Information and Computation 213 (2012) 13–22

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Bidirectional search in a string with wavelet trees and bidirectional matching statistics

Thomas Schnattinger*, Enno Ohlebusch, Simon Gog

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm, Germany

ARTICLE INFO

Article history:
Available online 2 February 2012

Keywords:
String matching
Bidirectional search
Matching statistics

ABSTRACT

Searching for genes encoding microRNAs (miRNAs) is an important task in genome analysis. Because the secondary structure of miRNA (but not the sequence) is highly conserved, the genes encoding it can be determined by finding regions in a genomic DNA sequence that match the structure. It is known that algorithms using a bidirectional search on the DNA sequence for this task outperform algorithms based on unidirectional search. The data structures supporting a bidirectional search (affix trees and affix arrays), however, are rather complex and suffer from their large space consumption. Here, we present a new data structure called *bidirectional wavelet index* that supports bidirectional search with much less space. With this data structure, it is possible to search for candidates of RNA secondary structural patterns in large genomes, for example the complete human genome. Another important application of this data structure is short read alignment. As a second contribution, we show how *bidirectional matching statistics* can be computed in linear time.

1. Introduction

It is now known that microRNAs (miRNAs) regulate the expression of many protein-coding genes and that the proper functioning of certain miRNAs is important for preventing cancer and other diseases. microRNAs are RNA molecules that are encoded by genes from whose DNA they are transcribed, but they are not translated into protein. Instead each primary transcript is processed into a secondary structure (consisting of approximately 70 nucleotides) called a pre-miRNA and finally into a functional miRNA. This so-called mature miRNA is 21–24 nucleotides long, so a gene encoding a miRNA is much longer than the processed mature miRNA molecule itself. Mature miRNA molecules are either fully or partially complementary to one or more messenger RNA (mRNA) molecules, and their main function is to down-regulate gene expression. The first miRNA was described by Lee et al. [1], but the term miRNA was only introduced in 2001 when the abundance of these tiny regulatory RNAs was discovered; see [2] for an overview. miRNAs are highly conserved during evolution, not on the sequence level, but as secondary structures. Thus, the task of finding the genes coding for a certain miRNA in a genome is to find and further investigate all regions in the genomic DNA sequence that match its structural pattern. Because the structural pattern often consists of a hairpin loop and a stem (which may also have bulges), the most efficient algorithms first search for candidate regions matching the loop and then try to extend both ends by searching for complementary base pairs A–U, G–C, or G–U that form the stem. Because T (thymine) is replaced with U (uracil) in the transcription from DNA to RNA, one must search for the pairs A–T, G–C, or G–T in the DNA sequence. For example, if the loop is the sequence GGAC, then it is extended by one of the four nucleotides to the left or to the right, say by G to the right, and all regions in the DNA sequence matching GGACG are searched for (by forward search). Out of these candidate regions only those survive that

* Corresponding author. Fax: +49 731 50 24102.
E-mail addresses: thomas.schnattinger@uni-ulm.de (T. Schnattinger), enno.ohlebusch@uni-ulm.de (E. Ohlebusch), simon.gog@uni-ulm.de (S. Gog).

0890-5401/$ – see front matter
doi:10.1016/j.ic.2011.03.007

i	SA	BWT	$S_{SA[i]}$
1	19	n	$
2	3	l	_anele_lepanelen$
3	9	e	_lepanelen$
4	4	_	anele_lepanelen$
5	13	p	anelen$
6	8	l	e_lepanelen$
7	1	$	el_anele_lepanelen$
8	6	n	ele_lepanelen$
9	15	n	elen$
10	17	l	en$
11	11	l	epanelen$
12	2	e	l_anele_lepanelen$
13	7	e	le_lepanelen$
14	16	e	len$
15	10	_	lepanelen$
16	18	e	n$
17	5	a	nele_lepanelen$
18	14	a	nelen$
19	12	e	panelen$

Fig. 1. Suffix array and Burrows–Wheeler-transform of $T =$ el_anele_lepanelen$.

can be extended by C or T to the left because only C and T (U, respectively) form a base pair with G, and the stem is formed by complementary base pairs. In other words, in the next step one searches for all regions in the DNA sequence matching either CGGACG or TGGACG (by backward search). Such a search strategy can be pursued only if bidirectional search is possible. Mauri and Pavesi [3] used affix trees for this purpose, while Strothmann [4] employed affix arrays.

Research on data structures supporting bidirectional search in a string started in 1995 with Stoye's diploma thesis on affix trees (the English translation appeared in [5]), and Maaß [6] showed that affix trees can be constructed on-line in linear time. Basically, the affix tree of a string S comprises both the suffix tree of S (supporting forward search) and the suffix tree of the reverse string S^{rev} (supporting backward search). It requires approximately $45n$ bytes, where n is the length of S. Strothmann [4] showed that affix arrays have the same functionality as affix trees, but they require only $18n$–$20n$ bytes (depending on the implementation). An affix array combines the suffix arrays of S and S^{rev}, but it is a complex data structure because the interplay between the two suffix arrays is rather difficult to implement. In this article, we present a new data structure called *bidirectional wavelet index* that consists of the wavelet tree of the Burrows–Wheeler transformed string of S (supporting backward search) and the wavelet tree of the Burrows–Wheeler transformed string of S^{rev} (supporting forward search). In contrast to affix arrays, however, the interplay between the two is easy to implement. Our experiments show that the bidirectional wavelet index decreases the space requirement by a factor of 23 (compared to affix arrays), making it possible to search bidirectionally in very large strings.

Independently and contemporaneously, Lam et al. [7] presented a similar data structure, which they call *bidirectional BWT*. Their main motivation was short read alignment, so they use bidirectional search to find approximate matches of relatively short DNA sequences within a whole genome. It turns out that the two approaches use the same basic idea, but a closer look reveals that none is superior to the other. On the one hand, one search step with the bidirectional wavelet index takes $O(\log\sigma)$ time while it takes $O(\sigma)$ time with the bidirectional BWT, where σ is the size of the underlying alphabet. On the other hand, the bidirectional BWT uses less space than the bidirectional wavelet index.

Very recently, it was shown that the *matching statistics ms* of a string S^2 w.r.t. another string S^1 can be computed in linear time by backward search [8], where $ms[i]$ is the length of the longest substring of S^2 starting at position i that matches a substring somewhere in S^1. Matching statistics were introduced by Chang and Lawler [9] in the context of approximate string matching. Among other things, they were used in the computation of string kernels [10] and the design of DNA chips [11]. We define $bms[i]$ to be the length of the longest substring of S^2 containing position i that matches a substring somewhere in S^1, and call these values the *bidirectional matching statistics* of S^2 w.r.t. S^1. In the second part of this article, we show how bidirectional matching statistics can be computed in linear time.

2. Preliminaries

Let Σ be an ordered alphabet whose smallest element is the so-called sentinel character \$. If Σ consists of σ characters and is fixed, then we may view Σ as an array of size σ such that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] = \$ < \Sigma[2] < \cdots < \Sigma[\sigma]$. In the following, S is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $1 \leqslant i \leqslant n$, $S[i]$ denotes the *character at position* i in S. For $i \leqslant j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the ith suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S, that is, it satisfies $S_{SA[1]} < S_{SA[2]} < \cdots < S_{SA[n]}$; see Fig. 1 for an example. In the following, SA^{-1} denotes the inverse of the permutation SA. The suffix array was introduced by Manber and Myers [12]. In 2003, it was shown independently and contemporaneously by three research groups that a direct linear

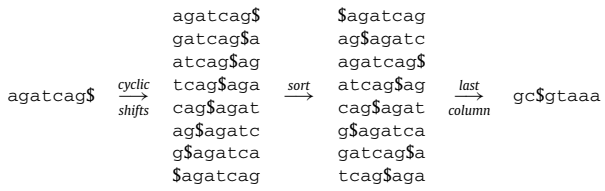

Fig. 2. Construction of the Burrows and Wheeler transform of the string `agatcag$`.

time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms are known; see [13] for details. Forward search on a suffix array can be done in $O(\log n)$ time per character by binary search [12] or in $O(\log \sigma)$ time per character using extra space [14].

2.1. Burrows and Wheeler transform

The reversible Burrows and Wheeler transform [15] has been developed as a first step in a data compression algorithm. It does not compress the data itself, but rearranges the input sequence into a more compressible form by tending to group same characters together. In virtually all cases, the Burrows–Wheeler transformed string compresses much better than the original string. It is today commonly used in data compression tools like *bzip2* [16] and in text indexing.

Given the input string $S[1..n]$, having the sentinel character \$ as its last character, the transformation consists of three steps; see Fig. 2 for an example.

1. Form a conceptual matrix $\mathcal{M}'$ whose rows are the cyclic shifts of string S.
2. Compute the matrix $\mathcal{M}$ by sorting the rows of $\mathcal{M}'$ lexicographically.
3. Construct the transformed string BWT by taking the last column of $\mathcal{M}$.

So the result of the transformation is a sequence BWT of n characters, where the i-th character is the last character of the i-th sorted rotation of S. Note the strong relationship between the Burrows and Wheeler transform and the suffix array SA of S. $\mathsf{SA}[k]$ specifies the suffix of lexicographical order k, which is a prefix of the k-th row of $\mathcal{M}$. Consequently, the string BWT can alternatively also be defined using SA as

$$\mathsf{BWT}[i] = \begin{cases} S[\mathsf{SA}[i]-1] & \text{if } \mathsf{SA}[i] \neq 1, \\ \$ & \text{otherwise.} \end{cases}$$

The first and the last column of the conceptual matrix $\mathcal{M}$ are usually called F and L, respectively. The mapping $LF(i)$ is defined by $LF(i) = C[c] + Occ(c, i)$ where $c = \mathsf{BWT}[i]$, $C[c]$ is the overall number (of occurrences) of characters in S which are strictly smaller than c, and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$. It is called *last-to-first mapping*, because the character $L[i]$ is located in F at position $LF(i)$. So it is clear that $LF(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i]-1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $LF(i) = 1$ otherwise, which explains its central role in the following index. Note that our array BWT is exactly the same as the column L of $\mathcal{M}$.

Details about the Burrows and Wheeler transform and related topics can for instance be found in [17].

2.2. Backward search

Ferragina and Manzini [18] showed that it is possible to search a pattern character-by-character backwards in the suffix array SA of string S, without storing SA. Their so-called FM-index can be implemented such that a search for a pattern of length m takes only $O(m)$ time [19], and it works as follows. The search process has two parts. The first one is called *backwardSearch* and determines the interval corresponding to the pattern. The second is *locate*, which identifies the positions of these occurrences in the text. Both operations are based on the array C and the function $Occ(c, i)$.

In the following, the ω-interval in SA of a substring ω of S is the interval $[i..j]$ such that ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \leqslant k \leqslant j$, but ω is not a prefix of any other suffix of S. So the *locate* step must return the values of SA in the interval $[i..j]$. Using the LF-mapping it suffices to store only n/α sampled values for some $\alpha > 1$ rather than the complete suffix array, because with the relationship $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ the missing values can be reconstructed in $\alpha/2$ steps (on average).

For example, the `le`-interval in the suffix array of Fig. 1 is the interval [13..15]. Searching backwards in the string $S =$ `el_anele_lepanelen`\$ for the pattern `le` works as follows. By definition, backward search for the last character of the pattern starts with the ε-interval $[1..n]$, where ε denotes the empty string. Algorithm 1 shows the pseudo-code of one backward search step. In our example, $backwardSearch(\mathtt{e}, [1..19])$ returns the `e`-interval [6..11] because $C[\mathtt{e}] + Occ(\mathtt{e}, 1-1) + 1 = 5 + 0 + 1 = 6$ and $C[\mathtt{e}] + Occ(\mathtt{e}, 19) = 5 + 6 = 11$. In the next step, $backwardSearch(\mathtt{l}, [6..11])$ delivers the `le`-interval [13..15] because $C[\mathtt{l}] + Occ(\mathtt{l}, 6-1) + 1 = 11 + 1 + 1 = 13$ and $C[\mathtt{l}] + Occ(\mathtt{l}, 11) = 11 + 4 = 15$.

In the last step *locate*, the three values of $\mathsf{SA}[13], \mathsf{SA}[14], \mathsf{SA}[15]$ have to be computed from the sampled suffix array. Suppose the sampling parameter α is 4, so the five values $\mathsf{SA}[1], \mathsf{SA}[5], \ldots, \mathsf{SA}[17]$ are stored explicitly. The first value $\mathsf{SA}[13]$ is one of the stored samples, so it can immediately be returned as a result. The second value $\mathsf{SA}[14]$ does not

Algorithm 1 Given $c \in \Sigma$ and an ω-interval $[i..j]$, *backwardSearch*$(c, [i..j])$ returns the $c\omega$-interval if it exists, and $\perp$ otherwise.

backwardSearch$(c, [i..j])$
  $i \leftarrow C[c] + Occ(c, i-1) + 1$
  $j \leftarrow C[c] + Occ(c, j)$
  if $i \leqslant j$ **then return** $[i..j]$
  else return $\perp$

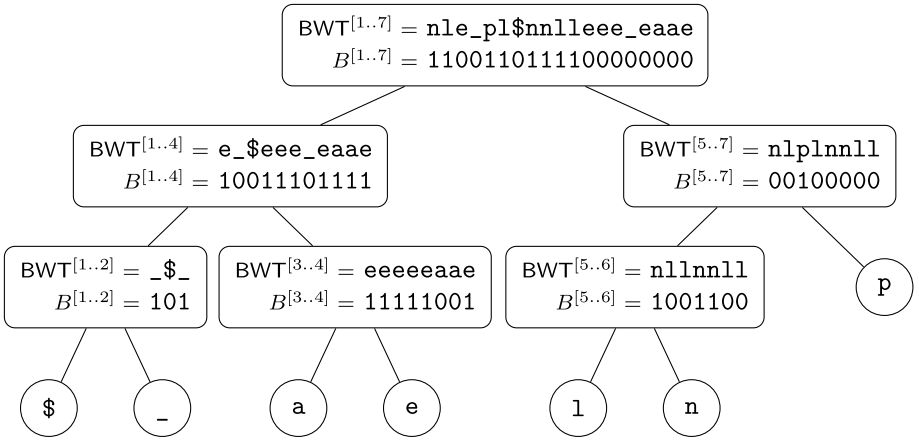

Fig. 3. Conceptual illustration of the wavelet tree of the string BWT = nle_pl$nnlleee_eaae. Only the bit vectors are stored; the corresponding strings are shown for clarity.

belong to the sampled values, so we check whether the value of SA at position $LF(14) = 9$ is stored. That being the case, we compute $\mathsf{SA}[14] = \mathsf{SA}[9] + 1 = 15 + 1 = 16$. This procedure is repeated for $\mathsf{SA}[15] = 10$. We get as a result that the pattern le occurs in S at positions 7, 10 and 16.

2.3. Wavelet tree

With the *wavelet tree* introduced by Grossi et al. [20], the computation of a value of the function Occ, and so each step of the backward search in string S, takes $O(\log\sigma)$ time as we shall see next. We say that an interval $[l..r]$ is an *alphabet interval*, if it is a subinterval of $[1..\sigma]$, where $\sigma = |\Sigma|$. For an alphabet interval $[l..r]$, the string $\mathsf{BWT}^{[l..r]}$ is obtained from the Burrows–Wheeler transformed string BWT of S by deleting all characters in BWT that do not belong to the sub-alphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. As an example, consider the string BWT = nle_pl$nnlleee_eaae and the alphabet interval [1..4]. The string $\mathsf{BWT}^{[1..4]}$ is obtained from nle_pl$nnlleee_eaae by deleting the characters l, n, and p. Thus, $\mathsf{BWT}^{[1..4]}$ = e_$eee_eaae.

The wavelet tree of the string BWT over the alphabet $\Sigma[1..\sigma]$ is a balanced binary search tree defined as follows. Each node v of the tree corresponds to a string $\mathsf{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\mathsf{BWT} = \mathsf{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child corresponds to the string $\mathsf{BWT}^{[l..m]}$ and its right child corresponds to the string $\mathsf{BWT}^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$. In this case, v stores a bit vector $B^{[l..r]}$ whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the sub-alphabet $\Sigma[l..m]$ and 1 if it belongs to the sub-alphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it belongs to the right subtree; see Fig. 3. Moreover, each bit vector B in the tree is preprocessed such that the queries $rank_0(B, i)$ and $rank_1(B, i)$ can be answered in constant time [21], where $rank_b(B, i)$ is the number of occurrences of bit b in $B[1..i]$. Obviously, the wavelet tree has height $O(\log\sigma)$. Because in an actual implementation it suffices to store only the bit vectors, the wavelet tree requires only $n\log\sigma$ bits of space plus $o(n\log\sigma)$ bits for the data structures that support rank-queries in constant time.

The query $Occ(c, i)$ can be answered by a top-down traversal of the wavelet tree in $O(\log\sigma)$ time. As an example, we compute $Occ(\mathtt{e}, 16)$ on the wavelet tree of the string BWT = nle_pl$nnlleee_eaae from Fig. 3. Because e belongs to the first half $\Sigma[1..4]$ of the ordered alphabet Σ, the occurrences of e correspond to zeros in the bit vector at the root, and they go to the left child, say node v_1, of the root. Now the number of e's in $\mathsf{BWT}^{[1..7]}$ = nle_pl$nnlleee_eaae up to position 16 equals the number of e's in the string $\mathsf{BWT}^{[1..4]}$ = e_$eee_eaae up to position $rank_0(B^{[1..7]}, 16)$. So we compute $rank_0(B^{[1..7]}, 16) = 8$. Because e belongs to the second quarter $\Sigma[3..4]$ of Σ, the occurrences of e correspond to ones in the bit vector at node v_1, and they go to the right child, say node v_2, of v_1. The number of e's in $\mathsf{BWT}^{[1..4]}$ = e_$eee_eaae up to position 8 is equal to the number of e's in $\mathsf{BWT}^{[3..4]}$ = eeeeeaae up to position $rank_1(B^{[1..4]}, 8) = 5$. In the third step, we must go to the right child of v_2, and the number of e's in $\mathsf{BWT}^{[3..4]}$ = eeeeeaae up to position 5

Algorithm 2 For a character c, an index i, and an alphabet interval $[l..r]$, the function $Occ'(c, i, [l..r])$ returns the number of occurrences of c in the string $\mathsf{BWT}^{[l..r]}[1..i]$, unless $l = r$ (in this case, it returns i).

```
Occ'(c, i, [l..r])
  if l = r then return i
  else
    m = ⌊(l+r)/2⌋
    if c ⩽ Σ[m] then
      return Occ'(c, rank_0(B^[l..r], i), [l..m])
    else
      return Occ'(c, rank_1(B^[l..r], i), [m+1..r])
```

i		$S_{\mathsf{SA}[i]}$
1	n	$
2	l	_anele_lepanelen$
3	e	_lepanelen$
4	_	anele_lepanelen$
5	p	anelen$
6	l	e_lepanelen$
7	$	el_anele_lepanelen$
8	n	ele_lepanelen$
9	n	elen$
10	l	en$
11	l	epanelen$
12	e	l_anele_lepanelen$
13	e	le_lepanelen$
14	e	len$
15	_	lepanelen$
16	e	n$
17	a	nele_lepanelen$
18	a	nelen$
19	e	panelen$

i		$S^{rev}_{\mathsf{SA}^{rev}[i]}$
1	e	$
2	l	_elena_le$
3	a	_le$
4	n	a_le$
5	n	apel_elena_le$
6	l	e$
7	p	el_elena_le$
8	_	elena_le$
9	n	elenapel_elena_le$
10	l	ena_le$
11	l	enapel_elena_le$
12	e	l_elena_le$
13	_	le$
14	e	lena_le$
15	e	lenapel_elena_le$
16	e	na_le$
17	e	napel_elena_le$
18	$	nelenapel_elena_le$
19	a	pel_elena_le$

Fig. 4. Bidirectional wavelet index of $S = \texttt{el\_anele\_ lepanelen\$}$, consisting of the backward index (left) and the forward index (right).

equals the number of e's in $\mathsf{BWT}^{[4..4]} = \texttt{eeeeee}$ up to position $rank_1(B^{[3..4]}, 5) = 5$. Since $\mathsf{BWT}^{[4..4]}$ consists solely of e's (by the way, that is the reason why it does not appear in the wavelet tree) the number of e's in $\mathsf{BWT}^{[4..4]}$ up to position 5 is 5. Pseudo-code for the computation of $Occ(c, i) = Occ'(c, i, [1..\sigma])$ can be found in Algorithm 2.

3. Bidirectional search

The *bidirectional wavelet index* of a string S consists of

- the *backward index*, supporting backward search based on the wavelet tree of the Burrows–Wheeler transformed string BWT of S, and
- the *forward index*, supporting backward search on the reverse string S^{rev} of S (hence forward search on S) based on the wavelet tree of the Burrows–Wheeler transformed string BWT^{rev} of S^{rev}.

The difficult part is to synchronize the search on both indexes. To see this, suppose we know the ω-interval $[i..j]$ in the backward index as well as the ω^{rev}-interval $[i^{rev}..j^{rev}]$ in the forward index, where ω is some substring of S. Given $[i..j]$ and a character c, $backwardSearch(c, [i..j])$ returns the $c\omega$-interval in the backward index (cf. Algorithm 1), but it is unclear how the corresponding interval, the interval of the string $(c\omega)^{rev} = \omega^{rev}c$, can be found in the forward index. Vice versa, given $[i^{rev}..j^{rev}]$ and a character c, backward search returns the $c\omega^{rev}$-interval in the forward index, but it is unclear how the corresponding ωc-interval can be found in the backward index. Because both cases are symmetric, we will only deal with the first case. So given the ω^{rev}-interval, we have to find the $\omega^{rev}c$-interval in the forward index. As an example, consider the bidirectional wavelet index of the string $S = \texttt{el\_anele\_lepanelen\$}$ in Fig. 4, and the substring $\omega = \texttt{e} = \omega^{rev}$. The e-interval in both indexes is [6..11]. The le-interval in the backward index is determined by $backwardSearch(\texttt{l}, [6..11]) = [13..15]$ and the task is to identify the el-interval in the forward index.

All we know is that the suffixes of S^{rev} are lexicographically ordered in the forward index. In other words, the $\omega^{rev}c$-interval $[p..q]$ is a subinterval of $[i^{rev}..j^{rev}]$ such that (note that $|\omega^{rev}| = |\omega|$)

- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] < c$ for all k with $i^{rev} \leqslant k < p$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] = c$ for all k with $p \leqslant k \leqslant q$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] > c$ for all k with $q < k \leqslant j^{rev}$.

In the example of Fig. 4,

- $S^{rev}[\mathsf{SA}^{rev}[k] + 1] = \$ < \mathtt{l}$ for $k = 6$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + 1] = \mathtt{l}$ for all k with $7 \leqslant k \leqslant 9$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + 1] = \mathtt{n} > \mathtt{l}$ for all k with $9 < k \leqslant 11$.

Unfortunately, we do not know these characters, but if we would know the number *smaller* of all occurrences of characters at these positions that precede c in the alphabet, together with the size of the new $c\omega$-interval $[i^{new}..j^{new}]$ in the backward index, then we could identify the unknown $\omega^{rev}c$-interval $[p..q]$ by $p = i^{rev} + smaller$ and $q = p + (j^{new} - i^{new})$. In our example, the knowledge of $smaller = 1$ and $[i^{new}..j^{new}] = [13..15]$ would yield the $\mathtt{el}$-interval $[6+1..(6+1)+(15-13)] = [7..9]$. The *key observation* is that the multiset of characters $\{S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] : i^{rev} \leqslant k \leqslant j^{rev}\}$ coincides with the multiset $\{\mathsf{BWT}[k] : i \leqslant k \leqslant j\}$. In the example of Fig. 4, $\{S^{rev}[\mathsf{SA}^{rev}[k] + 1]\colon\ 6 \leqslant k \leqslant 11\} = \{\$, \mathtt{l}, \mathtt{l}, \mathtt{l}, \mathtt{n}, \mathtt{n}\} = \{\mathsf{BWT}[k] : 6 \leqslant k \leqslant 11\}$. In other words, it suffices to determine the number *smaller* of all occurrences of characters in the string $\mathsf{BWT}[i..j]$ that precede character c in the alphabet Σ, and the new interval $[i^{new}..j^{new}]$ in the backward index.

Here we present a new top-down traversal of the wavelet tree of BWT. Given a character c and an interval $[i..j]$, it is able to compute the three values $i' = Occ(c, i-1) + 1$, $j' = Occ(c, j)$, and *smaller*. Note that the new interval can then directly be determined by $[i^{new}..j^{new}] = [C[c] + i'..C[c] + j']$.

As an example, we compute the values of $Occ(\mathtt{l}, 5) + 1$, $Occ(\mathtt{l}, 11)$ and *smaller* for the interval [6..11] and the character $\mathtt{l}$. Because $\mathtt{l}$ belongs to the second half $\Sigma[5..7]$ of the ordered alphabet Σ, the occurrences of $\mathtt{l}$ correspond to ones in the bit vector at the root of the wavelet tree, and they go to the right child, say node v_1, of the root. In order to compute the number of occurrences of characters in the interval [6..11] that belong to $\Sigma[1..4]$ and hence are smaller than $\mathtt{l}$, we compute

$$(a_0, b_0) = \big(rank_0\big(B^{[1..7]}, 6-1\big), rank_0\big(B^{[1..7]}, 11\big)\big) = (2, 3)$$

and the number we are searching for is $b_0 - a_0 = 3 - 2 = 1$, so we add it to a variable *smaller*, which was initialized to 0 at the beginning. Then we descend to the right child v_1 and have to compute the boundaries of the search interval in the bit vector $B^{[5..7]}$ that corresponds to the search interval [6..11] in the bit vector $B^{[1..7]}$. These boundaries are $a_1 + 1$ and b_1, where

$$(a_1, b_1) = \big(rank_1\big(B^{[1..7]}, 6-1\big), rank_1\big(B^{[1..7]}, 11\big)\big) = (3, 8).$$

Proceeding recursively, we find that $\mathtt{l}$ belongs to the third quarter $\Sigma[5..6]$ of Σ, so the occurrences of $\mathtt{l}$ correspond to zeros in the bit vector at v_1, and they go to the left child, say node v_2, of v_1. Again, we compute

$$\big(a'_0, b'_0\big) = \big(rank_0\big(B^{[5..7]}, 4-1\big), rank_0\big(B^{[5..7]}, 8\big)\big) = (2, 7),$$
$$\big(a'_1, b'_1\big) = \big(rank_1\big(B^{[5..7]}, 4-1\big), rank_1\big(B^{[5..7]}, 8\big)\big) = (1, 1).$$

The number of occurrences of characters in the string $\mathsf{BWT}^{[3..8]}$ that belong to $\Sigma[7] = \mathtt{p}$ is $b'_1 - a'_1 = 1 - 1 = 0$ and the new search interval in the bit vector $B^{[5..6]}$ is $[a'_0 + 1..b'_0] = [3..7]$. In the third step, we compute

$$\big(a''_0, b''_0\big) = \big(rank_0\big(B^{[5..6]}, 3-1\big), rank_0\big(B^{[5..6]}, 7\big)\big) = (1, 4),$$
$$\big(a''_1, b''_1\big) = \big(rank_1\big(B^{[5..6]}, 3-1\big), rank_1\big(B^{[5..6]}, 7\big)\big) = (1, 3)$$

and find that there are $b''_1 - a''_1 = 2$ occurrences of the character $\mathtt{n}$ and $b''_0 - a''_0 = 3$ occurrences of the character $\mathtt{l}$. In the last step the search interval in the bit vector $B^{[5..5]}$ is $[a''_0 + 1..b''_0] = [2..4]$. We have reached a leaf of the wavelet tree, and $\mathsf{BWT}^{[5..5]} = \mathtt{llll}$. So obviously, $a''_0 + 1 = 2$ is exactly the value of $Occ(\mathtt{l}, 5) + 1$, and $b''_0 = 4$ is the value of $Occ(\mathtt{l}, 11)$. The variable $smaller = 1$ contains the sum of all found occurrences of characters in $\mathsf{BWT}[6..11]$ which are smaller than $\mathtt{l}$. Pseudo-code for the computation of the triple $(i', j', smaller) = traversal([i..j], c)$ where $i' = Occ(c, i-1) + 1$ and $j' = Occ(c, j)$ can be found in Algorithm 3.

The method of Lam et al. [7] is based on the same idea, but they use different data structures. They store the Burrows–Wheeler transformed string BWT, which requires $n \log \sigma$ bits, and a "rank and select data structure [22], which requires only $o(n)$ bits" (there are no details in [7]). We store the wavelet tree of BWT (but not the string BWT itself), which also requires $n \log \sigma$ bits, and the supporting rank data structures, which require $o(n \log \sigma)$ bits. It follows as a consequence that their

Algorithm 3 Given a BWT-interval $[i..j]$ and $c \in \Sigma$, $traversal([i..j], c)$ returns the triple $(i', j', smaller)$, where $[C[c] + i'..C[c] + j']$ is the new BWT-interval after a backward search step for c, and *smaller* is the number of occurrences of characters in BWT$[i..j]$ that are strictly smaller than c.

function $traversal([i..j], c)$
  return $traversal'([i..j], c, [1..\sigma], 0)$

function $traversal'([i..j], c, [l..r], smaller)$
  if $l = r$ **then return** $(i, j, smaller)$
  else
    $(a_0, b_0) \leftarrow (rank_0(B^{[l..r]}, i-1), rank_0(B^{[l..r]}, j))$
    $(a_1, b_1) \leftarrow (i - 1 - a_0, j - b_0)$
    $m = \lfloor \frac{l+r}{2} \rfloor$
    if $c \leqslant \Sigma[m]$ **then**
      return $traversal'([a_0 + 1..b_0], c, [l..m], smaller)$
    else
      return $traversal'([a_1 + 1..b_1], c, [m + 1..r], smaller + b_0 - a_0)$

Table 1
Comparison of the running times (in seconds) of the searches for the six RNA structural patterns in the human DNA sequence (about 3 billion nucleotides). The small numbers in parentheses are the numbers of matches found. Index ① is our new bidirectional wavelet index. Index ② consists of the suffix array SA of S (supporting binary search in the forward direction), and the wavelet tree of the Burrows–Wheeler transformed string of S (supporting backward search). Index ③ is similar to Index ②, but SA is replaced with a compressed suffix array. Index ④ is similar to index ③, but further enhanced by a balanced parenthesis sequence supporting faster forward search [14].

Index Size	① 2.2 GB	② 12.8 GB	③ 3.0 GB	④ 4.0 GB
`hairpin1` (2343)	13569 ms	14834 ms	285923 ms	105267 ms
`hairpin2` (286)	114 ms	71 ms	1420 ms	518 ms
`hairpin4` (3098)	884 ms	612 ms	12231 ms	4500 ms
`hloop(5)` (14870)	34297 ms	36993 ms	721301 ms	264904 ms
`acloop(5)` (294)	1163 ms	1284 ms	24950 ms	9228 ms
`acloop(10)` (224)	524 ms	576 ms	12468 ms	4754 ms

bidirectional BWT data structure uses less space than our bidirectional wavelet index. However, with the wavelet tree the number *smaller* can be computed in $O(\log\sigma)$ time (see above), so that one bidirectional search step takes $O(\log\sigma)$ time. By contrast, the method of Lam et al. [7] takes $O(\sigma)$ time for the same task. Let us recall the problem: given the ω-interval in the backward index and the ω^{rev}-interval in the forward index, the $c\omega$-interval is computed by backward search and the task is to find the $\omega^{rev}c$-interval in the forward index. For each character d that is smaller than c, Lam et al. determine the size of the $d\omega$-interval by backward search, and the number *smaller* is the sum of the sizes of these intervals. Because the calculation of the number *smaller* takes $O(\sigma)$ time, so does one search step with the bidirectional BWT.

4. Experimental results

An implementation of the bidirectional wavelet index is available at our website `http://www.uni-ulm.de/in/theo/research/seqana` under the GNU General Public License. To assess the performance of our new data structure, we used it to search for RNA secondary structures in large DNA sequences. We adopted the depth-first search method described in [4]; for space reasons, it is not repeated here. The following RNA secondary structures are also taken from [4]:

1. `hairpin1 = (stem:=N{20,50}) (loop:=NNN)^stem`
2. `hairpin2 = (stem:=N{10,50}) (loop:=GGAC)^stem`
3. `hairpin4 = (stem:=N{10,15}) (loop:=GGAC[1])^stem`
4. `hloop(length) = (stem:=N{15,20}) (loop:=N{length})^stem`
5. `acloop(length) = (stem:=N{15,20}) (loop:=(A|C){length})^stem`

The symbol `N` is used as a wildcard matching any nucleotide. The first pattern describes a hairpin structure with an apical loop consisting of three nucleotides. On the left- and right-hand sides of the loop are two reverse complementary sequences, each consisting of 20–50 nucleotides. The second pattern describes a similar structure, where the loop must be the sequence `GGAC`. The `[1]` in the third pattern means that one nucleotide can be inserted at any position, i.e., the loop is one of the sequences `GGAC`, `NGGAC`, `GNGAC`, `GGNAC`, `GGANC` or `GGACN`. In the last two patterns `length` denotes the length of the loop sequence. For example, in the pattern `acloop(5)` the loop consists of five nucleotides, each of which must either be `A` or `C`. In the experiments reported in Table 1, we searched for six patterns in the complete human genome,[1]

[1] `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/chromFa.tar.gz`.

which consists of about 3 billion nucleotides. All experiments were conducted on a PC with a *Dual-Core AMD Opteron 8218* processor (2,6 GHz) and 32 GB main memory. Unfortunately, the implementations of affix trees/arrays [3,4] are currently not available. For this reason, one cannot compare the running times. (We conjecture, however, that our method outperforms the affix array method.) Nevertheless, we can say something about the space consumption. According to Strothmann [4], an affix array requires 18 bytes per nucleotide, so approximately 50 GB for the human DNA sequence. The bidirectional wavelet index (index ①) takes only about 2.2 GB; see Table 1. Hence it decreases the space requirement by a factor of 23.

Due to the lack of affix tree/array implementations, we compared our method with the following three approaches to support bidirectional search. First, we combined the two well-known data structures supporting forward search (the suffix array SA of string S) and backward search (the wavelet tree of the Burrows–Wheeler transformed string BWT of S), and obtained an index (index ②) which also supports bidirectional search. Because both data structures deliver intervals of the suffix array, the two searches can directly be combined without synchronization. Interestingly enough, in the technical literature this natural approach has not been considered yet, i.e., it is new as well. Table 1 shows that index ② takes 12.8 GB for the human DNA sequence. Second, to reduce the space consumption even more, we replaced the suffix array in index ② by a compressed suffix array (CSA) which also supports binary search in the forward direction, yielding index ③. This reduces the memory consumption by another factor of 4, but slows down the running time by a factor of 20 (compared with index ②); see Table 1. This is because the CSA must frequently recover SA-values from its sampled SA-values (in our implementation every twelveth value is stored; more samples would decrease the running time, but increase the memory requirements). By enhancing this compressed suffix array with a balanced parenthesis sequence [14], faster forward search is possible, and we obtain index ④, which is about 2.7 times faster than index ③. By contrast, the time-space trade-off of our bidirectional wavelet index ① is much better: it reduces the space consumption by a factor of 5.8, while being about as fast as index ②.

We were not able to compare our implementation with the implementation of Lam et al. [7]. While our code is modular and for readability adheres closely to the presented algorithms, the code of the bidirectional BWT is integrated into a software tool for short read alignment. Its code is highly optimized for speed and some parts crucially depend on the DNA alphabet. This explains the reduced readability and modularity, and our inability to modify the code for an experimental comparison.

5. Bidirectional matching statistics

As already mentioned, matching statistics have been used, among other things, in approximate string matching [9], the computation of string kernels [10] and the design of DNA chips [11]. Let us recall the formal definition.

Definition 1. Let S^1 and S^2 be two strings of length $n_1 = |S^1|$ and $n_2 = |S^2|$. The *matching statistics* of S^2 w.r.t. S^1 is an array $ms[1..n_2]$ such that for each entry $ms[i] = \ell$ the string $S^2[i..i+\ell-1]$ is the longest prefix of $S^2[i..n_2]$ that is a substring of S^1.

Matching statistics can be computed in linear time by matching S^2 in forward direction against the suffix tree or the enhanced suffix array of S^1 [9,23]. Very recently, it was shown that they can also be computed in linear time by matching S^2 in backwards direction against a compressed full-text index (which consists e.g. of the backward index of S^1—as explained at the beginning of Section 3—and a compressed suffix tree that allows one to determine parent nodes in constant time); see [8].

In the context of bidirectional search, it is quite natural to raise the following problem. For any position i in S^2, find the length of the longest substring of S^2 containing position i that matches a substring somewhere in S^1. In other words, what is the maximal bidirectional extension $S^2[s..e]$ of $S^2[i]$, where $s \leqslant i \leqslant e$, such that $S^2[s..e]$ matches a substring somewhere in S^1 (provided, of course, that character $S^2[i]$ occurs in S^1 at all)?

Definition 2. The *bidirectional matching statistics* of S^2 w.r.t. S^1 is an array $bms[1..n_2]$ such that for each entry $bms[i] = (\ell, s)$ the string $S^2[s..s+\ell-1]$ is a longest substring of S^2 containing position i (i.e., $s \leqslant i \leqslant s+\ell-1$) that matches a substring somewhere in S^1.

Algorithm 4 contains pseudo-code that computes bidirectional matching statistics from matching statistics.

Before we analyze Algorithm 4, we illustrate it with an example. Let $S^1 = \texttt{gcgtcgc}$ and $S^2 = \texttt{atcgcg}$. The matching statistics of S^2 w.r.t. S^1 is $ms[1..6] = [0,4,3,3,2,1]$. In Table 2, one can see the values of i, $ms[i]$, *list* and $bms[i]$ at the end of each iteration of the for-loop in Algorithm 4. The result is $bms[1..6] = [(0,0),(4,2),(4,2),(4,2),(4,2),(3,4)]$.

The computation of the matchings statistics $ms[1..n_2]$ of string S^2 w.r.t. string S^1 takes $O(n_1+n_2)$ time [9,23,8]. Given $ms[1..n_2]$, Algorithm 4 computes the bidirectional matching statistics $bms[1..n_2]$ in $O(n_2)$ steps. This is because in each of the n_2 iterations of its for-loop at most one element is added to the list (line 8), and thus the while-loop also makes at most n_2 iterations. Because all other statements take constant time, this results in an amortized runtime of $O(n_2)$. Therefore, the overall time complexity of computing the bidirectional matching statistics is $O(n_1+n_2)$.

To prove the correctness of Algorithm 4, we show that it maintains the three invariants of the following lemma.

Algorithm 4 Given the matching statistics $ms[1..n_2]$ of S^2 w.r.t. S^1, this algorithm computes the bidirectional matching statistics. During a left-to-right scan over the array $ms[1..n_2]$, it maintains longest matches in a linked list having elements of the form $(length, start)$.

```
function bidirectionalMatchingStatistics(ms[1..n2])
1  list ← [ ]
2  for i = 1 to n2 do
3    if list.size > 0 and list.getFirst.start + list.getFirst.length ⩽ i then
4      list.removeFirst
5    if ms[i] > 0 and (i = 1 or ms[i] ≠ ms[i − 1] − 1) then
6      while (list.size > 0 and ms[i] ⩾ list.getLast.length)
7        list.removeLast
8      list.addLast((ms[i], i))
9    if list.size = 0 then
10     bms[i] ← (0, 0)
11   else
12     bms[i] ← list.getFirst
13 return bms
```

Table 2
The values of i, $ms[i]$, $list$ and $bms[i]$ at the end of each iteration of the for-loop.

i	$ms[i]$	$list$	$bms[i]$
1	0	[ ]	(0, 0)
2	4	[(4, 2)]	(4, 2)
3	3	[(4, 2)]	(4, 2)
4	3	[(4, 2), (3, 4)]	(4, 2)
5	2	[(4, 2), (3, 4)]	(4, 2)
6	1	[(3, 4)]	(3, 4)

Lemma 1. *If the list before the i-th iteration of the for-loop has the form $[(ms[i_1], i_1), (ms[i_2], i_2), \ldots, (ms[i_m], i_m)]$ (where $m \geqslant 0$ and $i_1 < i_2 < \cdots i_m$), then the following three properties hold true.*

1. *The sequence $ms[i_1], \ldots, ms[i_m]$ is strictly descending.*
2. *The sequence $i_1 + ms[i_1], \ldots, i_m + ms[i_m]$ is strictly ascending.*
3. *If the list is non-empty (i.e., $m > 0$), then $bms[i-1] = (ms[i_1], i_1)$. Otherwise $i = 1$ or $bms[i-1] = (0, 0)$.*

Proof. The initial list is empty, so the three properties hold before the first iteration of the for-loop. Suppose that the properties are true before the i-th iteration of the for-loop. We show that they also hold after that iteration (i.e., before the $(i+1)$-th iteration).

1. Before $(ms[i], i)$ is inserted at the end of the list, all elements with first component $length \leqslant ms[i]$ are removed from the end of the list. Hence the first property also holds after the i-th iteration of the for-loop (of course, a possible deletion of the first element does not harm).
2. Consider the sequence $a_j = j + ms[j]$, where $1 \leqslant j \leqslant n_2$. We claim that it is ascending. For a proof by contradiction suppose that $a_j = j + ms[j] < j - 1 + ms[j-1] = a_{j-1}$ for some $j > 1$. Clearly, this is equivalent to $ms[j] < ms[j-1] - 1$. This inequality, however, contradicts the definition of the matching statistics. Therefore, the sequence $a_1, \ldots a_{n_2}$ is ascending. By assumption, the sequence $i_1 + ms[i_1], \ldots, i_m + ms[i_m]$ before the i-th iteration of the for-loop is strictly ascending. Even if the first element of the list and/or elements at the end of the list are removed, the resulting list $[(ms[j_1], j_1), (ms[j_2], j_2), \ldots, (ms[j_k], j_k)]$ still has the second property. Clearly, $(ms[i], i)$ is inserted at the end of the resulting list if and only if $ms[i] > ms[i-1] - 1$, or equivalently, $i + ms[i] > i - 1 + ms[i-1]$. If it is not inserted, then the second property holds as discussed above. If it is inserted, then the new list after the i-th iteration has the form $[(ms[j_1], j_1), (ms[j_2], j_2), \ldots, (ms[j_k], j_k), (ms[i], i)]$. Because the sequence $a_1, \ldots a_{n_2}$ is ascending, we conclude that $j_k + ms[j_k] \leqslant i - 1 + ms[i-1]$. In conjunction with $i - 1 + ms[i-1] < i + ms[i]$, this implies that the list after the i-th iteration has the second property.
3. Suppose first that the list before the i-th iteration is empty. Then we have $bms[i-1] = (0, 0)$, that is, the character $S^2[i-1]$ does not occur in S^1. If $ms[i] > 0$, then the list after the i-th iteration contains only the element $(ms[i], i)$ and clearly $bms[i] = (ms[i], i)$. Otherwise $ms[i] = 0$, so the list after the i-th iteration is also empty and $bms[i] = (0, 0)$. In both cases, the third property holds.
   Now suppose that the list before the i-th iteration is not empty, i.e., $list = [(ms[i_1], i_1), (ms[i_2], i_2), \ldots, (ms[i_m], i_m)]$, where $m > 0$. By assumption $bms[i-1]$ is the first element $(ms[i_1], i_1)$ of the list. If this element is *not deleted* from the list (lines 3–4), then $S^2[i_1..i_1 + ms[i_1] - 1]$ still contains position i and it follows from the properties that

$$bms[i] = \begin{cases} (ms[i], i) & \text{if } ms[i] \geqslant ms[i_1], \\ (ms[i_1], i_1) & \text{otherwise.} \end{cases} \quad (1)$$

If the first element is *deleted* from the list (lines 3–4), then this is because $S^2[i_1..i_1 + ms[i_1] - 1]$ does not contain position i ($i_1 + ms[i_1] - 1 < i$). On the other hand, since $S^2[i_1..i_1 + ms[i_1] - 1]$ contains position $i - 1$ ($i_1 + ms[i_1] - 1 \geqslant i - 1$), it follows that $i_1 + ms[i_1] - 1 = i - 1$. Let us assume that the original list has a second element $(ms[i_2], i_2)$ (the other case follows by a similar reasoning as above). Then property (2) implies $i_2 + ms[i_2] - 1 > i - 1$, that is, $S^2[i_2..i_2 + ms[i_2] - 1]$ contains position i (consequently, $ms[i] > 0$). Again, it follows from the properties that

$$bms[i] = \begin{cases} (ms[i], i) & \text{if } ms[i] \geqslant ms[i_2], \\ (ms[i_2], i_2) & \text{otherwise.} \end{cases} \quad (2)$$

Now there are two possibilities: Either $ms[i] \geqslant ms[i_1]$ or $ms[i] < ms[i_1]$ in Eq. (1) (either $ms[i] \geqslant ms[i_2]$ or $ms[i] < ms[i_2]$ in Eq. 2, respectively). In the first case, all elements of the list are removed (in lines 6–7) and then $(ms[i], i)$ is inserted into the now empty list. Hence $bms[i] = (ms[i], i)$ is the first element of the new list. In the second case, some elements are removed from the end of the list, but not the first one. Thus $bms[i] = (ms[i_1], i_1)$ ($bms[i] = (ms[i_2], i_2)$, respectively) is the first element of the new list. □

References

[1] R.C. Lee, R.L. Feinbaum, V. Ambros, The C. elegans heterochronic gene lin-4 encodes small RNAs with antisense complementarity to lin-14, Cell 75 (1993) 843–854.
[2] V. Kim, J. Nam, Genomics of microRNA, Trends Genet. 22 (2006) 165–173.
[3] G. Mauri, G. Pavesi, Pattern discovery in RNA secondary structure using affix trees, in: CPM'03: Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, Springer-Verlag, 2003, pp. 278–294.
[4] D. Strothmann, The affix array data structure and its applications to RNA secondary structure analysis, Theor. Comput. Sci. 389 (2007) 278–294.
[5] J. Stoye, Affix trees, Technical Report 04, University of Bielefeld, 2000.
[6] M.G. Maaß, Linear bidirectional on-line construction of affix trees, Algorithmica 37 (2003) 43–74.
[7] T.W. Lam, R. Li, A. Tam, S. Wong, E. Wu, S.M. Yiu, High throughput short read alignment via bi-directional BWT, in: BIBM'09: Proceedings of the 2009 IEEE International Conference on Bioinformatics and Biomedicine, IEEE Computer Society, Washington, DC, USA, 2009, pp. 31–36.
[8] E. Ohlebusch, S. Gog, A. Kügel, Computing matching statistics and maximal exact matches on compressed full-text indexes, in: SPIRE'10: Proceedings of the 17th International Symposium on String Processing and Information Retrieval, in: LNCS, vol. 6393, Springer-Verlag, 2010, pp. 347–358.
[9] W. Chang, E. Lawler, Sublinear approximate string matching and biological applications, Algorithmica 12 (1994) 327–344.
[10] C.H. Teo, S.V.N. Vishwanathan, Fast and space efficient string kernels using suffix arrays, in: ICML'06: Proceedings of the 23rd International Conference on Machine Learning, ACM, New York, NY, USA, 2006, pp. 929–936.
[11] S. Rahmann, Fast and sensitive probe selection for DNA chips using jumps in matching statistics, in: CSB'03: Proceedings of the IEEE Computer Society Conference on Bioinformatics, IEEE Computer Society, Washington, DC, USA, 2003, pp. 57–64.
[12] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (1993) 935–948.
[13] S.J. Puglisi, W.F. Smyth, A.H. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2007) 1–31.
[14] E. Ohlebusch, S. Gog, A compressed enhanced suffix array supporting fast string matching, in: SPIRE'09: Proceedings of the 16th International Symposium on String Processing and Information Retrieval, in: LNCS, vol. 5721, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 51–62.
[15] M. Burrows, D.J. Wheeler, A block-sorting lossless data compression algorithm, Research Report 124, Digital Systems Research Center, 1994.
[16] J. Seward, bzip2 v1.0.5, http://www.bzip.org/, 2007.
[17] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching, Springer-Verlag, 2008.
[18] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: FOCS'00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, USA, 2000, pp. 390–398.
[19] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (2007), Article 2.
[20] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: SODA'03: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003, pp. 841–850.
[21] G. Jacobson, Space-efficient static trees and graphs, in: FOCS'89: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, USA, 1989, pp. 549–554.
[22] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theor. Comput. Sci. 387 (2007) 332–347.
[23] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discrete Algorithms 2 (2004) 53–86.