we can afford to switch to a comparison based algorithm without increasing the overall amount of internal work.

CRCW PRAM: We employ the stable integer sorting algorithm [35] that works in $\mathcal{O}(\log n)$ time using linear work for keys with $\mathcal{O}(\log\log n)$ bits. This algorithm can be used for the first

[2] Simpler randomized algorithms with favorable constant factors are also available [10].

$\Theta(\log\log\log n)$ iterations. Then we can afford to switch to the algorithm [22] that works for polynomial size keys at the price of being inefficient by a factor $\mathcal{O}(\log\log n)$. Lexicographic naming can be implemented by computing prefix sums using linear work and logarithmic time. Comparison based merging can be implemented with linear work and $\mathcal{O}(\log n)$ time using [23]. □

The resulting algorithms are simple except that they may use complicated subroutines for sorting to obtain theoretically optimal results. There are usually much simpler implementations of sorting that work well in practice although they may sacrifice determinism or optimality for certain combinations of parameters.

4 Longest Common Prefixes

Let $\mathrm{lcp}(i,j)$ denote the length of the longest common prefix (lcp) of the suffixes S_i and S_j. The longest common prefix array LCP contains the lengths of the longest common prefixes of suffixes that are adjacent in the suffix array, i.e., $\mathrm{LCP}[i] = \mathrm{lcp}(SA[i], SA[i+1])$. A well-known property of lcps is that for any $0 \le i < j < n$,

$$\mathrm{lcp}(i,j) = \min_{i \le k < j} \mathrm{LCP}[k] \;.$$

Thus, if we preprocess LCP in linear time to answer range minimum queries in constant time [3, 4, 24], we can find the longest common prefix of any two suffixes in constant time.

We will show how the LCP array can be computed from the LCP^{12} array corresponding to SA^{12} in linear time. Let $j = \mathrm{SA}[i]$ and $k = \mathrm{SA}[i+1]$. We explain two cases; the others are similar.

First, assume that $j \bmod 3 = 1$ and $k \bmod 3 = 2$, and let $j' = (j-1)/3$ and $k' = (n+k-2)/3$ be the corresponding positions in s^{12}. Since j and k are adjacent in SA, so are j' and k' in SA^{12}, and thus $\ell = \mathrm{lcp}^{12}(j',k') = \mathrm{LCP}^{12}[\overline{\mathrm{SA}}^{12}[j']-1]$. Then $\mathrm{LCP}[i] = \mathrm{lcp}(j,k) = 3\ell + \mathrm{lcp}(j+3\ell, k+3\ell)$, where the last term is at most 2 and can be computed in constant time by character comparisons.

As the second case, assume $j \bmod 3 = 0$ and $k \bmod 3 = 1$. If $s[j] \ne s[k]$, $\mathrm{LCP}[i] = 0$ and we are done. Otherwise, $\mathrm{LCP}[i] = 1 + \mathrm{lcp}(j+1, k+1)$, and we can compute $\mathrm{lcp}(j+1,k+1)$ as above as $3\ell + \mathrm{lcp}(j+1+3\ell, k+1+3\ell)$, where $\ell = \mathrm{lcp}^{12}(j',k')$ with $j' = ((j+1)-1)/3$, $k' = (n+(k+1)-2)/3$. An additional complication is that, unlike in the first case, $j+1$ and $k+1$ may not be adjacent in SA, and consequently, j' and k' may not be adjacent in SA^{12}. Thus we have to compute ℓ by performing a range minimum query in LCP^{12} instead of a direct lookup. However, this is still constant time.

Theorem 3. *The extended skew algorithm computing both* SA *and* LCP *can be implemented to run in linear time.*

To obtain the same extension for other models of computation, we need to show how to answer $\mathcal{O}(n)$ range minimum queries on LCP^{12}. We can take advantage of the balanced distribution of the range minimum queries shown by the following property.

Lemma 1. *No suffix is involved in more than two lcp queries at the top level of the extended skew algorithm.*

Proof. Let S_i and S_j be two suffixes whose lcp $\mathrm{lcp}(i,j)$ is computed to find the lcp of the suffixes S_{i-1} and S_{j-1}. (The other case that $\mathrm{lcp}(i,j)$ is needed for the lcp of S_{i-2} and S_{j-2} is similar.) Then S_{i-1} and S_{j-1} are lexicographically adjacent suffixes and $s[i-1] = s[j-1]$. Thus, there cannot be another suffix S_k, $S_i < S_k < S_j$, with $s[k-1] = s[i-1]$. This shows that a suffix can be involved in lcp queries only with its two lexicographically nearest neighbors that have the same preceding character. □

We describe a simple algorithm for answering the range minimum queries that can be easily adapted to the models of Theorem 2. It is based on the ideas in [3,4] (which are themselves based on earlier results).

The LCP^{12} array is divided into blocks of size $\log n$. For each block $[a, b]$, precompute and store the following data:

- For all $i \in [a, b]$, a $\log n$-bit vector Q_i that identifies all $j \in [a, i]$ such that $\mathrm{LCP}^{12}[j] < \min_{k \in [j+1, i]} \mathrm{LCP}^{12}[k]$.
- For all $i \in [a, b]$, the minimum values over the ranges $[a, i]$ and $[i, b]$.
- The minimum for all ranges that end just before or begin just after $[a, b]$ and contain exactly a power of two full blocks.

If a range $[i, j]$ is completely inside a block, its minimum can be found with the help of Q_j in constant time (see [3] for details). Otherwise, $[i, j]$ can be covered with at most four of the ranges whose minimum is stored, and its minimum is the smallest of those minima.

Theorem 4. *The extended skew algorithm computing both* SA *and* LCP *can be implemented to achieve the complexities of Theorem 2.*

Proof. (Outline) **External Memory and Cache Oblivious:** The range minimum algorithm can be implemented with sorting and scanning.

Parallel models: The blocks in the range minima data structure are distributed over the processors in the obvious way. Preprocessing range minima data structures reduces to local operations and a straightforward computation proceeding from shorter to longer ranges. Lemma 1 ensures that queries are evenly balanced over the data structure. □

5 Discussion

The skew algorithm is a simple and asymptotically efficient direct algorithm for suffix array construction that is easy to adapt to various models of computation. We expect that it is a good starting point for actual implementations, in particular on parallel machines and for external memory.

The key to the algorithm is the use of suffixes S_i with $i \bmod 3 \in \{1, 2\}$ in the first, recursive step, which enables simple merging in the third step. There are other choices of suffixes that would work. An interesting possibility, for example, is to take suffixes S_i with $i \bmod 7 \in \{3, 5, 6\}$. Some adjustments to the algorithm are required (sorting the remaining suffixes in multiple groups and performing a multiway merge in the third step) but the main ideas still work. In general, a suitable choice is a periodic set of positions according to a *difference cover*. 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 $\{1, 2\}$ is a difference cover modulo 3 and $\{3, 5, 6\}$ is a difference cover modulo 7, but $\{1\}$ is not a difference cover modulo 2. Any nontrivial difference cover modulo a constant could be used to obtain a linear time algorithm. Difference covers and their properties play a more central role in the suffix array construction algorithm in [5], which runs in $\mathcal{O}(n \log n)$ time using sublinear extra space in addition to the string and the suffix array.

An interesting theoretical question is whether there are faster CRCW-PRAM algorithms for direct suffix array construction. For example, there are very fast algorithms for padded sorting, list sorting and approximate prefix sums [22] that could be used for sorting and lexicographic naming in the recursive calls. The result would be some kind of suffix list or padded suffix array that could be converted into a suffix array in logarithmic time.

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. M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. 9th Symposium on String Processing and Information Retrieval*, volume 2476 of *LNCS*, pages 31–43. Springer, 2002.
3. S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proc. 14th Annual Symposium on Parallel Algorithms and Architectures*, pages 258–264. ACM, 2002.
4. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Symposium on Theoretical INformatics*, volume 1776 of *LNCS*, pages 88–94. Springer, 2000.
5. S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, June 2003. To appear.
6. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC (digital, Palo Alto), May 1994.
7. A. Chan and F. Dehne. A note on coarse grained parallel integer sorting. *Parallel Processing Letters*, 9(4):533–538, 1999.
8. R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.
9. A. Crauser and P. Ferragina. Theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.
10. R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. 15th Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2003. To appear.
11. M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.
12. M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pages 174–183. IEEE, 1998.
13. M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proc. 23th International Conference on Automata, Languages and Programming*, pages 550–561. IEEE, 1996.
14. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
15. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–298. IEEE, 1999.
16. N. Futamura, S. Aluru, and S. Kurtz. Parallel suffix sorting. In *Proc. 9th International Conference on Advanced Computing and Communications*, pages 76–81. Tata McGraw-Hill, 2001.
17. A. V. Gerbessiotis and C. J. Siniolakis. Merging on the BSP model. *Parallel Computing*, 27:809–822, 2001.
18. 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.
19. M. T. Goodrich. Communication-efficient parallel sorting. *SIAM J. Comput.*, 29(2):416–432, 1999.
20. R. Grossi and G. F. Italiano. Suffix trees and their applications in string algorithms. Rapporto di Ricerca CS-96-14, Università "Ca' Foscari" di Venezia, Italy, 1996.
21. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
22. T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proc. 33rd Annual Symposium on Foundations of Computer Science*, pages 628–637. IEEE, 1992.
23. T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW-PRAM. *Information Processing Letters*, 33:181–185, 1989.
24. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

25. J. Jájá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.
26. J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In Z. Galil and E. Ukkonen, editors, *Proc. 6th Annual Symposium on Combinatorial Pattern Matching*, volume 937 of *LNCS*, pages 191–204. Springer, 1995.
27. 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.
28. 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, June 2003. To appear.
29. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, June 2003. To appear.
30. N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical report LU-CS-TR:99-214, Dept. of Computer Science, Lund University, Sweden, 1999.
31. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, Oct. 1993.
32. E. M. McCreight. A space-economic suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
33. M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th Annual Symposium on Parallel Algorithms and Architectures*, pages 120–129. ACM, 1993.
34. M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *J. ACM*, 42(4):919–933, 1995.
35. S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.
36. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
37. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 22(8):103–111, Aug. 1990.
38. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.
39. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

A Source Code

The following C++ file contains a complete linear time implementation of suffix array construction. This code strives for conciseness rather than for speed — it has only 50 lines not counting comments, empty lines, and lines with a bracket only. A driver program can be found at `http://www.mpi-sb.mpg.de/~sanders/programs/suffix/`.

```
inline bool leq(int a1, int a2,   int b1, int b2) // lexicographic order
{ return(a1 < b1 || a1 == b1 && a2 <= b2); }                 // for pairs
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3)
{ return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3)); }     // and triples

// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];                              // counter array
  for (int i = 0;  i <= K;  i++) c[i] = 0;         // reset counters
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;   // count occurrences
  for (int i = 0, sum = 0;  i <= K;  i++)   // exclusive prefix sums
  {  int t = c[i];  c[i] = sum;  sum += t; }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];      // sort
  delete [] c;
}

// find the suffix array SA of s[0..n-1] in {1..K}^n
// require s[n]=s[n+1]=s[n+2]=0, n>=2
void suffixArray(int* s, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* s0   = new int[n0];
  int* SA0  = new int[n0];

  // generate positions of mod 1 and mod  2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);
```

```
  // find lexicographic names of triples
  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0;  i < n02;  i++) {
    if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2)
    {  name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2]; }
    if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; } // left half
    else                  { s12[SA12[i]/3 + n0] = name; } // right half
  }

  // recurse if names are not yet unique
  if (name < n02) {
    suffixArray(s12, SA12, n02, name);
    // store unique names in s12 using the suffix array
    for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
  } else // generate the suffix array of s12 directly
    for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;

  // stably sort the mod 0 suffixes from SA12 by their first character
  for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
  radixPass(s0, SA0, s, n0, K);

  // merge sorted SA0 suffixes and sorted SA12 suffixes
  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    if (SA12[t] < n0 ? // different compares for mod 1 and mod 2 suffixes
        leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    {        // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
    } else { // suffix from SA0  is smaller
      SA[k] = j;  p++;
      if (p == n0)  // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}
```