Bidirectional Search in a String with Wavelet Trees

Thomas Schnattinger, Enno Ohlebusch, and Simon Gog

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm
{Thomas.Schnattinger,Enno.Ohlebusch,Simon.Gog}@uni-ulm.de

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 RNA secondary structural patterns in large genomes, for example the human genome.

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 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

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 40–50, 2010.

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 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 paper, 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 21 (compared to affix arrays), making it possible to search bidirectionally in very large strings.

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., $\Sigma[1] = \$ < \Sigma[2] < \ldots < \Sigma[\sigma]$. In the following, S is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i..j]$ 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 $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \ldots < S_{\mathsf{SA}[n]}$; see Fig. 1 for an example. In the following, SA^{-1} denotes the inverse of the permutation SA. The suffix array was introduced by

i	SA	BWT	$S_{\mathsf{SA}[i]}$
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$

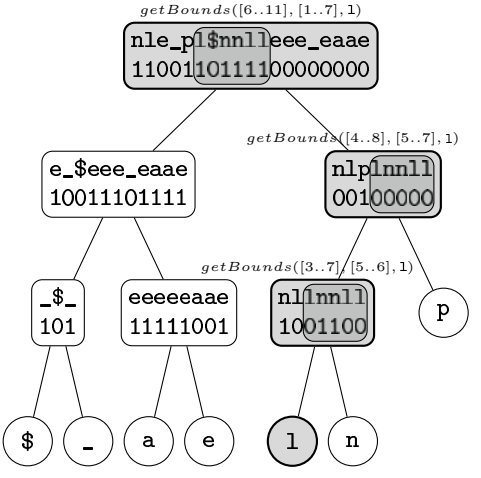

Fig. 1. Left: Suffix array and Burrows-Wheeler-transformed string BWT of string $S =$ `el_anele_lepanelen$`. Right: Conceptual illustration of the wavelet tree of the string BWT = `nle_pl$nnlleee_eaae`. Only the bit vectors are stored; the corresponding strings are shown for clarity. The shaded regions and the function *getBounds* will be explained later.

Manber and Myers [7]. In 2003, it was shown independently and contemporaneously by three research groups that a direct linear time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms are known; see [8] for details. Forward search on a suffix array can be done in $O(\log n)$ time per character by binary search; see [7].

Given the suffix array SA of a string S, the Burrows and Wheeler transformation $\mathsf{BWT}[1..n]$ of S is defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 1. In virtually all cases, the Burrows-Wheeler transformed string compresses much better than the original string; see [9]. The permutation LF, defined by $LF(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $LF(i) = 1$ otherwise, is called LF-mapping. Its inverse permutation is usually called ψ-function. Both LF and ψ can be represented more compactly than the suffix array. A compressed full-text index based on a compressed form of the LF-mapping is commonly referred to as FM-index [10]. If it is based on a compressed ψ-function it is usually called *compressed suffix array* [11]. The LF-mapping can be implemented 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 $\mathsf{BWT}[1..i]$. Details about the Burrows and Wheeler transform and related topics can for instance be found in [12].

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 $i \leq j$ **then return** $[i..j]$
  else return $\perp$

Ferragina and Manzini [10] showed that it is possible to search a pattern character-by-character backwards in the suffix array SA of string S, without storing SA. Backward search can be implemented such that each step takes only constant time, albeit a more space-efficient implementation takes $\mathcal{O}(\log \sigma)$ time; see below. 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 $i \leq k \leq j$, but ω is not a prefix of any other suffix of S. 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 $C[\mathtt{l}] + Occ(\mathtt{l}, 11) = 11 + 4 = 15$.

With the space-efficient *wavelet tree* introduced by Grossi et al. [13], 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 subalphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. As an example, consider the string $\mathsf{BWT} =$ `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 $B^{[l..r]}$ of size $r - l + 1$ whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the subalphabet $\Sigma[l..m]$ and 1 if it belongs to the subalphabet $\Sigma[m+1..r]$. 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

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])
```

belongs to the right subtree; see Fig. 1. 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 [14], 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 $\mathsf{BWT} = \mathtt{nle\_pl\$nnlleee\_eaae}$ from Fig. 1. 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 $\mathsf{BWT}^{[1..7]} = \mathtt{nle\_pl\$nnlleee\_eaae}$ up to position 16 equals the number of `e`'s in the string $\mathsf{BWT}^{[1..4]} = \mathtt{e\_\$eee\_eaae}$ 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 $\mathsf{BWT}^{[1..4]} = \mathtt{e\_\$eee\_eaae}$ up to position 8 is equal to the number of `e`'s in $\mathsf{BWT}^{[3..4]} = \mathtt{eeeeeaae}$ 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 $\mathsf{BWT}^{[3..4]} = \mathtt{eeeeeaae}$ up to position 5 equals the number of `e`'s in $\mathsf{BWT}^{[4..4]} = \mathtt{eeeeee}$ 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

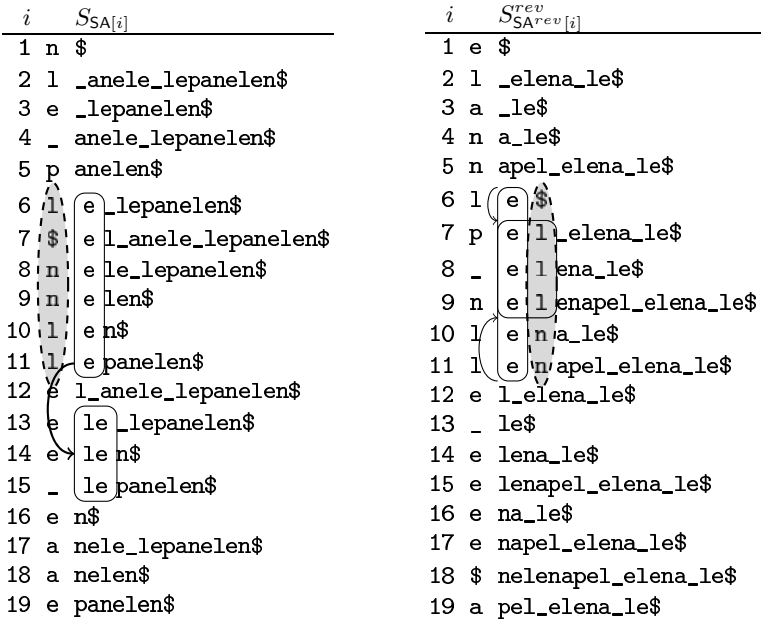

Fig. 2. Bidirectional wavelet index of $S =$ `el_anele_lepanelen$`

– 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 $S =$ `el_anele_lepanelen$` in Fig. 2, and the substring $\omega =$ `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 $i^{rev} \leq k < p$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] = c$ for all k with $p \leq k \leq q$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] > c$ for all k with $q < k \leq j^{rev}$.

In the example of Fig. 2,

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

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 and the number *greater* of all occurrences of characters at these positions that follow c in the alphabet, then we could identify the unknown $\omega^{rev}c$-interval $[p..q]$ by $p = i^{rev} + smaller$ and $q = j^{rev} - greater$. In our example, the knowledge of $smaller = 1$ and $greater = 2$ would yield the $\texttt{el}$-interval $[6 + 1..11 - 2] = [7..9]$. The *key observation* is that the multiset of characters

$$\{S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] : i^{rev} \leq k \leq j^{rev}\}$$

coincides with the multiset $\{\mathsf{BWT}[k] : i \leq k \leq j\}$. In the example of Fig. 2,

$$\{S^{rev}[\mathsf{SA}^{rev}[k]+1] : 6 \leq k \leq 11\} = \{\ \texttt{\$}, \texttt{l}, \texttt{l}, \texttt{l}, \texttt{n}, \texttt{n}\} = \{\mathsf{BWT}[k] : 6 \leq k \leq 11\}$$

In other words, it suffices to determine the numbers *smaller* and *greater* of all occurrences of characters in the string $\mathsf{BWT}[i..j]$ that precede and follow character c in the alphabet Σ. And this task can be accomplished by a top-down traversal of the wavelet tree of BWT. The procedure is similar to the implementation of $Occ(c, i)$ as explained above. As an example, we compute the values of *smaller* and *greater* for the interval [6..11] and the character $\texttt{l}$. This example is illustrated in Fig. 1. Because $\texttt{l}$ belongs to the second half $\Sigma[5..7]$ of the ordered alphabet Σ, the occurrences of $\texttt{l}$ correspond to ones in the bit vector at the root, 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 in the left child of the root, we compute

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

and the number we are searching for is $b_0 - a_0 = 3 - 2 = 1$. 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

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

Proceeding recursively, we find that $\texttt{l}$ belongs to the third quarter $\Sigma[5..6]$ of Σ, so the occurrences of $\texttt{l}$ 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

Algorithm 3. Given a BWT-interval $[i..j]$, an alphabet-interval $[l..r]$, and $c \in \Sigma$, $getBounds([i..j],[l..r],c)$ returns the pair $(smaller, greater)$, where $smaller$ ($greater$) is the number of all occurrences of characters from the subalphabet $\Sigma[l..r]$ in $\mathsf{BWT}[i..j]$ that are smaller (greater) than c.

```
getBounds([i..j],[l..r],c)
  if l = r then return (0,0)
  else
    (a0,b0) ← (rank0(B^[l..r],i−1), rank0(B^[l..r],j))
    (a1,b1) ← (i−1−a0, j−b0)
    /* (a1,b1) = (rank1(B^[l..r],i−1), rank1(B^[l..r],j)) */
    m = ⌊(l+r)/2⌋
    if c ≤ Σ[m] then
      (smaller,greater) ← getBounds([a0+1..b0],[l..m],c)
      return (smaller, greater+b1−a1)
    else
      (smaller,greater) ← getBounds([a1+1..b1],[m+1..r],c)
      return (smaller+b0−a0, greater)
```

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

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

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

and find that there are $b''_1 - a''_1 = 2$ occurrences of the character `n` and $b''_0 - a''_0 = 3$ occurrences of the character `l`. In summary, during the top-down traversal, we found that in the string $\mathsf{BWT}[6..11]$ there is one character smaller than `l` (so $smaller = 1$), there are two characters greater than `l` (so $greater = 2$), and three characters coincide with `l`. Pseudo-code for the computation of $(smaller, greater) = getBounds([i..j],[1..\sigma],c)$ can be found in Algorithm 3.

4 Experimental Results

An implementation of the bidirectional wavelet index is available under the GNU General Public License at `http://www.uni-ulm.de/in/theo/research/seqana`. 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]:

Table 1. Comparison of the running times (in seconds) of the searches for the six RNA structural patterns in the human DNA sequence (about one billion nucleotides). The numbers in parentheses below the pattern names 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	MB	`hairpin1` (2343)	`hairpin2` (286)	`hairpin4` (3098)	`hloop(5)` (14870)	`acloop(5)` (294)	`acloop(10)` (224)
①	799	11.053	0.079	0.792	28.373	0.958	0.420
②	4408	8.855	0.041	0.365	22.208	0.781	0.336
③	1053	137.371	0.651	5.642	345.860	12.174	6.381

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 first five chromosomes of the human genome.[1] The concatenation of the DNA sequences of these five chromosomes is called "human DNA sequence" in the following; it constitutes about one third of the whole genome (one billion nucleotides). All experiments were conducted on a PC with a *Dual-Core AMD Opteron 8218* processor (2,6 GHz) and 8 GB main memory. Unfortunately, the implementations of affix trees/arrays [3,4] are currently not available.[2] 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 16.8 GB for the human DNA sequence. The bidirectional wavelet index (index ①) takes only 799 MB; see Table 1. Hence it decreases the space requirement by a factor of 21.

[1] `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/chromFa.tar.gz`
[2] However, a reimplementation of the affix array method is under way.

Due to the lack of affix tree/array implementations, we compared our method with the following two 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 4.4 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 15.5 (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 contrast, the time-space trade-off of our bidirectional wavelet index ① is much better: it reduces the space consumption by a factor of 5.5, but it is only 1.2 - 2.2 time slower than index ②. This can be attributed to the fact that SA-values are solely needed to output the positions of the matching regions in the string S (in our implementation a hundredth of all SA-values is stored).

References

1. Lee, R., Feinbaum, R., Ambros, V.: The C. elegans heterochronic gene lin-4 encodes small RNAs with antisense complementarity to lin-14. Cell 75(5), 843–854 (1993)
2. Kim, N., Nam, J.W.: Genomics of microRNA. TRENDS in Genetics 22(3), 165–173 (2006)
3. Mauri, G., Pavesi, G.: Pattern discovery in RNA secondary structure using affix trees. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 278–294. Springer, Heidelberg (2003)
4. Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theoretical Computer Science 389, 278–294 (2007)
5. Stoye, J.: Affix trees. Technical report 2000-04, University of Bielefeld (2000)
6. Maaß, M.: Linear bidirectional on-line construction of affix trees. Algorithmica 37, 43–74 (2003)
7. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
8. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
9. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
10. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
11. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. ACM Symposium on the Theory of Computing, pp. 397–406. ACM Press, New York (2000)

12. Manzini, G.: The Burrows-Wheeler Transform: Theory and practice. In: Kutyłowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 34–47. Springer, Heidelberg (1999)
13. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
14. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th Annual Symposium on Foundations of Computer Science, pp. 549–554. IEEE, Los Alamitos (1989)