Simple In-Place Suffix Array Construction by Walking along Burrows-Wheeler Transforms

Sven Rahmann and Jasmin Smula

Bioinformatics for High-Throughput Technologies group
Computer Science 11 (Algorithm Engineering)
TU Dortmund, D-44221 Dortmund, Germany
Sven.Rahmann@tu-dortmund.de

Abstract. We present a family of simple algorithms that construct a representation of the suffix array of a text t by walking along the Burrows-Wheeler transforms (BWTs) of increasingly longer suffixes of t. One variant, called Walk-bothLR, builds the suffix array, its inverse, the BWT, and the longest common prefix array in linear time in practice (quadratic in the worst case), requiring only constant working space, and only consecutive left-to-right read/write access to secondary storage. Another variant, called Walk-minLR, is twice as fast as Walk-bothLR on random texts, and has worst-case time complexity $O(n \log n)$ for a text of length n. It needs one additional integer array as working space. While asymptotically faster algorithms exist, and compressed suffix arrays require even less memory, the algorithms presented here provide a good compromise between running time and memory requirements. They are especially well suited to build suffix arrays of large genomes, especially for long repeats. The $O(n \log n)$ time bound for Walk-minLR sheds a new light on the structure of BWTs. The algorithms are simple and easy to implement; software is freely available at http://verjinxer.googlecode.com.

1 Introduction

A variety of ideas and techniques, recently surveyed in [1], has been developed for suffix array construction. Starting with Manber and Myers' $O(n \log n)$ direct construction algorithm [2] for a length-n text in 1990, a theoretical breakthrough came in 2003, when three linear-time algorithms were published whose journal versions appeared as [3–5]. Carefully engineered algorithms without a linear time guarantee (e.g. [6–8]) often perform better than theoretically optimal algorithms.

In biological sequence analysis, the alphabet Σ is small (DNA, RNA: $|\Sigma| = 4$, protein sequences: $|\Sigma| = 20$), and we frequently need to index large numbers of long sequences, so especially low working space requirements are desirable. Manzini et al. [8] call algorithms that need only Kn additional bytes with a small constant $K \ll 1$, *lightweight* algorithms. Going a step further, we call algorithms that need only $O(1)$ bytes of working space *in-place* algorithms. Recent work in this area includes sophisticated optimal algorithms for general and constant-size alphabets [9, 10]; their implementation status is unknown.

An idea that has experienced much attention and development recently is the use of *compressed suffix arrays* that can store a compressed version of the text and a representation of its suffix array in less space than the text itself (e.g. [11–13]. Asymptotically optimal algorithms exist [11], but practical access to the suffix array suffers from high constant factors hidden in the O-notation.

Ideally, we desire algorithms that simultaneously run in linear time on typical sequence data, maybe even in the worst case, require only a small amount of (or no) working space, and rely on simple ideas, using no advanced data structures to allow for a simple implementation and keep constant factors low. This paper presents two simple algorithms for suffix array construction that come close to the above ideals, but have different space-time trade-offs, and may shed a new light on the interplay between the suffix array and the Burrows-Wheeler transform. To our knowledge, the ideas behind compressed suffix arrays have not yet been used to develop a simple direct (uncompressed) construction algorithm.

In the algorithms, a linked-list simulation of a suffix array of a string $t = cu$, $c \in \Sigma$, $u \in \Sigma^*$, is constructed from that of its suffix u. The main step executed by the algorithms is a walk along the Burrows-Wheeler Transform (BWT) of u to insert suffix cu at the correct position in lexicographic order.

The algorithms assume that $|\Sigma| = O(1)$, and the present implementation assumes $|\Sigma| < 256$ and $n < 2^{31}$, so the text needs one byte per character, and integer arrays need 4 bytes per character.

One algorithm (Walk-minLR) needs linear time on typical sequence data (but also on Fibonacci strings and other strings with long repeats), but its worst-case time complexity is $O(n \log n)$. It needs $4n$ bytes of additional working space, which however can be efficiently used to construct the longest common prefix array (without further working space; see Section 4).

The other algorithm (Walk-bothLR) only needs $O(1)$ working space and is thus a simple in-place suffix sorting algorithm, with linear running time in practice, but quadratic running time in the worst case. Even better, by only using $5n + O(1)$ bytes of memory (for the text and one integer array at a time), we can construct all of the following arrays with only consecutive left-to-right read/write accesses to secondary storage: the suffix array pos, its inverse rank, the Burrows-Wheeler Transform, and the longest common prefix array lcp. These arrays are formally defined in the next section.

After giving basic definitions in the next section, we describe and analyze the algorithms in Section 3, followed by descriptions of typical downstream computations in Section 4. Empirical studies are then presented in Section 5, and we conclude the article in Section 6.

2 Basic Definitions

The basic *suffix array* pos of a text $t = t_0 \dots t_{n-1}$ of length n over a finite ordered alphabet Σ is a permutation of $\{0, \dots, n-1\}$, where $\mathsf{pos}[r]$ is the starting position of the lexicographically r-th smallest suffix ($r = 0, \dots, n-1$) in t. By convention

the end of the string is lexicographically smaller than any character and often represented by a special end-of-string marker $.

The inverse permutation rank, defined by $\mathsf{rank}[\mathsf{pos}[r]] = r$ for all $r \in \{0, \ldots, n-1\}$, thus gives the lexicographic rank of the text suffix $t^p := t_p \ldots t_{n-1}$.

(While other notations for the suffix array and its inverse exist, we advocate $\mathsf{pos}[r]$ and $\mathsf{rank}[p]$, as it appears to be the most mnemonic notation available.)

The (suffix-based) Burrows-Wheeler Transform (BWT) $\widehat{t}$ of t is defined by $\widehat{t}_r := t_{\mathsf{pos}[r]-1}$ for $r \neq \mathsf{rank}[0]$, and $\widehat{t}_{\mathsf{rank}[0]} := t_{n-1} = \$$. As is evident from the definition, the BWT can be easily constructed from the suffix array and the text.

The *longest common prefix array* lcp array is defined by $\mathsf{lcp}[0] := 0$, and for $0 < r < n$, $\mathsf{lcp}[r]$ is the length of the longest common prefix of the lexicographically adjacent suffixes $t^{\mathsf{pos}[r-1]}$ and $t^{\mathsf{pos}[r]}$. Kasai et al. [14] describe a simple linear-time algorithm to compute lcp from pos and rank. Together, pos and lcp suffice to simulate bottom-up traversals of suffix trees [15].

For two strings $u = u_0 u_1 \cdots \in \Sigma^*$ and $v = v_0 v_1 \cdots \in \Sigma^*$, we write $u \prec v$ if u is lexicographically smaller than v, that is,

$$u \prec v :\Longleftrightarrow \begin{cases} |u| = 0, |v| > 0 \\ \text{or } |u| > 0, |v| > 0 \text{ and } \left[(u_0 \prec v_0) \text{ or } (u_0 = v_0 \text{ and } u^1 \prec v^1)\right]. \end{cases}$$

We write $p \prec_t^* q$ if suffix t^p is the immediate lexicographic predecessor of t^q among all suffixes of t; that is if $t^p \prec t^q$ and there exists no $i \in \{0, \ldots, n-1\}$ with $t^p \prec t^i \prec t^q$.

The algorithms in this article do not compute pos directly, but a different representation of the same information. For $0 \leq p < n$, we define $\mathsf{lexnextpos}[p] := \mathsf{pos}[\mathsf{rank}[p]+1]$, the starting position of the suffix that comes next in lexicographic order after t^p. If such a suffix does not exist (because $\mathsf{rank}[p] = n-1$), we let $\mathsf{lexnextpos}[p] := \perp$ (nil). Similarly we define $\mathsf{lexprevpos}[p] := \mathsf{pos}[\mathsf{rank}[p]-1]$.

Note that similar ideas are standard in the literature on succinct data structures for suffix arrays; yet a subtle and important difference is that they use the function $\mathsf{rightrank}[r] := \mathsf{rank}[\mathsf{pos}[r]+1]$ (called $\Psi[r]$ in [11]), which can be more efficiently stored, but them requires constant overhead for accessing.

For each $c \in \Sigma$, we further define $\mathsf{lexfirstpos}[c]$ and $\mathsf{lexlastpos}[c]$ as the starting positions of the lexicographically first and last suffixes that begin with c.

3 Suffix Array Construction Algorithms

We construct the suffix array of $t = t_0 \ldots t_{n-1}$ in n rounds, starting with round $n-1$, counting backwards, and finishing with round 0. After round p, we have a representation of the suffix array of $t^p = t_p \ldots t_{n-1}$. Thus on a high level, the algorithm behaves like Weiner's original suffix tree construction algorithm [16].

Instead of pos or rank, the algorithms construct lexnextpos and lexprevpos, from which pos, rank, $\widehat{t}$, and lcp arrays can be easily derived (see Section 4). With lexnextpos and lexprevpos, we simulate a doubly linked list (without constructing it explicitly) that keeps the suffix start positions in lexicographic order.

The auxiliary arrays lexfirstpos and lexlastpos facilitate inserting suffixes $t^p = cu$, where $c \in \Sigma$ does not occur in $u \in \Sigma^*$. Initially (before round $n-1$), we have $\mathsf{lexfirstpos}[c] = \mathsf{lexlastpos}[c] = \perp$ for all $c \in \Sigma$.

The central question is thus, how do we obtain the linked-list representation of the suffix array of cu from the suffix array of u ($c \in \Sigma$, $u \in \Sigma^*$)? In round p, we must solve the problem of inserting the suffix t^p into the already sorted set of suffixes $\{t^{p+1}, t^{p+2}, \ldots, t^{n-1}\}$.

3.1 Preparations: Algorithms Walk-L and Walk-R

The algorithms in this subsection serve as preparations to explain the algorithms in the next subsection, which combine them in different ways to achieve different space/time trade-offs. We now describe round p of the algorithm Walk-L that performs the insertion of suffix t^p into the suffix array of t^{p+1} by walking left along the BWT of t^{p+1}.

At the beginning of round p, the arrays $\mathsf{lexfirstpos}[p+1, \ldots, n-1]$, $\mathsf{lexlastpos}[p+1, \ldots, n-1]$, $\mathsf{lexfirstpos}[]$ and $\mathsf{lexlastpos}[]$ represent the suffix array of the text t^{p+1}. There are two possibilities, based on t_p.

1. The character t_p does not occur in t^{p+1} ($\mathsf{lexfirstpos}[t_p] = \mathsf{lexlastpos}[t_p] = \perp$). We only need to find the largest character $c^- < t_p$ for which $p^- := \mathsf{lexlastpos}[c^-] \neq \perp$ and the smallest character $c^+ > t_p$ for which $p^+ := \mathsf{lexfirstpos}[c^+] \neq \perp$. Since presently $p^- \prec^*_{t^{p+1}} p^+$, we have $\mathsf{lexnextpos}[p^-] = p^+$ and $\mathsf{lexprevpos}[p^+] = p^-$. To reflect the new situation $p^- \prec^*_{t^p} p \prec^*_{t^p} p^+$, we update $\mathsf{lexnextpos}[p^-] \leftarrow p$, $\mathsf{lexprevpos}[p] \leftarrow p^-$, $\mathsf{lexnextpos}[p] \leftarrow p^+$, $\mathsf{lexprevpos}[p^+] \leftarrow p$, and set $\mathsf{lexfirstpos}[t_p] \leftarrow \mathsf{lexlastpos}[t_p] \leftarrow p$.
2. The character t_p already occurs somewhere in t^{p+1}. The following lemma contains the main idea.

Lemma 1. *(1) If there exists $q > p$ such that $c := t_p = t_q$ and $q \prec^*_{t^p} p$, then $t^{q+1} \prec t^{p+1}$, and all $i > p$ that satisfy $t^{q+1} \prec t^i \prec t^{p+1}$ also satisfy $t_{i-1} \neq c$.*

(2) Conversely, if there exists $q > p$ with $c := t_p = t_q$, if $t^{q+1} \prec t^{p+1}$, and there is no $i > p$ with $t_{i-1} = c$ that satisfies $t^{q+1} \prec t^i \prec t^{p+1}$, then $q \prec^*_{t^p} p$.

Proof. (1) states that if t^q directly precedes t^p in the suffix array of t^p, and both start with the same letter c, then the suffix t^{p+1} must also (but not necessarily directly) precede t^{q+1}. All the (potential) intermediate suffixes t^i cannot have c as preceding character; otherwise, by the definition of lexicographic order, the corresponding t^{i-1} would have to be between t^p and t^q. (2) is similar. □

The lemma leads to an algorithm to find the said position q if it exists. Starting at position $p+1$ (the last suffix inserted), follow the lexprevpos links to the left in the simulated linked list. Thus, let $i \leftarrow \mathsf{lexprevpos}[p+1]$, and then check whether $t_{i-1} = c$. If not, continue following the lexprevpos links ($i \leftarrow \mathsf{lexprevpos}[i]$). If eventually $t_{i-1} = c$, insert p into the list after $i-1 =: p^-$. In other words, writing $p^+ := \mathsf{lexnextpos}[p^-]$, we update lexnextpos and lexprevpos to reflect the update from $p^- \prec^*_{t^{p+1}} p^+$ to $p^- \prec^*_{t^p} p \prec^*_{t^p} p^+$, just as in case 1 above.

Additionally, if previously $\mathsf{lexlastpos}[c] = p^-$, we must update $\mathsf{lexlastpos}[c] \leftarrow p$. On the other hand, if we eventually fall off the list ($i = \perp$), position t^p must be the lexicographically first suffix starting with c, and we update the arrays accordingly.

A small technical complication arises when the sought q equals $n-1$, in which case t^{q+1} is not well defined in the lemma. To avoid this, we adopt the customary convention that the text ends with a unique special character $ that is smaller than any character in the alphabet.

This concludes the description of Walk-L. The algorithm Walk-R is perfectly symmetrical; now i follows the $\mathsf{lexnextpos}$ links starting at $p+1$, and we insert position p *before* the first position $i-1$ for which $t_{i-1} = c$.

The algorithms Walk-L and Walk-R thus walk (to the left or right) along the BWT of t^{p+1} to insert the suffix t^p at the correct position; see Figure 1.

An elementary analysis shows that the worst-case complexity of these algorithms in $O(n^2)$. One can show that for Walk-L and odd string length n, the worst binary string is $(01)^{\lfloor n/2 \rfloor}1$ with $n^2/8 + 3n/2 - 13/8$ steps; for even n the worst string is $(01)^{n/2-1}10$ with $n^2/8 + 3n/2$ steps.

3.2 Algorithms Walk-minLR and Walk-bothLR

Walk-minLR. Instead of walking only left or right in the BWT of t^{p+1} to insert p, we walk both ways in an alternating manner and stop as soon as we find the target character in either direction. We call this algorithm Walk-minLR. Surprisingly, this simple modification considerably improves the running time.

Theorem 1. *Algorithm* Walk-minLR *needs* $O(n \log n)$ *time on texts of length* n *over a binary alphabet.*

Proof. (1) We define a *BWT block* as a consecutive run of the same character in the BWT. In Figure 1 (left), after inserting t^3 and noting that t^3 will eventually be preceded by 1, the BWT consists of 6 blocks of lengths $2, 4, 1, 3, 1, 10$, respectively.

(2) We define a potential function $\Phi(r)$, where $r = 1, \ldots, n-1$ denotes the step of inserting suffix t^{n-r}. Let $\Phi(0) := 1$ and $\Phi(r) := 2r \log_2 r + 3r + 1 - 2 \cdot \sum_{i=1}^{r} \phi_r(i)$, where $\phi_r(i)$ denotes the base-2 logarithm of the length of the BWT block containing the predeceding character of suffix t^{n-i} ($i = 1, \ldots, r$) after round r, i.e., after t^{n-r} has been correctly placed (and the character t^{n-r-1} determined). In Figure 1 (left), $r = 21$, and $\Phi(r) = 42 \log 21 + 64 - 2 \cdot (2 \log 2 + 4 \log 4 + 0 + 3 \log 3 + 0 + 10 \log 10)$; all logs are base-2.

(3) We show that the difference $\Phi(r) - \Phi(r-1)$ is at least the number of steps required to insert suffix t^{n-r-1} in round $r+1$. Since $\Phi(0) = 1$, by induction, $\Phi(n-1)$ bounds the number of steps to process the whole text. It is evident that $3n + 1 \le \Phi(n) \le 2n \log_2 n + 3n + 1$, establishing the $O(n \log n)$ bound.

We consider two cases, according to what happens in round r when a character is inserted into the BWT. (a) The length of a single existing BWT block grows by 1. In this case, it can be seen that $\Phi(r) - \Phi(r-1) \ge 3$. Also, in

	11111111112222
Position p	01234567890123456789023
Text $t = (t_p)$	10100000100010000000001$

$p = \mathsf{lexnextpos}[p']$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$p' = \mathsf{lexprevpos}[p]$	?	?	?	17	18	9	10	11	12	19	20	21	22	23	13	14	15	16	3	4	5	6	7	⊥

```
23: [1] $                          23: [1] $
13: [1] 0000000001$                13: [1] 0000000001$
14: [0] 000000001$                 14: [0] 000000001$
15: [0] 00000001$                  15: [0] 00000001$
16: [0] 0000001$                   16: [0] 0000001$
17: [0] 000001$                    17: [0] 000001$
 3: [ ] 000001000100000000001$     3: [1] 000001000100000000001$
18: [0] 00001$                     18: [0] 00001$
 4: [0] 00001000100000000001$      4: [0] 00001000100000000001$
19: [0] 0001$                      19: [0] 0001$
 9: [1] 00010000000001$            9: [1] 00010000000001$
 5: [0] 00010001000000000001$      5: [0] 00010001000000000001$
20: [0] 001$                       20: [0] 001$
10: [0] 0010000000001$             10: [0] 0010000000001$
 6: [0] 00100010000000001$         6: [0] 00100010000000001$
21: [0] 01$                        21: [0] 01$
11: [0] 010000000001$              11: [0] 010000000001$
 7: [0] 01000100000000001$         7: [0] 01000100000000001$
22: [0] 1$                         22: [0] 1$
12: [0] 10000000001$               12: [0] 10000000001$
 8: [0] 100010000000001$           2: [ ] 10000010001000000001$
                                   8: [0] 100010000000001$
```

Fig. 1. Illustration of Walk-L and Walk-R. *Top*: Text with positions. *Middle*: $\mathsf{lexprevpos}$ array after inserting t^3. *Left*: Visualization of the suffix array after inserting t^3. Numbers in square brackets denote the preceding character; read from top to bottom, this spells the BWT of t^3, except for the empty square bracket, which will contain $t_2 = \mathtt{1}$ that has not yet been looked at. Now, to insert position 2 into the array, Walk-L walks left (up) from the current position (3:), following the $\mathsf{lexprevpos}$-links $3 \to 17 \to 16 \to 15 \to 14 \to 13$, looking at the characters preceding the suffixes at positions 17, 16, 15, 14, and 13 (00001). Since the 1 is found preceding position 13, position 2 must be inserted after 12. Alternatively, Walk-R walks right (down), looking at the characters preceding the suffixes at position 18, 4, 19, 9 (0001), and decides that 2 must be inserted before 8. *Right*: Result after inserting t^2 between 12 and 8.

round $r + 1$, Walk-minLR takes at most 3 steps (look left, look right, insert), since the looked-for character will immediately found in at least one direction. As an example, consider Figure 1 (right), where the long 0-block at the end of the BWT is lengthened from 10 to 11 after inserting 2 and realizing that $t_1 = \mathtt{0}$. In the next step, t^1 is immediately inserted between 11 and 7. (b) A block of length x is split into three blocks of lengths $x_1, 1, x_2$, respectively, where $x_1 + x_2 = x$. W.l.o.g., let $x_1 \leq x_2$, so $x_1 \leq x/2$. In round $r + 1$, the algorithm will need at most $2x_1 + 3$ steps to locate the searched-for character. In this case,

$\Phi(r)-\Phi(r-1) = 2(r\log r-(r-1)\log(r-1))+3-2(x_1\log x_1+x_2\log x_2-x\log x) \geq 3+2(x\log x-x_1\log x_1-x_2\log x_2) \geq 3+2(x\log x-x_1\log x/2-x_2\log x) \geq 3+2x_1$. As an example, in Figure 1 (left), a 0-block of length 7 is split into a 0-block of length 4, a singleton 1, and another 0-block of length 3, respectively. The number of steps taken in round $r+1$ has been accounted for in the potential function in both cases, and the bound is established. □

To generalize the theorem to larger alphabets, we may encode these texts as bit sequences over a binary alphabet. This lengthens the text by a constant factor of $\log|\Sigma|$. From the resulting suffix array, we only keep those positions that correspond to beginnings of actual characters in the original text.

Walk-bothLR. The final option is to walk left and right in the BWT until the target character is found on *both* sides. The required time is the sum of the running times of Walk-L and Walk-R and hence quadratic. At first sight, this variant seems useless.

However, we can now use an old folklore trick to store both lexnextpos and lexprevpos in a single array lexxorpos$[p]$:= lexprevpos$[p]$ xor lexnextpos$[p]$, reducing the total amount of working space from $4n+O(1)$ bytes to $O(1)$. Knowing one of the two values lexprevpos$[p]$, lexnextpos$[p]$, the other one can be derived by xor-ing the known value with lexxorpos$[p]$. In particular, we can still follow the links from position $p+1$ to the left and to the right, since we just inserted $p+1$ into the list in the previous step.

A difficulty arises when the cursor i reaches the point where $t_{i-1} = t_p$. Assume we moved only to the left, and now we want to insert p after $p^- := i-1$, as described above. This is impossible, because we neither know lexnextpos$[p^-]$ nor lexprevpos$[p^-]$. This is why we also move right: we obtain the index i' such that $p^- \prec^*_{tp} p \prec^*_{tp} p^+ := i'-1$; so we know that lexnextpos$[p^-] = p^+$ and lexprevpos$[p^+] = p^-$ and can update the p^-, p, and p^+ values of the xor-ed array to reflect the insertion of p:

1: lexxorpos$[p^-] \leftarrow$ lexxorpos$[p^-]$ xor p^+ xor p
2: lexxorpos$[p] \leftarrow p^-$ xor p^+
3: lexxorpos$[p^+] \leftarrow$ lexxorpos$[p^+]$ xor p^- xor p

Also, as we show in Section 5, the running time of both Walk-minLR and Walk-bothLR is linear for many practcial cases, including (and especially for) strings with long repeats. The running time of the in-place Walk-bothLR is often only slightly more than twice that of Walk-minLR.

4 Downstream Computations

Using either the faster Walk-minLR that needs $9n+O(1)$ bytes of memory ($4n+O(1)$ bytes of working space) or the slower in-place Walk-bothLR that needs $5n+O(1)$ bytes of memory (only a constant amount of working space), we have constructed a linked-list representation of the suffix array and need to transform it into the usual arrays defined in Section 2. Here we describe how to

do it without using additional memory resources. We may use secondary storage, but only consecutive left-to-right read/write access.

Two in-memory integer arrays. Let us start with the WALK-MINLR case where we can work with two integer arrays; initially these contain lexprevpos and lexnextpos. The array lexprevpos can be overwritten with longest common prefix information. The following code fragment is based on Kasai et al.'s linear-time lcp algorithm [14]. Note that instead of requiring both arrays pos and rank and creating lcp (which would need $13n$ bytes of memory, including the text), we only need lexprevpos and overwrite lexprevpos$[p]$ with lcp at pos$[p]$:= lcp[rank$[p]$] for all p. To obtain the correctly sorted lcp array in rank-order, we need to sort these values differently (see below).

PREPARE-LCP:

1: $\ell \leftarrow 0$
2: **for** $(p \leftarrow 0;\ p < n;\ p \leftarrow p + 1)$ **do**
3: $\quad p^- \leftarrow$ lexprevpos$[p]$ $\quad$ [▷ Kasai et al.: $p^- \leftarrow$ pos[rank$[p] - 1$]]
4: $\quad \ell \leftarrow \ell +$ prefix-match-length$(t^{p^- + \ell}, t^{p+\ell})$
5: $\quad$ lcp at pos$[p] \leftarrow \ell$ $\quad$ [▷ Kasai et al.: lcp[rank$[p]$] $\leftarrow \ell$]
6: $\quad \ell \leftarrow \max\{\ell - 1, 0\}$

The function prefix-match-length(u, v) does a character-by-character comparison of the prefixes of u and v and returns the length of their longest common prefix. (If in the above algorithm, $p^- = \perp$ or the positions $p^- + \ell$ or $p + \ell$ fall outside the text, it returns 0.) Note that in line 5, lcp at pos$[p]$ is assumed to point to the same memory location as lexprevpos$[p]$, so the array is overwritten in-place.

The comments in the algorithm show the version of [14]. Our variation is a pos-based counterpart of the space-saving trick proposed by Manzini in [17] that uses an auxiliary array that for a given lexicographic rank r computes the rank of the suffix at position pos$[r] + 1$. Since we have lexprevpos directly available, we do not need to build any additional auxiliary arrays.

In the next step, we walk through the positions in lexicographic order using the lexnextpos links and directly write pos, lcp and the BWT $\widehat{t}$ to secondary storage. We also overwrite lexnextpos with rank in memory during the process (and write the final rank array to secondary storage as well at the end). We begin at p_0 := lexfirstpos$[c]$, where c is the smallest character in Σ.

WRITEARRAYS:

1: $r \leftarrow 0$; $p \leftarrow p_0$
2: **while** $p \neq \perp$ **do**
3: $\quad p^+ \leftarrow$ lexnextpos$[p]$
4: $\quad$ rank$[p] \leftarrow r$ $\quad$ [▷ rank and lexnextpos share the same memory locations]
5: $\quad$ **write** p to file pos
6: $\quad$ **write** lcp at pos$[p]$ to file lcp
7: $\quad$ **write** t_{p-1} to file bwt $\quad$ [▷ Here $t_{-1} := t_{n-1}$]
8: $\quad r \leftarrow r + 1$; $p \leftarrow p^+$
9: **write** the complete array rank[] to file rank

In line 6, we assume that lcp at pos$[p]$, has been created using PREPARE-LCP. Since we traverse the positions p in lexicographic order, it is evident that we store the values in the files pos, lcp, bwt in the correct order.

One in-memory integer array. We now produce the same arrays as above with only one (random-access) integer array in main memory; this appears to be the first description of a practical method that uses only $5n + O(1)$ bytes of main memory (and only consecutive access to secondary storage) to construct all above-mentioned arrays. We assume that lexxorpos and the text are in memory.

First, note that we can run WRITEARRAYS without line 6 with no difficulty; thus creating the rank file; it remains to construct the lcp array.

WRITEARRAYS+:

1: **write** array lexxorpos to file lexxorpos
2: execute WRITEARRAYS without line 6 (this creates the rank file)
3: **load** file lexxorpos to restore lexxorpos into memory
4: traverse lexxorpos in reverse lexicographic to overwrite it with lexprevpos
5: **write** array lexprevpos to file lexprevpos, overwriting file lexxorpos
6: Note: the in-memory array will now be used to build lcp
7: $\ell \leftarrow 0$; rewind files lexprevpos and rank
8: **for** $(p \leftarrow 0;\ p < n;\ p \leftarrow p + 1)$ **do**
9: $\quad p^- \leftarrow$ next integer from file lexprevpos
10: $\quad r \leftarrow$ next integer from file rank
11: $\quad \ell \leftarrow \ell +$ prefix-match-length$(t^{p^- + \ell}, t^{p+\ell})$
12: $\quad$ lcp$[r] \leftarrow \ell$
13: $\quad \ell \leftarrow \max\{\ell - 1, 0\}$
14: **write** array lcp to file lcp, overwriting file lexprevpos

5 Empirical Results

Implementation and source code. The VerJInxer software is a **Ver**satile **J**ava-based **In**de**xer** and can be obtained under the Artistic License / GPLv2 at verjinxer.googlecode.com. It includes Java5 implementations of the presented algorithms. The design of the (enhanced) suffix array as a collection of arrays stored in different files in VerJInxer is similar to Stefan Kurtz' `vmatch` (http:///www.vmatch.de). Like `vmatch`, VerJInxer also supports alphabet transformations, and by using special characters (that are compared by position instead of by value), it allows to build an index of several strings at once. Unlike `vmatch`, VerJInxer is open source.

Expected practical behavior. While the worst case complexity of WALKMINLR is $O(n \log n)$ and that of WALK-BOTHLR is $O(n^2)$, we expect both algorithms to need only linear time in practice. Consider a random text where each letter has the same independent occurrence probability $1/|\Sigma|$ at each position. If all strings in the sequence of BWTs $(\widehat{t^{n-1}}, \ldots, \widehat{t^1}, \widehat{t^0})$ also behave like

random strings, we expect to need on average $|\Sigma|$ array lookups and character comparisons to find the target character in each round. This leads to an expected running time of $n|\Sigma|$ steps (twice as much for WALK-BOTHLR), suggesting excellent behavior for small alphabets, such as genomic sequences. The argument also holds for random texts with skewed character distributions: less frequent characters take longer to find, but are looked for less frequently.

Measured practical behavior. We have instrumented the code to count the number of array lookups and character comparisons. The algorithms are executed on a single processor of a new two-processor Intel Core2 Duo PC at 2.66 GHz with 3 GB of RAM, and for comparison, on a single 900 MHz processor of a Sun-Fire V1280 UltraSparc-III processor with 96 GB of RAM. The same machine was used in [7] for large-scale genome experiments. A detailed comparison of many algorithms was recently published in [7], and we compare the new algorithms to the best ones mentioned there, mainly BPR, and DEEPSHALLOW [8].

Random strings and π. Testing on large amounts of random strings (with both uniform and non-uniform character probabilities) for all alphabet sizes $|\Sigma| \in \{2, \ldots, 255\}$ reveals no surprise: For WALK-L, WALK-R, and WALK-MINLR, the number of lookups is almost identical to $|\Sigma|n$ (for WALK-BOTHLR, to $2|\Sigma|n$), with negligible standard deviation, as predicted. This confirms that BWTs of random strings behave again like random strings. The same statements apply to the first million decimal digits of π with $|\Sigma| = 10$. As real-time performance is concerned, it is evident that the present algorithms can only compete for very small alphabets. For random binary texts of 100 MB, WALK-MINLR and WALK-BOTHLR need 18.0 and 33.4 sec on the PC, respectively, while BPR and DEEPSHALLOW, which performed equally well on random strings in [7], need 25 sec. We note that the new algorithms are implemented in Java, whereas the others are implemented in C/C++.

Fibonacci strings. We define the k-th Fibonacci string S_k over $\Sigma = \{a, b\}$ by $S_0 := b$, $S_1 := a$, and $S_k := S_{k-1}S_{k-2}$ for $k \geq 2$. The length of S_k is F_k, the k-th Fibonacci number. For example, S_{36} has length 24 157 817 (24 MB text).

For odd k, both WALK-L and WALK-R need $\approx 1.38\,F_k$ steps; WALK-BOTHLR thus needs $2.76\,F_k$ steps; but WALK-MINLR needs only $1.00\,F_k$ steps.

For even k, WALK-L needs $1.00\,F_k$ steps; WALK-R about $2.09\,F_k$ steps; WALK-BOTHLR $3.09\,F_k$ steps; and WALK-MINLR again only $1.00\,F_k$ steps.

On the PC, the computation for S_{36} takes 0.6 seconds. These times scale linearly for longer Fibonacci strings. WALK-MINLR is thus the fastest method for Fibonacci strings. In [7], it is reported that BPR and DIVIDE&CONQUER are the fastest algorithms for Fibonacci strings so far; they both need over 10 seconds on our PC. Of course, our algorithms are, in a sense, engineered for Fibonacci strings, and especially benefit from the small alphabet size.

English text. The text of the bible forms an $n = 4$ MB file over an alphabet of size 93. The large alphabet size becomes problematic (this could be remedied by

binary encoding): 75.7 n steps for WALK-L, 67.8 n for WALK-R, and 25.73 n for WALK-MINLR; the latter takes approximately 5.4 seconds on the PC, clearly *not* competitive with the performance of other algoithms reported in [7], the best of which (DEEPSHALLOW) is lightweight and needs less than 1 second.

DNA and repetitive DNA. The results so far suggest that the algorithms should perform well on DNA sequences. Table 1 (left) shows statistics for indexing collections of different strains of bacteria; the strains of the same species are quite similar but not identical, so long repeated regions occur. In [7], the best algorithms on DNA were BPR and DEEPSHALLOW [8], with comparable performance, while DEEPSHALLOW is lightweight, requiring slightly more than $5n$ bytes of memory. The WALK-algorithms are competitive, considering that they are implemented in Java (vs. C/C++ for the others). Table 1 (right) shows statistics for the human genome. Not many implementations construct the suffix array of a 500 Mbp sequence on a 3 GB PC due to their memory requirements. As sequence length or repetitiveness increase, BPR and DEEPSHALLOW require more time per input character, while for the WALK-algorithms, time grows approximately linearly. This allows us to estimate the time required for the whole human genome by extrapolation (a factor of 12.5 from chromosome 1, or 20.7 from chromosome 8, which we double to account for `long` integers in this case), resulting in the proposed estimates, which compare favorably to the reported running time of the 64-bit version of BPR on the same server in [7].

6 Concluding Discussion

We have presented two variations WALK-MINLR and WALK-BOTHLR with different space-time trade-offs for constructing the (enhanced) suffix array pos, rank, lcp and BWT $\widehat{t}$ of a text t. Most notably, we have proved that walking in both directions alternatingly improves the worst-case time from $O(n^2)$ to $O(n \log n)$.

Table 1. *Left*: Statistics for indexing collections of genomes from different strains of bacteria (3 *E. coli* strains, 4 *Chlamydophila* strains, 6 *Streptococci* strains) Times are measured on the PC. *Right*: Human genome data; times are measured on the server. —: unable to measure because we only had the 32-bit version available. *: estimated by extrapolation. The 429 minutes are from [7]. *Abbreviation*: Mbp = 10^6 base pairs.

Data set [7]	3Ec	4Ch	6St
length n [Mbp]	14.78	4.86	11.64
Steps / n			
WALK-MINLR	2.61	3.46	3.55
WALK-BOTHLR	7.01	9.31	8.66
Time [sec]; PC			
WALK-MINLR	3.4	1.2	3.9
WALK-BOTHLR	8.3	3.1	8.2
BPR	3.1	0.8	2.1
DEEPSHALLOW	6.0	1.3	2.6

Human	chr8	chr1	$(\text{chr1})^2$	genome
length n [Mbp]	149	247	494	3080
Steps / n				
WALK-MINLR	3.76	3.58	2.57	—
WALK-BOTHLR	7.90	7.71	7.04	—
Time [sec]; Server				
WALK-MINLR	242	386	508	*160 min
WALK-BOTHLR	451	731	1354	*300 min
BPR	160	—	—	429 min
DEEPSHALLOW	192	359	—	—

While the relations between lexnextpos, the BWT, compressed and uncompressed suffix arrays are well known, we are not aware that a direct simple (in-place) suffix array construction algorithm based on these principles has been previously described or implemented. Our main result (which we do not believe has been stated as clearly before) is thus: *Given N bytes of available main memory, we can construct tables* pos, rank, lcp, *and* bwt *of the enhanced suffix array of a text of length $n = N/5$ in $O(|\Sigma|n)$ expected time.*

The main practical drawback of the proposed algorithms is their abysmal cache performance. Each walk through lexprevpos or lexnextpos constitutes of random memory accesses and will cause cache faults in long texts.

The strength of the algorithms is on long (and potentially repetitive) DNA sequences: extrapolations indicate that the presented algorithm could be the method of choice for the whole human genome. To prove this, however, it will be necessary to re-implement the method in C, where arrays with more than 2G elements are available and we expect another speedup of a factor of at least 2, compared to the times in Table 1.

Acknowledgments. We thank several members of the Genome Informatics and Practical Informatics groups, and the DFG graduate program GK Bioinformatik, at Bielefeld University, in particular Klaus-Bernd Schürmann, Robert Homann, Peter Husemann, Wolfgang Gerlach, Jens Stoye, and Robert Giegerich. Tobias Marschall and Marcel Martin gave valuable comments on the manuscript.

References

1. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys **39**(2) (2006) Article 4
2. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5) (1993) 935–948
3. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6) (2006) 918–936
4. Kim, D.K., Sim, J.S., Park, H., Park, K.: Constructing suffix arrays in linear time. Journal of Discrete Algorithms **3**(2-4) (June 2005) 126–142
5. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Discrete Algorithms **3**(2-4) (2005) 143–156
6. Kim, D.K., Jo, J., Park, H.: A fast algorithm for constructing suffix arrays for fixed-size alphabets. In Ribeiro, C.C., Martins, S.L., eds.: Proceedings of the 3rd International Workshop on Experimental and Efficient Algorithms (WEA). Volume 3059 of Lecture Notes in Computer Science., Springer (2004) 301–314
7. Schürmann, K.B., Stoye, J.: An incomplex algorithm for fast suffix array construction. Software: Practice and Experience **37**(3) (2006) 309–329
8. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**(1) (2004) 33–50
9. Franceschini, G., Muthukrishnan, S.: In-place suffix sorting. In: Proceedings of ICALP. Volume 4596 of LNCS., Springer (2007) 533–545
10. Nong, G., Zhang, S.: Optimal lightweight construction of suffix arrays for constant alphabets. In: Proceedings of WADS. Volume 4619 of Lecture Notes in Computer Science., Springer (2007) 613–624

11. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. In: FOCS, IEEE Computer Society (2003) 251–260
12. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In Ziviani, N., Baeza-Yates, R.A., eds.: SPIRE. Volume 4726 of Lecture Notes in Computer Science., Springer (2007) 229–241
13. Ross A. Lippert, Clark M. Mobarry, B.P.W.: A space-efficient construction of the burrows-wheeler transform for genomic data. Journal of Computational Biology **12**(7) (2005) 943–951
14. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, London, UK, Springer (2001) 181–192
15. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1) (2004) 53–86
16. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory. (1973) 1–11
17. Manzini, G.: Two space saving tricks for linear time lcp array computation. In Hagerup, T., Katajainen, J., eds.: SWAT. Volume 3111 of Lecture Notes in Computer Science., Springer (2004) 372–383