Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Theoretical Computer Science 389 (2007) 278–294

Theoretical Computer Science

www.elsevier.com/locate/tcs

The affix array data structure and its applications to RNA secondary structure analysis

Dirk Strothmann*

Technische Fakultät, Prakt. Informatik, University of Bielefeld, Universitätsstraße 25, 33615 Bielefeld, Germany

Received 30 March 2006; received in revised form 25 July 2007; accepted 30 September 2007

Communicated by A. Apostolico

Abstract

Efficient string-processing in large data sets like complete genomes is strongly connected to the suffix tree and similar index data structures. With respect to complex structural string analysis like the search for RNA secondary structure patterns, unidirectional suffix tree algorithms are inferior to bidirectional algorithms based on the affix tree data structure. The affix tree incorporates the suffix tree and the suffix tree of the reverse text in one tree structure but suffers from its large memory requirements. In this paper I present a new data structure, denoted *affix array*, which is equivalent to the affix tree with respect to its algorithmic functionality, but with smaller memory requirements and improved performance. I will show a linear time construction of the affix array without making use of the linear time construction of the affix tree. I will also show how bidirectional affix tree traversals can be transferred to the affix array and present the impressive results of large scale RNA secondary structure analysis based on the new data structure.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Suffix tree; Suffix array; Affix tree; Pattern matching

1. Introduction

Index data structures are essential tools for string processing in large data sets. The search for textual patterns in complete genomes which contain millions of nucleotides is still one of the fundamental tasks of bioinformatics and can be solved efficiently based on the most important elements constituting the suffix tree and the suffix array data structures. In addition to the search of genomic sequences in large genomic data sets like the genome of an organism or a genome database, the detection of complex structural elements becomes more and more important in the field of modern molecular biology [1,2,16,17]. For example, RNA secondary structure patterns containing hairpin loops are of primary interest for the field of microRNA detection whereas the concrete sequence information might only be given partially. The efficient search for RNA secondary structure patterns requires bidirectional search algorithms which are not applicable to unidirectional data structures like the suffix tree and the more space efficient suffix array. Both can be constructed in linear time and space for a constant-size alphabet and allow us to locate a sequence of length

* Tel.: +49 5425 930547; fax: +49 5425 930547.
E-mail address: dirks@techfak.uni-bielefeld.de.

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.09.029

m in $\mathcal{O}(m)$ time. The space consumption of the suffix tree is 20 bytes per text symbol in the worst case and clearly exceeds the space consumption of the suffix array which, depending on the implementation, consumes 4–6 bytes per text symbol. In order to analyze a text S in reverse orientation we may consider the reverse prefix tree respectively the reverse prefix array constituting the suffix tree respectively suffix array of the reverse text S^{-1}.

The affix tree introduced by Stoye [3] and Maass [4] combines suffix tree and reverse prefix tree in one data structure and allows for bidirectional applications which are essential for the search for structural biological motifs as shown by Mauri et al. [5,15]. The affix tree can be constructed in linear time and space, but suffers from its large memory requirements (approximately 45 bytes per text symbol) and its complex structure which decreases the algorithmic performance.

In Section 3 I will present a new data structure called *affix array* which is equivalent to the affix tree with respect to its functionality (bidirectional search applications), but with smaller memory requirements and improved performance comparable to the difference between suffix tree and suffix array. The *affix array* is an extension of the suffix array and allows for efficient bidirectional string search. It transfers the idea of the affix tree (which is an extension of the suffix tree) to an array structure. The affix array can be constructed in linear time and space and consumes 18–20 bytes per text symbol. Because of its simple, memory efficient array structure and the reduced memory requirements, the affix array allows for efficient large scale applications as will be demonstrated in Section 4.

2. Basic notions

2.1. Alphabets and strings

In the following let Σ denote a finite set of characters, the underlying alphabet. A string over Σ is a sequence of characters from Σ. Σ^m is the set of all strings of length m over Σ, denoted m-strings. $|S|$ denotes the length of the string S. For any partitioning of $S = uv$ into possibly empty u and v we call u a prefix and v a suffix of S. The k-th suffix is the suffix starting at position k while the k-th prefix ends at position k. Let $S_i \in \Sigma$ denote the i-th character of a string S of length n over Σ. For substrings of S we use the following notation:

- $(i : S : j)$ denotes the substring: $S_i, S_{i+1} \dots S_j$
- $(i : S)$ denotes the i-th suffix of S: $S_i, S_{i+1} \dots S_{n-1}$
- $(S : j)$ denotes the j-th prefix of S: $S_0, S_1 \dots S_j$.

If a string u is prefix respectively suffix of string S we write $S = u-$ respectively $S = -u$. We call a string u a *proper* prefix (suffix) of string S if u is prefix (suffix) of S and $u \neq S$ and write $S = u\dashv$ $(S = \vdash u)$.

2.2. Suffix tree and suffix array

The suffix tree T of a text S is a rooted directed tree with $|S|$ leaves. Each edge of the *atomic* suffix tree (see Fig. 1) is labeled with a character of S and every substring of S is represented by a path from the root to a node q denoted *suffix path* of S by concatenation of the corresponding edge labels. Every suffix of S is represented by a suffix path from the root to a leaf of T. The *prefix path* of a node q leads from q to the root and represents the reverse of the suffix path of q. In contrast to the atomic suffix tree the *compact* suffix tree (see Fig. 1) can be constructed in linear time and space and concentrates non-branching edges of the atomic suffix tree in one edge labeled with the concatenation of the single character edge labels.

Definition 2.1. We define the forward string $\overrightarrow{q}$ of a node q as the concatenation of edge labels on the suffix path of q and the backward string $\overleftarrow{q} = (\overrightarrow{q})^{-1}$ as concatenation of the edge labels on the prefix path of q .

Each node q of the atomic suffix tree either corresponds to an *explicit* node in the compact suffix tree or an *implicit* node which is represented by an explicit node e supplemented by a substring s of an outgoing edge label of e such that $\overrightarrow{q}$ is the concatenation of $\overrightarrow{e}$ and s.

Definition 2.2. For any substring s of S let $\overline{s}$ denote the explicit or implicit node q representing s ($\Rightarrow \overrightarrow{q} = s$ and $\overleftarrow{q} = s^{-1}$).

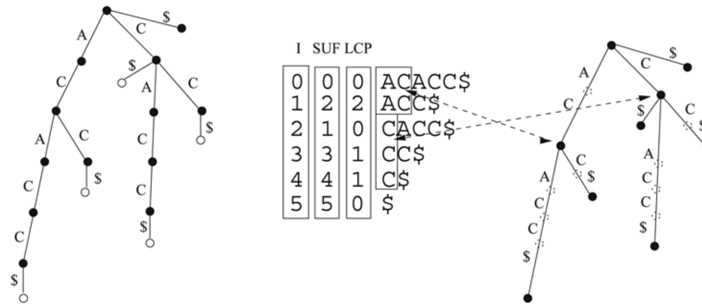

Fig. 1. Atomic (left) and compact (right) suffix tree for the string ACACC. The end character $\$$ is appended to mark the end-positions of suffixes. In the middle the corresponding suffix array is shown with lcp-intervals representing explicit branching nodes in the compact suffix tree marked by boxes. Implicit nodes in the compact suffix tree are indicated by dashed circles.

The suffix array [6] is the lexicographically ordered list of all suffixes of the text S with the addition of an end character \$ which is considered as the lexicographically last symbol of the alphabet. With Suf_k we denote the k-th suffix with respect to the lexicographical order. Instead of storing all suffixes only the start positions of the suffixes with respect to the lexicographical order are stored in a table suftab of size $4n$ bytes where each position is stored in 4 bytes and n is the length of S. For any $k \in [0..n-1]$, $\mathsf{suftab}[k] = i$ means: Suf_k starts at text position i. For texts longer than 2^{32} characters respectively 4 GB (we can store integers between 0 and $2^{32}-1$ in 4 bytes) we have to use more than 4 bytes per position. From an algorithmic point of view the suffix array data structure is equivalent to the suffix tree as it allows for equivalent operations if intervals in the suffix array are regarded as nodes in the tree. Since child nodes can be reached in constant time on the suffix tree but in logarithmical time on the suffix array in its raw form, the time complexity of m-string localization is $\mathcal{O}(m)$ on the suffix tree compared to $\mathcal{O}(m \log n)$ on the classical suffix array. Nevertheless in most cases suffix array algorithms perform better due to the simple array structure which can be handled more efficiently by caching processes than the complex tree structure.

2.3. The enhanced suffix array

Algorithms on the suffix array can be accelerated by two additional tables lcptab (longest common prefix) and $\mathsf{childtab}$ each of size n bytes [7]. Abouelhoda et al. [8] showed that this so-called *enhanced* suffix array data structure allows for algorithms of the same time complexity as corresponding suffix tree implementations. In respect to string search, this means that we can locate m-strings in time $\mathcal{O}(m)$ on the enhanced suffix array.

Definition 2.3.

$$\mathsf{lcptab}[i] = \begin{cases} 0, \text{ if } i = 0 \text{ or } i = n \\ \max\{j \mid (0 : \mathsf{Suf}_{i-1} : j) = (0 : \mathsf{Suf}_i : j)\} + 1, \text{otherwise} \end{cases}$$

where n is the length of the underlying text.

In other words, $\mathsf{lcptab}[i]$ contains the length of the longest common prefix of Suf_i and Suf_{i-1}. In most cases $\mathsf{lcptab}[i]$ is smaller than 255 and can be stored in one byte. If $\mathsf{lcptab}[i]$ is bigger than 254, the value is stored in an exception table. In Fig. 1 the lcp-values are shown in the table with caption LCP.

Algorithms on the enhanced suffix array are based on *lcp-intervals* which are uniquely determined by their left and right border.

Definition 2.4. An interval $[i..j]$, $0 \le i < j \le n$, is called an lcp-interval of lcp-value $l > 0$ (or l-interval) if

- $\mathsf{lcptab}[i] < l$
- $\mathsf{lcptab}[k] \ge l, \forall k \in [i+1..j]$
- $\mathsf{lcptab}[k] = l$ for at least one $k \in [i+1..j]$
- $\mathsf{lcptab}[j+1] < l$.

The root interval $[0..n]$ is denoted lcp-interval of lcp-value 0 and represents the empty string. Every index $k \in 1 - [i+1..j]$ with $\mathsf{lcptab}[k] = l$ is called l-index. An m-interval $[r..s]$ is *embedded* in an l-interval $[i..j]$ if it is

a subinterval of $[i..j]$ and $m > l$. $[i..j]$ is called the interval *enclosing* $[r..s]$. If $[i..j]$ encloses $[r..s]$ but no interval that also encloses $[r..s]$, then $[r..s]$ is called a *child interval* of $[i..j]$. The additional table childtab of size n allows us to locate a child interval of an l'-interval in $|\Sigma|$ steps. childtab consists of 3 components *up*, *down* and *nextℓIndex* each requiring 4 bytes in the worst case.

Definition 2.5.

$$\mathsf{childtab}[i].up = \min\{q \in [0..i-1] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[i] \text{ and } \forall k \in [q+1..i-1] : \mathsf{lcptab}[k] \geq \mathsf{lcptab}[q]\}$$

$$\mathsf{childtab}[i].down = \max\{q \in [i+1..n] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[i] \text{ and } \forall k \in [i+1..q-1] : \mathsf{lcptab}[k] > \mathsf{lcptab}[q]\}$$

$$\mathsf{childtab}[i].next\ell Index = \min\{q \in [i+1..n] \mid \mathsf{lcptab}[q] = \mathsf{lcptab}[i] \text{ and } \forall k \in [i+1..q-1] : \mathsf{lcptab}[k] > \mathsf{lcptab}[i]\}.$$

The space requirements for childtab can be slashed since all three components can be stored in one table with 1 byte per field as shown in [8]. Given an l-interval $[i..j]$ we can find the first l-index k in $[i..j]$ in constant time and in $\mathsf{lcptab}[k]$ we find the lcp-value l of the interval $[i..j]$. We can locate the child intervals of $[i..j]$ in constant time using the additional childtab information which is the basic property for the location of an m-string in $\mathcal{O}(m)$ time.

2.4. Virtual nodes

A node q in the suffix tree represents a set of substrings of the underlying text by its suffix path equivalent to an interval $I(\vec{q}) = [q_1, q_2]$ of suffix start positions in the suftab table where the corresponding suffixes start with the prefix $\vec{q}$. Thus a bijection between the nodes of the suffix tree and the corresponding suffix array intervals exists which is reflected by the following definitions:

Definition 2.6. Given a node q of the suffix tree different from the root. Let i be the smallest and j be the largest suftab index in the corresponding suffix array of a suffix with prefix $\vec{q}$. The pair $v = ([i..j], |\vec{q}|)$ is called *virtual node* of $\vec{q}$. If only one leaf below v exists, v is denoted *singular*, while if there are more leaves below v, v is denoted *multiple* node. v is called *explicit* virtual node if $[i..j]$ is an lcp-interval, denoted as lcp-interval of v, otherwise v is called *implicit* virtual node. Notice that leaves are considered as implicit virtual nodes since no lcp-interval represents a leaf.

If $v = ([i..j], d)$ is a multiple virtual node and $[i..j]$ is an l-interval then obviously $d \leq l$. v is uniquely determined by i and d since the right bound j of the interval $[i..j]$ is implicitly given by i and d: $j = \min\{r \mid i < r < n, \mathsf{lcptab}[r+1] < d\}$. Consequently we can omit the right bound of the interval $[i..j]$ and use the reduced form (i, d) to specify the virtual node v. The virtual nodes are connected by edges that do not exist explicitly, but are implied by the structure of the suffix array. In the following we identify the virtual nodes of the suffix array with the nodes of the suffix tree.

Lemma 2.1. *Let $q = (i, d)$ represent a virtual node in the suffix array. Then q is multiple, if and only if $\mathsf{lcptab}[i+1] \geq d$, otherwise q is singular.*

Proof. If $\mathsf{lcptab}[i+1] \geq d$ Suf_i and Suf_{i+1} are identical in the first d positions. Thus we find at least two leaves below the virtual node $q = (i, d)$ and therefore q is multiple. If $\mathsf{lcptab}[i+1] < d$ there is no further suffix identical to Suf_i in the first d positions, thus q has only one child. The opposite direction follows equivalently. □

2.5. The affix tree data structure

The affix tree is the incorporation of suffix tree and reverse prefix tree, according to the property that any node of the suffix tree can be regarded as a node of the reverse prefix tree and vice versa (see Fig. 2). The *suffix-descendants*

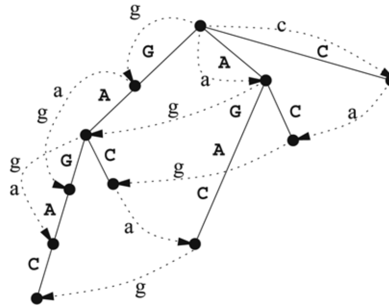

Fig. 2. Compact affix tree for the string GAGAC. Solid lines indicate edges of the suffix tree. Dotted lines indicate edges of the reverse prefix tree.

of a node q of the affix tree are the descendants of the corresponding node in the suffix tree and the *prefix-descendants* of q are the descendants of the corresponding node in the reverse prefix tree. With respect to the suffix tree the edges of the reverse prefix tree can be considered as the reverse of suffix links[1] that are a by-product of the compact suffix tree construction [9]. Since each node can be simultaneously regarded as a member of the suffix tree and the reverse prefix tree, the affix tree allows for the combination of suffix tree and reverse prefix tree applications and hence for bidirectional applications.

3. The affix array data structure

The affix array data structure combines the suffix array and the reverse prefix array and can be considered as a space-optimized variant of the affix tree with simplified and thus accelerated memory access. First we consider some elementary properties of virtual nodes with respect to suffix and reverse prefix array.

Lemma 3.1. *Given a text S of length n. If a virtual node q in the suffix array of S is multiple (singular) the node r representing the string $\overleftarrow{q}$ in the reverse prefix array of S is also multiple (singular).*

Proof. If q is singular we can find exactly one suffix Suf_i with prefix $\overrightarrow{q}$ respectively the substring $\overrightarrow{q}$ occurs exactly once in S. This means, the substring $\overleftarrow{q}$ occurs exactly once in the reverse prefix array, thus the corresponding node is singular. If q is multiple the reversal of the proof yields that r is multiple. □

Definition 3.1. For every l-interval $[i..j]$ in the data structure D (suffix array or reverse prefix array[2]) we define its common prefix string:

$$\{[i..j]\}_D := (0 : \mathsf{Suf}_i : l).$$

Suf_i is the i-th suffix with respect to the table suftab_D. (Notice that for $D = \mathcal{P}$ the i-th suffix Suf_i is defined with respect to the reverse text S^{-1}.)

Lemma 3.2. *For every l-interval $I = [i..j]$ in the data structure D there exists exactly one l'-interval $I^{-1} = [i'..j']$, denoted* the reverse interval *of I, in the reverse structure D^{-1} with $l' \geq l$ such that the common prefix of I^{-1} starts with $\{I\}_D{}^{-1}$:*

$$\{I^{-1}\}_{D^{-1}} = (\{I\}_D)^{-1} - .$$

For the borders of I^{-1} we find: $j' - i' = j - i$.

[1] A suffix link in the suffix tree points from a node $\overline{\omega s}$ to $\overline{s}$ where ω is a string over the underlying alphabet Σ.

[2] In order to distinguish between the tables of suffix and reverse prefix array we add data structure identifiers:

- $\mathcal{S}$ for the suffix array
- $\mathcal{P}$ for the reverse prefix array.

For example, $\mathsf{suftab}_\mathcal{S}$ denotes the suftab table in the suffix array while $\mathsf{suftab}_\mathcal{P}$ stands for the table suftab in the reverse prefix array. Here $S_\mathcal{P}$ denotes the reverse text S^{-1} while $S_\mathcal{S}$ denotes the original text S.

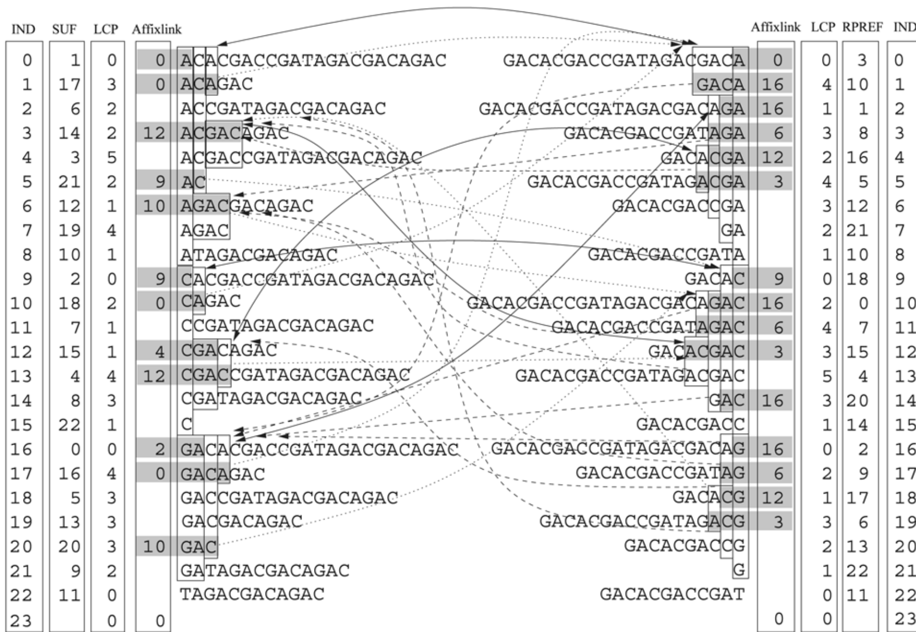

Fig. 3. Affix array for the text $S = \text{GACACGACCGATAGACGACAGAC}$. The affix links between the suffix array (on the left side) and the reverse prefix array (on the right side) are represented by arcs pointing from the *home* position (see Section 3.2) of an lcp-interval $[i..j]$ to the left index i' of the corresponding reverse interval $[i'..j']$. (Dotted arcs point from the suffix array towards the reverse prefix array while dashed arcs point in the reverse direction. Solid arcs point in both directions.) Each box indicates an lcp-interval in the suffix array respectively reverse prefix array.

Proof. Consider a text S of length n. Since $[i..j]$ is an l-interval there exist exactly $j-i+1$ suffixes $\mathsf{Suf}_i, \ldots \mathsf{Suf}_j$ respectively text positions $\mathsf{suftab}_D[i], \ldots \mathsf{suftab}_D[j]$ starting with $\{[i..j]\}_D = \mathsf{Suf}_i -$. Consequently there exist exactly $j-i+1$ positions $r_k = n - (\mathsf{suftab}_D[k] + l - 1)$ with $i \le k \le j$ in the reverse text respectively exactly $j-i+1$ reverse prefixes p_k in the reverse structure D^{-1} starting with $\{[i..j]\}_D{}^{-1}$. Let $Z = \{z_k \mid \mathsf{suftab}_{D^{-1}}[z_k] = r_{k+i}$ and $0 \le k \le j-i+1\}$. Since the table $\mathsf{suftab}_{D^{-1}}$ is ordered alphabetically, no index $x \notin Z$ exists with $z_{i_1} \le x \le z_{i_2}$ for any $z_{i_1}, z_{i_2} \in Z$. Hence, $[\min Z .. \max Z]$ is an l'-interval in the reverse prefix array containing the $j-i+1$ reverse prefixes $p_i, \ldots, p_j$. Since all p_k $(i \le k \le j)$ start with $\{[i..j]\}_D{}^{-1}$ we know that $l' \ge l$.

If $S[\mathsf{suftab}_D[i]-1] = S[\mathsf{suftab}_D[i+1]-1] = \cdots = S[\mathsf{suftab}_D[j]-1]$ then $l' \ge l+1$ because all the reverse prefixes $p_i \ldots p_j$ are identical in position l, otherwise $l' = l$. □

It follows that in general $(I^{-1})^{-1} \neq I$ for an l-interval I. (As an example consider Fig. 3: $I = [0..1]$ is a 3-interval in the suffix array, while the reverse interval $I^{-1} = [0..1]$ in the reverse prefix array is a 4-interval and $(I^{-1})^{-1} = [16..17] \neq I$ is also a 4- interval in the suffix array.)

Definition 3.2. The *affix array* of a text S is the union of suffix array and reverse prefix array, each extended by an additional table $\mathsf{affixlinktab}$ which, for each lcp-interval in the data structure D, contains a pointer to its reverse interval in the reverse data structure D^{-1}, called *affix link*:

$$\mathsf{affixlinktab}_D[I] = I^{-1}.$$

In the following I will show that this definition makes sense in a way that allows for efficient bidirectional applications on the proposed data structure. In particular, I will give a detailed specification of the structure of the table $\mathsf{affixlinktab}$, which here is defined on lcp-intervals. I will show that for each lcp-interval exists a unique index, which is denoted *home* position and can be used to store the affix links in the table $\mathsf{affixlinktab}$.

Definition 3.3. Balanced interval.

An l-interval I is called balanced if the reverse interval I^{-1} is an l-interval in the reverse data structure.

Lemma 3.3. *If the interval I in the data structure D is balanced the reverse interval I^{-1} is also balanced in the reverse structure D^{-1} and $I = (I^{-1})^{-1}$.*

Proof. If the l-interval $I = [i..j]$ is balanced I^{-1} is an l-interval and thus $\{I\}_D = (\{I^{-1}\}_{D^{-1}})^{-1}$. By Lemma 3.2 $\{(I^{-1})^{-1}\}_D = (\{I^{-1}\}_{D^{-1}})^{-1}-$.

Assume I^{-1} is not balanced, then $(I^{-1})^{-1}$ is an l'-interval with $l' > l$ and $\{(I^{-1})^{-1}\}_D = (\{I^{-1}\}_{D^{-1}})^{-1}\dashv$. We obtain $\{(I^{-1})^{-1}\}_D = \{I\}_D\dashv$. This means that all suffixes $\mathsf{Suf}_i, \ldots, \mathsf{Suf}_j$ in the data structure D not only have the common prefix $\{I\}_D$ (of length l) but $\{(I^{-1})^{-1}\}_D$ (of length $l' > l$) which means that I would have to be an l'-interval which is not the case. Consequently it follows that I^{-1} is also balanced. □

Lemma 3.4. *If the l-interval $I = [i..j]$ is not balanced in the data structure D the reverse l'-interval I^{-1} is a balanced interval in the reverse data structure DS^{-1}. In this case $I \neq (I^{-1})^{-1}$.*

Proof. In the same way as before we conclude

$$\{I^{-1}\}_{D^{-1}} = (\{I\}_D)^{-1}\dashv \Rightarrow (\{I^{-1}\}_{D^{-1}})^{-1} = \vdash\{I\}_D \Rightarrow \{I\}_D = (l' - l : (\{I^{-1}\}_{D^{-1}})^{-1}).$$

Assume I^{-1} is not balanced then $(I^{-1})^{-1}$ is an l''-interval with $l'' > l'$ and $\{(I^{-1})^{-1}\}_D = (\{I^{-1}\}_{D^{-1}})^{-1}\dashv$. We obtain

$$(l' - l : \{(I^{-1})^{-1}\}_D) = \{I\}_D\dashv.$$

By Lemma 3.2 we have $j - i + 1$ suffixes in D starting with $\{(I^{-1})^{-1}\}_D$. We conclude that the $j - i + 1$ suffixes $\mathsf{Suf}_i, \ldots, \mathsf{Suf}_j$ not only have the common prefix $\{I\}_D$, but also $(l' - l : \{(I^{-1})^{-1}\}_D)$. This means that I would be an $(l + l'' - l')$-interval which is not the case. Hence, the assumption is wrong and I^{-1} is balanced. □

It follows that for every l-interval I the reverse interval is balanced. For example, in Fig. 3 the 3-interval [0..1] in the suffix array $\mathcal{S}$ with $\{[0..1]\}_{\mathcal{S}} = \text{ACA}$ is not balanced while the reverse 4-interval [0..1] in the reverse prefix $\mathcal{P}$ array with $\{[0..1]\}_{\mathcal{P}} = \text{ACAG}$ is balanced.

3.1. Bidirectional traversals on the affix array data structure

In unidirectional data structures like the suffix tree or the reverse prefix tree, the analysis of the underlying text either happens in the forward or in the backward direction. This means, with each step from the root in direction of the leaves the set of strings under consideration is extended in the same direction. For example in the suffix tree we can find all occurrences of a pattern p in a text S by matching the characters of p from left to right, respectively traversing the corresponding suffix tree nodes.

The affix tree allows for bidirectional text analysis which means that at each step we can either continue processing in the forward or in reverse direction by following either the edges of the suffix tree or the edges of the reverse prefix tree. With respect to string matching we can start at any pattern position and at each step extend the matches in the set of matching substrings either on the right or on the left side.

In the following we will see how bidirectional traversals on the affix tree can be transferred to traversals of suffix array and reverse prefix array, in such a way that for each explicit or implicit virtual node under consideration the direction of string processing can be switched in constant time. In the case of explicit virtual nodes this can easily be done by a look-up in the table $\mathsf{affixlinktab}$ which contains a direct link to the reverse data structure for each lcp-interval. Since $\mathsf{affixlinktab}$ is not defined for implicit virtual nodes a constant time switch of the search direction for this kind of node is required. Therefore the concept of virtual nodes is extended by *left context* nodes of value k, in shorthand notation denoted as k-context nodes.

Definition 3.4. A *left context node* of value k is a pair $v = \langle k, q\rangle$ where $q = (i, d)$ is a multiple virtual node and no (singular or multiple) virtual node $p \neq q$ exists with $|\overrightarrow{p}| = |\overrightarrow{q}|$ and $(k : \overrightarrow{q} : d - 1) = (k : \overrightarrow{p} : d - 1)$. The forward string $\overrightarrow{q}$ represented by v is defined as $\overrightarrow{v} = (k : \overrightarrow{q} : d - 1)$. Correspondingly we define $\overleftarrow{v} = (k : \overrightarrow{q} : d - 1)^{-1} = \overrightarrow{v}^{-1}$. This means all occurrences of the substring represented by v share the same left context $(0 : q : k - 1)$ of length k in the underlying text. In other words, v corresponds to the multiple virtual node q but ignores the first k characters of $\overrightarrow{q}$ which are uniquely determined by $\overrightarrow{v}$.

A 0-context node $\langle 0, q\rangle$ is identical to q. With respect to pattern matching $\langle k, q\rangle$ can be considered as a special form of the node q where the first k positions of $\overrightarrow{q}$ have not been explored so far. $\langle k, q\rangle$ is called explicit (implicit) if q is explicit (implicit).

Lemma 3.5. *Given an l-interval $I = [i..j]$ in the data structure D. For every 0-context node $v = \langle 0, ([i..j], d)\rangle$ the pair $v' = \langle l - d, (\mathsf{affixlinktab}[I], l)\rangle$ is an $(l - d)$-context node in the reverse structure D^{-1} and $\overleftarrow{\overrightarrow{v}} = v'$.*

Proof. For simplicity we only consider $D = \mathcal{S}$ (suffix array). For $D = \mathcal{P}$ (reverse prefix array) we just have to switch the notation.

(1) Since v is an explicit or implicit node of the l-interval I it follows that $l \geq d$. Hence, every suffix in $\mathcal{S}$ with prefix $\overrightarrow{v}$ has the prefix $\{I\}_{\mathcal{S}}$: $\{I\}_{\mathcal{S}} = \overrightarrow{v}-$.
(2) We know that $\{I^{-1}\}_{\mathcal{P}} = \{I\}_{\mathcal{S}}{}^{-1}-$. This means that all reverse prefixes in $I^{-1} = [i'..j']$ denoted $\mathsf{Pref}_{i'}, \dots \mathsf{Pref}_{j'}$ have the common prefix $\{I\}_{\mathcal{S}}{}^{-1}$ of length l representing the 0-node $\langle 0, (\mathsf{affixlinktab}[I], l)\rangle$ in $\mathcal{P}$.

Assume $v' = \langle l - d, ([i'..j'], l)\rangle$ in $\mathcal{P}$ representing $\overrightarrow{v}^{-1}$ is not an $(l - d)$-context node. From Lemma 3.1 we know that the virtual node $([i'..j'], l)$ in $\mathcal{P}$ is multiple. Hence, the assumption v' is not a left-context node implies that there exists a reverse prefix $\mathsf{Pref}_{h'}$ $(h' \notin I')$ in $\mathcal{P}$ with $(l - d : \mathsf{Pref}_{h'} : l) = (l - d : \{I^{-1}\}_{\mathcal{P}} : l)$ but $(0 : \mathsf{Pref}_{h'} : l) \neq (0 : \{I^{-1}\}_{\mathcal{P}} : l)$. Thus there exists a suffix Suf_h in $\mathcal{S}$ $(h \notin I)$ with $\mathsf{Suf}_h = \overrightarrow{v}-$ but not $\mathsf{Suf}_h = \{I\}_D-$ in contradiction to conclusion 1. □

Lemma 3.6. *For every k-context node $v = \langle k, (i, d)\rangle$ with $k > 0$ the pair $\langle k - 1, (i, d)\rangle$ is an $(k - 1)$-context node.*

Proof. Since v is a k-context node no virtual node $(i', d) \neq (i, d)$ exists with $(k : \mathsf{Suf}_i : d) = (k : \mathsf{Suf}_{i'} : d)$.

Assume $\langle k - 1, (i, d)\rangle$ is not a $(k - 1)$-context node. The assumption implies that either the node (i, d) is not multiple which is not the case, or there exists at least one virtual node (j, d) with $(0 : \mathsf{Suf}_i : k - 1) \neq (0 : \mathsf{Suf}_j : k - 1)$ and $(k - 1 : \mathsf{Suf}_i : d) = (k - 1 : \mathsf{Suf}_j : d)$. Suppose the latter is true, this means $(0 : \mathsf{Suf}_i : k) \neq (0 : \mathsf{Suf}_j : k)$ and also $(k : \mathsf{Suf}_i : d) = (k : \mathsf{Suf}_j : d)$. Consequently v is not a k-context node in contrast to the assumption. □

Unidirectional traversals of the suffix array visit multiple virtual nodes respectively 0-context nodes and singular virtual nodes. Since a left context node $\langle k, q\rangle$ only ignores the left k-context of $\overrightarrow{q}$, any child node $q*$ of q which can be located in constant time, can be considered as a child node of $\langle k, q\rangle$. If $q*$ is multiple then obviously $\langle k, q*\rangle$ is a left-context node, otherwise we denote $\langle k, q*\rangle$ as *singular left-context child node* of value k, or shorthand singular k-context child. We extend this definition by specifying that any singular virtual node v is denoted singular 0-context child node. Simply spoken this means that the set of virtual nodes with left context is separated into two classes: (multiple) left-context nodes and singular left context child nodes.

Definition 3.5. A *k-context* affix node (or short *affix node*) is a pair $v = \langle k, q\rangle$ (with $d = |\overrightarrow{q}| > k$) in the suffix array (on the reverse prefix array the notation has to be switched) where either

(1) $\langle k, q\rangle$ is a k-context node with $k \geq 0$, or
(2) $\langle k, q\rangle$ is a singular k-context child with $k \geq 0$ or $k < 0$.

In the first case the forward string of v is defined as $\overrightarrow{v} = (k : \overrightarrow{q} : d - 1)$. In the second case q is singular and hence, q is a pair $([i..i], d)$, and represents exactly one substring $(\mathsf{suftab}[i] : S : \mathsf{suftab}[i] + d - 1)$ of the underlying text S. The forward string $\overrightarrow{v}$ is defined as $(\mathsf{suftab}[i] + k : S : \mathsf{suftab}[i] + d - 1)$ where $k < 0$ means an extension of $\overrightarrow{q}$ to the left.

Obviously traversals of virtual nodes in the suffix array respectively reverse prefix array are equivalent to traversals of the corresponding 0-context affix nodes in the affix array. In the following we will show that bidirectional traversals of the affix tree data structure are equivalent to bidirectional traversals of affix nodes in the affix array.

Each node of the affix tree of a text S can be considered as a node of the corresponding suffix tree and simultaneously as a node of the corresponding reverse prefix tree. In the following let $\mathcal{R}(q)$ denote the set of d-substrings of the text S that are identical to the forward string $\overrightarrow{q}$ of length d. In the affix tree we reach the child nodes of q in depth $d + 1$

(which may be explicit or implicit and which represent one-character extensions of substrings in $\mathcal{R}(q)$ to the right) in the suffix tree in constant time. Correspondingly we reach the child nodes of q in the reverse prefix tree (representing extensions of substrings in $\mathcal{R}(q)$ to the left) in constant time.

Lemma 3.7. *Given an affix node $v = \langle k, q\rangle$ with $q = ([i..j], d)$ in the affix array of the text S. We find the suffix children*[3] *of v, which represent one-character extensions of substrings in $\mathcal{R}(v)$ to the right, in constant time. Correspondingly we find the prefix children of v in constant time.*

Proof. We consider different kinds of nodes in the suffix array separately. For the reverse prefix array we just have to switch the notation:

(1) The suffix children of v can be located in constant time in the usual way:
- if v is a (multiple) left-context node we use the child table information in order to locate the suffix children of q in constant time (which may be multiple or singular) and then consider their k-context variants.
- if v is a singular k-context child, q and also v represent exactly one substring of S. Hence the suffix child can be located by evaluation of $S[\mathsf{suftab}[i] + |\vec{q}|]$, if present.

(2) In order to find the prefix children of $v = \langle k, q\rangle$ with $q = [i..j]$ we have to switch to the reverse direction:
- If v is a singular k-context child we evaluate $S[\mathsf{suftab}[i] + k - 1]$ which leads to the singular $(k - 1)$-context child $\langle (k - 1), q\rangle$, if present.
- In case of a (multiple) left-context node of value $k > 0$ we know that all substrings in $\mathcal{R}(v)$ have an identical left context of length k. Hence, there is exactly one left child of v which, according to Lemma 3.6, is the left-context node $\langle (k - 1), q\rangle$ of value $k - 1$.
- If $v = \langle 0, ([i..j], d)\rangle$ is a 0-context node we proceed in the reverse prefix array: By Lemma 3.2 we know that $I = [i..j]$ is an l-interval with $d \le l$ and reverse l'-interval I^{-1} with $l \le l'$. We switch to the *reverse node*
  $$v^{-1} = \langle l - d, (\mathsf{affixlinktab}[I], l)\rangle$$
  in the reverse prefix array. According to Lemma 3.5 v^{-1} is a left-context node and represents the reverse string $\vec{q}^{\,-1}$. If I is balanced then by Lemma 3.3 we know that $l = l'$ and v^{-1} is explicit. If I is not balanced then by Lemma 3.4 we know that $l < l'$ and v^{-1} is implicit. □

3.2. Implementation of the table affixlinktab

By Definition 3.2, for each lcp-interval affixlinktab contains a pointer to its reverse interval. affixlinktab can be implemented by two tables (for the suffix array and for the reverse prefix array) each of size $4n$ (where n is the length of the text) containing a link to an lcp-interval in the reverse structure for each lcp-interval. In order to keep the data structure small we only store the left border of an lcp-interval which costs 4 bytes per field. According to Lemma 3.2 the length of the reverse interval $I^{-1} = \mathsf{affixlinktab}[I]$ is identical to I, thus the right border of the interval can be omitted.

The number of lcp-intervals in the suffix array is at most $n - 1$, which means there are enough places to store all links. To avoid collisions we need a unique position for every lcp-interval to store the link. Therefore I present a new elementary property of lcp-intervals (respectively explicit virtual nodes): For each lcp-interval there exists a unique position which can be computed by a simple comparison of lcp-values. This position can be used to store specific information for each lcp-interval.

Definition 3.6. home of an lcp-interval

For an l-interval I=$[i..j]$ we define

$$\mathsf{home}(I) = \begin{cases} i, & \text{if } \mathsf{lcptab}[i] \ge \mathsf{lcptab}[j+1] \\ j, & \text{otherwise.} \end{cases}$$

For the root interval $[0..n]$ representing the empty string we define
$\mathsf{home}([0..n]) = n$.

[3] A *suffix child* of an affix node v is a child node of v in the appertaining suffix array while a *prefix child* is a child node of v in the appertaining reverse prefix array.

Lemma 3.8. $\mathsf{home}(I) = \mathsf{home}(I') \Rightarrow I = I'$.

Proof. Given an l-interval $I = [i..j]$ and an l'-interval $I' = [i'..j']$. For two l-intervals we know that either I encloses I', I' encloses I or I and I' are disjunct. In other words, there is no *non-enclosing intersection* between two intervals.
Now assume that $I \neq I'$.

(1) Consider the case $\mathsf{home}(I) = i$: If $\mathsf{home}(I') = j'$ then $i = j'$ because $\mathsf{home}(I) = \mathsf{home}(I')$. But then $i' < j' = i < j$ which cannot be the case for any two l-intervals (no non-enclosing intersection).

$$\Rightarrow \mathsf{home}(I') = i' = i.$$

We still have to show that $j = j'$. Without loss of generality assume $j' < j \Rightarrow l < l'$. Since $\mathsf{home}(I') = i'$ we know by definition of home that $\mathsf{lcptab}[i'] \geq \mathsf{lcptab}[j' + 1]$. By definition of l-intervals we find $l > \mathsf{lcptab}[i] = \mathsf{lcptab}[i']$ (since $i = i'$).

$$\Rightarrow l > \mathsf{lcptab}[i'] \geq \mathsf{lcptab}[j' + 1].$$

By definition of l-intervals we also know that $\mathsf{lcptab}[k] \geq l, \forall k \in [i + 1..j]$ which contradicts $l > \mathsf{lcptab}[j' + 1]$ since $j' + 1 \in [i + 1..j]$.

(2) Consider the case $\mathsf{home}(I) = j$. Equivalently to the first case we conclude $\mathsf{home}(I') = j' = j$.

Without loss of generality we assume $i < i' \Rightarrow l < l'$. By definition of home we obtain $\mathsf{lcptab}[i'] < \mathsf{lcptab}[j' + 1]$. Since $j = j'$ we conclude $\mathsf{lcptab}[i'] < \mathsf{lcptab}[j + 1]$.

By definition of l-intervals we know that $\mathsf{lcptab}[j + 1] < l$ and thus $\mathsf{lcptab}[i'] < l$. By definition of l-intervals we also know that $\mathsf{lcptab}[k] \geq l, \forall k \in [i + 1..j]$ which contradicts $l > \mathsf{lcptab}[i']$ since $i' \in [i + 1..j]$. Hence, we have shown $i = i'$. □

3.3. Linear time construction of the table affixlinktab

From [10] and [8] we know how to construct the (enhanced) suffix array and thus also how to construct the (enhanced) reverse prefix array in linear time and space. Hence, a linear time construction of the table affixlinktab provides a linear time construction for the affix array data structure. A simple algorithm solves the affixlinktab construction in quadratic time in the worst case but is much faster on average:

- In the first step the lcp-intervals of the suffix array are ordered in alphabetical order of their reverse common prefixes which consumes quadratic time in the worst case.
- In the second step which runs in linear time, the ordered list and the lcp-intervals of the reverse prefix array (in ascending alphabetical order) are traversed simultaneously while affix links are generated for each list element.

The construction of the table affixlinktab can be done in linear time using the compact affix tree which is linear in (construction) time and space [4]. This method has the disadvantage that an enormous amount of space is required for the construction of the compact affix tree (more than 45 bytes per text symbol).

In the following a linear time construction of the table affixlinktab is presented avoiding the construction of the affix tree structure. This algorithm makes use of a temporary table suflinktab which can be constructed in linear time as shown in [8].

Definition 3.7. Given an l-interval I with $\{I\}_D = \omega s$ and $\omega \in \Sigma$. The $(l - 1)$-interval I' with $\{I'\}_D = s$ is called the suffix link interval of I.

Since suffix array and suffix tree are equivalent data structures and lcp-intervals in the suffix array correspond to explicit nodes in the tree, it directly follows from the definition of suffix trees that in each case there exists exactly one such $(l - 1)$- interval I'.

Definition 3.8. For each lcp-interval I with suffix link interval $[i'..j']$

$$\mathsf{suflinktab}[\mathsf{home}(I)] = i'.$$

In [8] and [11] a linear time construction of the table suflinktab is presented. In [8] the construction is based on *range minimum queries* introduced in [10] for the construction of the suffix array in linear time.

For the construction and further applications of suflinktab in [8] the left index i and the right index j is stored (consuming 8 bytes) to represent the suffix link interval l-$[i..j]$. In a memory critical environment the right border may be omitted since we can find j in constant time if we only know i and l and the child table information is given. Nevertheless, if space does not play an important role also the right index should be stored, since algorithms based on the suffix link table (including the construction of the table) run faster.

I propose a second way to construct the table suflinktab in linear time which expects that the enhanced suffix array is already given (see Fig. 4). The construction reflects the linear time construction of the suffix tree based on suffix links [9]. The main idea is the traversal of prefixes $(0 : S : i)$ $(i = 0, 1, \dots n)$ of the text S. At each step we consider the *actual* lcp-interval which represents the longest multiple suffix of the actual prefix $(0 : S : i)$ and compute an *active* link which is the suffix link lcp-interval of the active lcp-interval.

(1) If the $S[i+1]$ child of the actual lcp-interval I represents a multiple node (which is found in constant time, because the suffix array is already present), the next actual lcp-interval I_A is the $S[i+1]$ child of I.
   (a) If suflinktab[home(I_A)] has already been computed, then we have to do nothing and continue with the next character of S.
   (b) Otherwise we locate the next active link interval L_A in constant time, which is the $S[i+1]$ child of the present active link, and update the table suflinktab: suflinktab[home(I_A)] $= L_A$.

(2) Otherwise, we follow the suffix link path of I until a node with multiple child is found which is the next actual lcp-interval.

In total the number of visited lcp-intervals in (1) respectively the number of visited suffix links in (2) is limited by n:

(1) If implicit virtual nodes are traversed (in cases the lcp-value of the next active lcp-interval, with respect to the current, increases by the value 2 or more) we have to consider *implicit* suffix links which point from an implicit virtual node $\overline{\omega s}$ to an explicit virtual node $\overline{s}$ but are not stored in affixlinktab. This means, when traversing the path from the current active suffix link to the next active suffix link, we might have to locate lcp-intervals, that are suffix link of an implicit virtual node and have been traversed earlier, hence no new entry is added to the table suflinktab. When traversing the text from left to right, with each explicit or implicit virtual node that is visited we explore one character of the text and thus, at most n *actual* lcp-intervals and at most n *implicit* suffix links have to be evaluated.

(2) Although suffix links may be visited several times, in total the number of visited suffix links is limited by n: from any l-interval under consideration at most l suffix links are traversed to find the next interval with multiple child. When traversing the text from left to right, with each new character the actual lcp-value increases by 1. Thus, in total at most n suffix links are traversed.

From (1) and (2) we conclude that the construction of the table suflinktab is linear in time.

3.3.1. *Construction of the table* affixlinktab *with suffix links*

Given the table suflinktab we can construct the table affixlinktab in linear time and space. The main idea of the algorithm is the recursive computation of reverse intervals, in each case based on reverse intervals of suffix link intervals.

Consider the recursive function `reverseInterval` (see Algorithm 3.1) which is applied to the home position of an interval I in the suffix array and an integer `length` and returns the borders i' and j' of $I^{-1} = [i'..j']$ in the reverse prefix array.

The function `child` is applied to an lcp-interval I and a character $a \in \mathcal{A}$ in the reverse prefix array and returns the a-child lcp-interval of I which can be located in constant time via the child table. We store the right border of an lcp-interval $[i'..j']$ located in the reverse prefix array in a temporary table `affixlinktab_right` which consumes $4n$ extra bytes but can be deleted after the construction of affixlinktab.

The function `traverse` (see Algorithm 3.2) recursively traverses the lcp-intervals of the suffix array in ascending lexicographical order with initial call `traverse([0..n])` to start the construction of the table affixlinktab. The functions `firstInterval` and `nextInterval` which are called by `traverse` both consume constant time in order

Algorithm 3.1. *(Recursively Find Reverse Intervals)*

```
interval reverseInterval(integer home, integer length)
{
  if (affixlinktab[home]!=EMPTY)
     return [affixlinktab[home],affixlinktab_right[home]]
  revinterval=child(reverseInterval(suflinktab[home],length-1),
                            S[suftab[home]])
  affixlinktab[home]= Min(revinterval)
  affixlinktab_right[home]= Max(revinterval)
  return revinterval
}
```

Algorithm 3.2. *(Recursively Traverse lcp-Intervals)*

```
affixlinktab[n]=0 //initially set affix link for the root interval
traverse([0..n])  //initial call of traverse

traverse(interval start)
{
  nextchild=firstInterval(start)
  while (nextchild != NULL)
  {
    if (affixlinktab[home(nextchild)]==EMPTY)
    {
      revinterval=reverseInterval(home(nextchild),lvalue(nextchild))
      affixlinktab[home(nextchild)]= Min(revinterval)
      affixlinktab_right[home(nextchild)]= Max(revinterval)
    }
    traverse(nextchild)
    nextchild= nextInterval(start,nextchild)
  }
}
```

to locate the lexicographically first child interval of an lcp-interval I, respectively the lexicographically next lcp-interval of an lcp-interval I with respect to its parent interval J (or NULL if J is the last lcp-interval of parent I).

The number of lcp-intervals visited by the function `traverse` is at most $n - 1$ since we have at most $n - 1$ lcp-intervals. Each call of the function `reverseInterval` evokes a varying number of recursive subcalls. With each new child interval explored in the function `reverseInterval` a new affix link is located and stored in the table affixlinktab. Since the number of affix links is at most n and recomputation of affix links is avoided by a look-up in the table affixlinktab, we achieve linear running time.

3.4. Space requirements

In total, the space amount of the affix array data structure in its raw form is 18 bytes (without child table information) with 5 bytes for the suffix array (1 byte for the lcp-table and 4 bytes for the suffix index) and 5 bytes for the reverse prefix array plus an additional 8 bytes for the link tables (4 bytes each). In this form child intervals are located via binary search which consumes logarithmic time $\mathcal{O}(\log n)$ with respect to a text of length n.

Definition 3.9. The enhanced affix array of a text S is the affix array of S extended by the child-tables of the enhanced suffix array and the enhanced reverse prefix array.

The space amount increases to 20 bytes per symbol with the advantage that child intervals can be located in constant time and a pattern of length m can be located in time $\mathcal{O}(m)$.

3.5. A different approach to affix arrays

In contrast to the affix tree data structure the affix array does not fuse suffix array and reverse prefix array, but connects them via link tables in order to keep algorithms fast in both directions. There is no loss of speed with respect to unidirectional search since the main data structures remain unchanged.

A different approach to affix arrays extends the suffix array by adding the reverse suflinktab tables (we have to

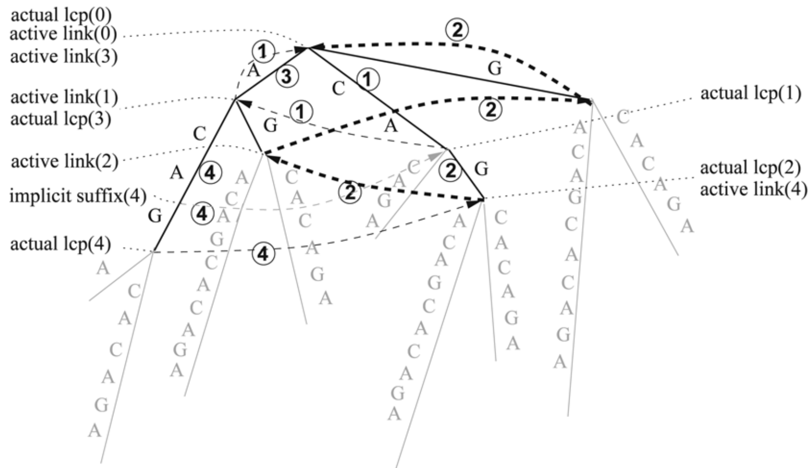

Fig. 4. Suffix link construction for the text S = CAGACAGCACAGA. In step 1 the 2-interval $\overline{\text{CA}}$ is located and the corresponding suffix links ($\overline{\text{CA}} \rightarrow \overline{\text{A}}$, $\overline{\text{A}} \rightarrow$ root), indicated by dashed vectors, are generated. In step 2 node $\overline{\text{CAG}}$ is located from $\overline{\text{CA}}$ with 3 new suffix links which are traversed in the following to find the next actual lcp-interval with multiple child (here the root node). In step 3 the virtual node $\overline{\text{A}}$ is located which has been visited earlier such that no additional suffix links are added. In step 4 $\overline{\text{ACAG}}$ is located from $\overline{\text{A}}$ with 1 additional suffix link to $\overline{\text{CAG}}$. When the next active link $\overline{\text{CAG}}$ is located from the root, $\overline{\text{CA}}$ has to be located a second time. This is indicated by the implicit suffix link from the implicit virtual node $\overline{\text{ACA}}$ to $\overline{\text{CA}}$.

add $|\mathcal{A}|$ tables of size $4 * n$ bytes, one for each alphabet symbol) which corresponds to a simulation of the suffix link edges of the affix tree without adding the reverse prefix array structure. Without further improvements this approach is inferior to the data structure presented here:

(1) Only for alphabet size 3 (and less) do we obtain reduced space requirements with respect to the presented affix array data structure ($3 * 4$ extra bytes per symbol which means 16–18 bytes per symbol in total).
(2) The performance of the reverse search decreases significantly because large (time consuming) jumps in the table suftab are necessary for a traversal in the reverse direction.

The usage of compressed indexes significantly accelerates backward searching [12,13] but since the performance is difficult to evaluate we avoid further comparisons in this paper.

4. Applications

4.1. RNA sequences and secondary structures

RNA is a sequence composed of the four different bases (nucleotides) **A**denine, **C**ytosine, **G**uanine and **U**racile. In the following we consider the corresponding RNA alphabet $\Sigma = \{\text{A}, \text{C}, \text{G}, \text{U}\}$. RNA molecules occur as single strands that fold on themselves due to the base pairing effects of the complementary bases A–U, G–C and G–U. One or more complementary bases bind together and form the secondary structure of the sequence. Hairpin structures, which are considered in the following, belong to the most important RNA secondary structures where two reverse complementary sequence parts[4] enclose a non pairing loop region.

4.2. Search for hairpin-like patterns

In this section, we look at applications where the benefit from the bidirectional affix array data structure justifies the additional efforts for the more complex structure compared to the suffix array. For simplification we discuss the simple

4 We consider the reverse complement of RNA: two sequences s and s' each of length $n > 0$ are considered reverse complementary, if

$$s[i] = \text{complement}(s'[n-i]), \text{ for all } i \in [0..n]$$

where the following pairs of nucleotides respectively bases are *complements*: (A, U), (U, A), (C, G), (G, C), (G, U), (U, G).

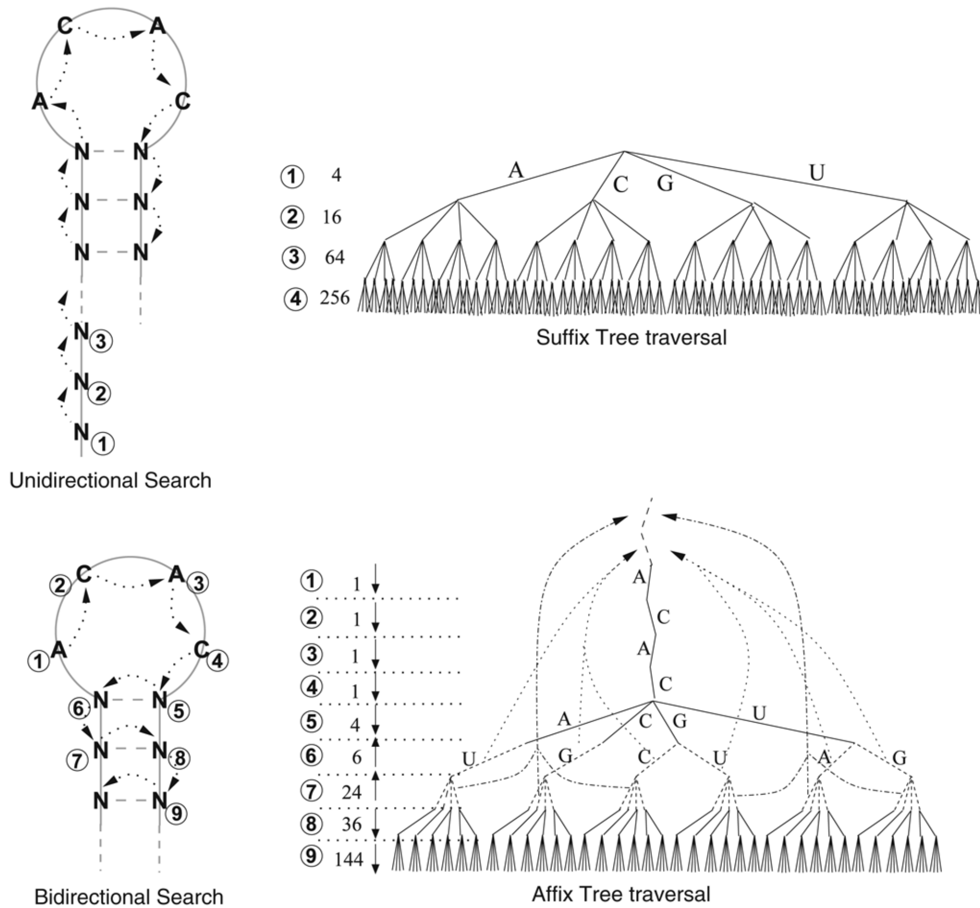

Fig. 5. Unidirectional vs bidirectional search of the `hairpin` pattern. On the right side the visited edges in the corresponding data structure are shown. In the suffix array/tree in the first steps the number of visited nodes increases by factor 4 at each step. In the affix array/tree initially 4 nodes are visited to match the interior `loop`. From step 5 to 6 the search direction is changed from forward to backward. In the tree the edges in depth 6 and 7 are represented by dashed lines since in fact a search in backward direction means a shift down of the branches considered so far, or in other words: we explore nodes on top of the ACAC branch, but for the sake of simplicity show them as suffix tree descendants. In steps 5, 7, 9, . . . a wildcard character is matched, thus in each case 4 additional subnodes per node are visited (under the assumption that the tree is complete up to depth 9). In steps 6 and 8 only 1.5 nodes have to be explored per node since complementary nucleotides are matched.

but important example of an RNA secondary structure hairpin loop. In the paper of Mauri et al. [5] the affix tree is used for the search of RNA secondary structure patterns with initial hairpin element by a breadth-first search algorithm. The algorithm can be transferred directly to the more efficient affix array data structure but should be replaced by a faster depth-first search algorithm which is presented in the following.

Suppose we search for the following *hairpin*-pattern written in the notation of the *hybrid pattern language* HyPaL [1]:

```
hairpin= stem loop ^(stem)
where stem:= N{5,50}
      loop:= ACAC
```

The pattern describes a hairpin structure with an apical loop, here specified by a concrete sequence ACAC (see Fig. 5). On the left and on the right side of the loop two reverse complementary nucleotide sequence parts `stem` and `^stem` follow, each consisting of 5–50 arbitrary nucleotides represented by the wildcard character *N* that matches any alphabet symbol.

In the suffix array/tree we search the pattern from left to right. Hence, we cannot make use of complementary nucleotides in order to shrink the search space in an early state. Instead we have to explore the complete search space

Algorithm 4.1. *(Recursively Match Hairpin Patterns)*

```
match(node start, integer depth) {
  direction= round((depth+1)/2) mod 2 //changes every two positions
  if (depth mod 2==0) {
    if (depth>=minlength)
      reportmatch(start)
    if (depth<maxlength) {
      forall (i in alphabet)
        match(childNode(start,i,direction), depth+1)
  } }
  else {
    if (depth<maxlength) {
      forall (i in alphabet) {
        if complement(start,i)
          match(childNode(start,i, direction), depth+1)
} } } }
```

specified by the `stem` part (all tree nodes with depth below 51) before matching the significant loop and the reverse complementary part specified by `^stem`.

In the affix array data structure we initially locate the affix node representing the loop sequence ACAC which means traversing at most 4 affix nodes. In the following, at each point we can expand the search in both directions which means we can explore complementary positions in subsequent steps:

(1) Match a wildcard symbol in the current match direction. This means visiting all (at most 4) suffix respectively prefix children of the current affix node.
(2) Match the complementary nucleotide with respect to the nucleotide matched in the previous step in the reverse direction. This means visiting on average $k(1.5/4)$ of the k prefix respectively suffix children of the current affix node.

Every two steps the maximal number of nodes that has been explored increases by factor 6 (compared to factor 16 in the suffix array/tree). When we reach nodes in depth 14 (4 `loop`-symbols plus `2*5` `stem`-symbols) and more the corresponding strings can be reported as matches. This means the algorithm locates matches iteratively and stops when no further complementary nucleotides can be found.

The recursive pseudo-code function `match` (see Algorithm 4.1) locates all occurrences of hairpin patterns containing `minlength` to `maxlength` basepairs by a depth-first search and pushes at most `depth` virtual nodes on the stack. With respect to the initial call `match(startnode,0)`, `startnode` may specify an intermediate result of a previous computation, in our example the virtual node $\overline{\text{ACAC}}$ representing the `loop`.

The function `childNode(start,i,direction)` returns the i-suffix child respectively i-prefix child of *start* with respect to `direction`. Depending on the `depth`-index, which represents the depth of the actual node `start` in the tree, the search direction switches every two steps from forward search (direction==0) to backward search (direction==1) and back. The function `reportmatch(start)` reports all matches corresponding to leaves below the virtual node `start`. The function `complement(start,i)` returns `true` if the nucleotides represented by `i` and the last character of $\overrightarrow{start}$ are complementary, otherwise it returns false.

If we compare the matching of complementary pairs of wildcard symbols the ratio of affix to suffix array nodes that have to be visited is $6/16$. The affix array search benefits from the rapidly decreasing tree density with increasing depth because in the "beginning" near to the root the number of visited nodes is small while in large depth less nodes have to be visited due to the minimal density of the tree.

In [5] a breadth-first search algorithm is proposed. Starting in depth d of the affix tree all complementary pairs in depth $d + 1$ and depth $d + 2$ are evaluated and stored (by 2 extra pointers in each case) before deeper nodes are considered. For hairpin-patterns with unspecific initial loop region (for example 8 or more wildcard characters in the pattern) this means an enormous memory overhead because the number of nodes that have to be managed simultaneously grows exponentially in the size of the alphabet with increasing tree depth. For example for an unspecific loop of 8 wildcard characters $6 * 4^8 = 384\,000$ pointer pairs have to be stored after the first basepair has been matched (if the tree is complete up to depth 8), and for large scale applications with sequence length far above one million characters the corresponding trees are complete up to depth 10 and more.

Table 1
Patterns that are used for the comparison of running times shown in Table 2

Pattern	Structure	Pattern	Structure
hairpin1= (stem:=N{20,50}) (loop:=NNN) ^stem	loop stem ^stem 20–50	hairpin2= (stem:=N{10,50}) (loop:=GGAC) ^stem	G A loop G C stem ^stem 10–50
hairpin3= (stem:=N{15,50}) (loop:=GAGAC) ^stem[1,1,1]	G A A loop G C stem ^stem 15–50 1 Deletion 1 Insertion 1 Mismatch	hairpin4= (stem:=N{10,15}) (loop:=GGAC[0,0,1]) ^stem	1 Insertion G A loop G C stem ^stem 10–15
bulge1= (stem0:=N{10,20}) (bulge:=N{4}) (stem:=N{5,10}) (loop:=NNN) ^stem ^stem0	loop stem ^stem 5–10 bulge stem0 ^stem0 10–20	bulge2= (stem0:=N{5,20}) (bulge:=AC) (stem:=N{1,20}) (loop:=GACAC[0,0,2]) ^stem ^stem0	2 Insertions C A A loop G C stem ^stem 10–15 C A stem0 ^stem0 5–20
sequence=CAGUAGAAA	CAGUAGAAA		

Patterns are shown in the notation of the pattern matching language HyPaL [1]. Approximate patterns are written in the form p[a,b,c] and match the pattern p, where at most a mismatches, b deletions and c insertions are allowed.

Our depth-first search algorithm presented above avoids this memory overhead. In fact, if we consider a node in depth d only d ancestors have to be kept in memory, which is negligible since the evaluation depth is determined by the pattern length. Although the number of nodes that have to be visited is the same as in a breadth-first search algorithm, we benefit from the redundant effort for node management.

4.3. Experimental results

I have applied an extended version of the bidirectional affix array search algorithm (implemented in C) to different patterns, which are shown in Table 1, and to data sets of different size. For the construction of the underlying enhanced suffix array and enhanced reverse prefix array the program *mkvtree* of the *Vmatch* package [14] of Kurtz is used. I have compared the bidirectional search on the affix array with bidirectional search on the plain text and unidirectional search on the suffix array on the same architecture (1.6 GHz processor with 512 MB main memory) . Table 2 shows the different running times and the number of matches on the different data sets. In all cases the bidirectional affix array search performs best where the factor of acceleration strongly depends on the pattern. Compared to plain text search the affix array search performs best on hairpins with very short respectively very specific loops. This ratio decreases with decreasing specificity of the loop.

5. Conclusions

I have introduced a new data structure which is similar in its functionality to the affix tree data structure but consumes distinctly less memory and allows for faster algorithms corresponding to the acceleration of suffix array algorithms compared to suffix tree algorithms. The affix array can be constructed in linear time and space and allows for efficient bidirectional applications on large data sets like the search of RNA secondary structure patterns. Since

Table 2
Running times (in seconds) for matching different patterns specified in Table 1

Affix constr.	*Pyrococcus* 1.7 MB 8.2				*Escherichia coli* 4.7 MB 23.3				Hum-UTR 10.1 MB 49.1			
Pattern	hits	aff	txt	suff	hits	aff	txt	suff	hits	aff	txt	suff
Hpin1	0	3.1	31.8	226	11	6.9	84	595	6	12	170	1077
Hpin2	2	0.02	3.1	152	3	0.08	7.94	396	0	0.08	17.5	707
Hpin3	2	2.7	5.9	141	7	4.4	13.2	368	32	20.6	47.7	681
Hpin4	2	0.1	6.5	34	19	0.2	17.4	81.5	33	0.43	36	127
Bulge1	0	3.5	32	497	10	6.9	83.4	1495	92	12.7	168	2490
Bulge2	3	0.24	8.39	152	23	0.76	24.5	480	65	1.2	47.8	726
Seq	9	0	2.9	0	17	0	7.5	0	54	0	15	0
Hloop(5)	1	3.22	47.3	47.2	56	7.02	127	126	147	17.8	354	334
Hloop(10)	6	22.6	48.2	47.5	39	54	131	127.6	4384	103	376	329
ACloop(5)	0	0.1	18.2	18.6	0	0.06	48.9	51.0	2	0.46	128.2	123
ACloop(10)	0	4.6	18.4	18.3	0	6.8	49.2	50.4	0	9.26	129.1	125
ACloop(15)	0	13.8	18.6	18.8	0	36.6	50.4	50.2	0	78.2	130.3	127

In column *hits* the number of occurrences of the corresponding pattern in the examined data set is shown (genomes of *pyrococcus horicoshii* (1.7 MB) and *E. coli* (4.7 MB), and *human UTR* sequence data (10.1 MB)). Columns *aff*, *txt* and *suff* show the running times of bidirectional affix array search, bidirectional search on the plain text and unidirectional suffix array search. The pattern $Hloop(x)$ is a variable pattern with a loop consisting of x wildcards and a stem of length between 15 and 20. The pattern $ACloop(x)$ is a variable pattern with a loop consisting of x A- or C symbols and a stem of length between 15 and 20. The time for the initial construction of the affix array for each data set strongly depends on the system architecture (main memory). For the example data sets only a few seconds were required.

the affix array data structure is not restricted to RNA, it may be applied successfully to other bidirectional string processing problems.

Acknowledgement

This work was partially supported by grants from the Deutsche Forschungsgemeinschaft (KU 1257/1-3).

References

[1] D. Strothmann, S. Kurtz, S. Gräf, G. Steger, The syntax and semantics of a language for describing complex patterns in biological sequences, Report 2000-06, Technische Fakultät, Universität Bielefeld, 2000. URL: http://bibiserv.techfak.uni-bielefeld.de/HyPa/patternlanguage.ps.gz.
[2] S. Gräf, D. Strothmann, S. Kurtz, G. Steger, HyPaLib: A database of RNAs and RNA structural elements defined by hybrid patterns, Nucleic Acids Research 29 (1) (2001) 196–198. URL: http://nar.oupjournals.org/cgi/reprint/29/1/196.
[3] J. Stoye, Affix trees, Report Nr. 04, Technische Fakultät, Universität Bielefeld, 2000.
[4] M. Maass, Linear bidirectional on-line construction of affix trees, Algorithmica 37 (2003) 320–334.
[5] G. Mauri, G. Pavesi, Algorithms for pattern matching and discovery in RNA secondary structure, Theoretical Computer Science 335 (2005) 29–51.
[6] U. Manber, E. Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.
[7] M. Abouelhoda, S. Kurtz, E. Ohlebusch, The enhanced suffix array and its applications to genome analysis, in: Proceedings of the Second Workshop on Algorithms in Bioinformatics, in: Lecture Notes in Computer Science, vol. 2452, Springer-Verlag, 2002, pp. 449–463.
[8] M. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms 2 (2004) 53–86.
[9] E. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM 23 (2) (1976) 262–272.
[10] D. Kim, J. Sim, H. Park, K. Park, Linear-time construction of suffix arrays, in: Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2003, in: Lecture Notes in Computer Science, vol. 2089, Springer Verlag, 2003, pp. 186–199.
[11] M. Maass, Computing suffix links for trees and arrays, Information Processing Letters 101 (6) (2007) 250–254.
[12] Makinen, Navarro, Compressed full text indexes, ACM Computing Surveys 39 (2) (2007) Art. 2.
[13] J. Sim, Time and space efficient search for small alphabets with suffix arrays, in: 2nd Conference on Fuzzy Systems and Knowledge Discovery, Springer, 2005, pp. 1102–1107.
[14] S. Kurtz, http://www.vmatch.de.
[15] G. Mauri, G. Pavesi, Pattern discovery in RNA secondary structure using affix trees, in: Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2003, in: Lecture Notes in Computer Science, vol. 2676, Springer Verlag, 2003, pp. 278–294.
[16] G. Pesole, S. Liuni, M. D'Souza, PatSearch: A pattern matcher software that finds functional elements in nucleotide and protein sequences and assesses their statistical significance, Bioinformatics 16 (2000) 439–450.
[17] G. Grillo, F. Licciulli, S. Liuni, E. Sbisa, P. G, PatSearch: A program for the detection of patterns and structural motifs in nucleotide sequences, Nucleic Acids Research 31 (2003) 3608–3612.