Structator: fast index-based search for RNA sequence-structure patterns

Fernando Meyer[1], Stefan Kurtz[1], Rolf Backofen[2],
Sebastian Will[*2,3], and Michael Beckstette[*1]

[1]Center for Bioinformatics, University of Hamburg, Bundesstrasse 43, 20146 Hamburg, Germany
[2]Chair for Bioinformatics, University of Freiburg, Georges-Köhler-Allee 106, 79110 Freiburg, Germany
[3]Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Email: FM - meyer@zbh.uni-hamburg.de; SK - kurtz@zbh.uni-hamburg.de; RB - backofen@informatik.uni-freiburg.de; SW*- swill@csail.mit.edu; MB*- beckstette@zbh.uni-hamburg.de;

*Joint corresponding author

Abstract

Background: The secondary structure of RNA molecules is intimately related to their function and often more conserved than the sequence. Hence, the important task of searching databases for RNAs requires to match sequence-structure patterns. Unfortunately, current tools for this task have, in the best case, a running time that is only linear in the size of sequence databases. Furthermore, established index data structures for fast sequence matching, like suffix trees or arrays, cannot benefit from the complementarity constraints introduced by the secondary structure of RNAs.

Results: We present a novel method and readily applicable software for time efficient matching of RNA sequence-structure patterns in sequence databases. Our approach is based on affix arrays, a recently introduced index data structure, preprocessed from the target database. Affix arrays support bidirectional pattern search, which is required for efficiently handling the structural constraints of the pattern. Structural patterns like stem-loops can be matched inside out, such that the loop region is matched first and then the pairing bases on the boundaries are matched consecutively. This allows to exploit base pairing information for search space reduction and leads to an expected running time that is sublinear in the size of the sequence database. The incorporation of a new chaining approach in the search of RNA sequence-structure patterns enables the description of molecules folding into complex secondary structures with multiple ordered patterns. The chaining approach removes spurious matches from the set of intermediate results, in particular of patterns with little specificity. In benchmark experiments on the Rfam database, our method runs up to two orders of magnitude faster than previous methods.

Conclusions: The presented method's sublinear expected running time makes it well suited for RNA sequence-structure pattern matching in large sequence databases. RNA molecules containing several stem-loop substructures can be described by multiple sequence-structure patterns and their matches are efficiently handled by a novel chaining method. Beyond our algorithmic contributions, we provide with *Structator* a complete and robust open-source software solution for index-based search of RNA sequence-structure patterns. The *Structator* software is available at http://www.zbh.uni-hamburg.de/Structator.

Background

The discovery of new roles of non-coding RNAs (ncRNAs) has made them of central research interest in molecular biology [1, 2]. Like proteins, ncRNA sequences that have evolved from a common ancestor can be grouped into families. For instance, the Rfam database [3, 4] release 10.0 compiles 1,446 such families. Members of a family share, to different degrees, sequence and structure similarity. In many cases, however, the members of a family share only few sequence features, but share by far more specific structural and functional properties. Prominent examples of such cases are tRNAs and microRNA precursors.

In this paper, we consider the problem of searching nucleotide databases for occurrences of RNA family members. As sequence similarity is often remote even within well-established RNA families, we cannot rely on pure sequence alignment and related techniques for this task. Indeed, it has been shown that sequence alignments of structured RNAs fail at pairwise sequence identities below about 60% [5]. Therefore, we briefly review nucleotide database search methods that make use of sequence and structure information. There are general sequence-structure alignment tools, which determine structural similarities and derive consensus structure patterns for RNAs that are too diverse to be alignable at sequence level. We identify two classes of such tools. The first class, with *RNAforrester* [6] and *MARNA* [7] being the main representatives, require a known or predicted secondary structure for both sequences as input. However, they suffer from the low quality of secondary structure prediction, especially if the boundary of the RNA elements are not exactly known. The second class of methods are derivatives of the Sankoff algorithm [8], which provides a general solution to the problem of simultaneously computing an alignment and the common secondary structure of the two aligned sequences. Due to its high complexity ($\mathcal{O}\left(n^6\right)$ time and $\mathcal{O}\left(n^4\right)$ memory) several variants of this approach have been introduced such as *foldalign* [9,10], *dynalign* [11] and *LocaRNA* [12]. Still, these tools have a time complexity that is generally too high for a rapid database search. Thus, more specialized tools for searching RNA families in nucleotide databases have been introduced. Tools like *RNAMotif* [13], *RNAMOT* [14], *RNABOB* [15], *RNAMST* [16], *PatScan* [17], and *PatSearch* [18] are based on motif descriptors defining primary and secondary structure properties of the families to be searched for. They provide a language for defining descriptors and a method to search with these in large nucleotide databases. For these tools, the motif descriptor for a family has to be extracted externally from other information (such as a multiple sequence-structure alignment) about the specific RNA family. There are also tools that automatically derive descriptors from structure-annotated sequences or a multiple sequence alignment of related RNA sequences such as *Infernal* [19,20], *RSEARCH* [21], and *PHMMTS* [22]. They use variants of stochastic context-free grammars as descriptors, whereas *ERPIN* [23] uses sequential and structural profiles. Despite being fast compared to other methods, descriptor-based tools available today have a running time that is, in the best case, linear in the size of the target sequence database. This makes their application challenging when it comes to large sequence databases. A solution with sublinear running time would require index data structures. However, widely used index structures like suffix trees [24] or arrays [25] or the FM-index [26] perform badly on typical RNA sequence-structure patterns, because they cannot take advantage of the RNA structure information.

Here, we present a fast descriptor-based method and software for RNA sequence-structure pattern matching. The method consists of initially building an affix array [27], i.e. an index data structure of the target database. Affix arrays cope well with structural pattern constraints by allowing for an efficient matching order of the bases constituting the pattern. Structurally symmetric patterns like stem-loops can be matched inside out, such that first the loop region is matched and, in subsequent extensions, pairing positions on the boundaries are matched consecutively. Because the matched substring is extended to the left and to the right, this pattern matching scheme is known as bidirectional search. Unlike traditional left-to-right search where the two substrings constituting the stem region of the pattern are matched sequentially, in bidirectional search, base complementarity constraints are checked as early as possible. This leads to a significant reduction of the search space that has to be explored and in turn to a reduced running time. We note that bidirectional search for RNA sequence-structure patterns was also presented by Mauri et al. in [28]. However, their method uses affix trees [29] instead of the more memory efficient affix arrays. Affix trees require with approximately 45 bytes per input symbol more than twice the memory of affix arrays (18 bytes per input symbol), making their application infeasible on a large scale. Moreover, their method traverses the affix tree in a breadth-first manner, leading to a space requirement that grows exponentially with increasing reading depth. We instead employ a depth-first search algorithm whose space requirement is only proportional to the length of the searched substring.

The affix array directly supports the search for sequence-structure patterns that describe sequence-structure motifs with non-branching structure, for example stem-loops. In contrast, e.g. the search for stems closing a multi-loop is not directly supported. Nevertheless, even for RNA containing multi-loops, the affix array can still speed up the search. Our general approach for finding RNA families with branching structure is to describe each stem-loop substructure by a sequence-structure pattern. Each of these patterns is matched independently using the affix array. Then, with a new efficient chaining algorithm, we compute chains of matches such that the chained matches reflect the order of occurrence of the respective patterns in the molecule. Note that complex structures containing one or more multi-loops can be expected to contain sufficiently many non-branching patterns, such that the proposed chaining strategy identifies true matches with high specificity.

For a better understanding of the concepts underlying our method, we begin with formalizing RNA structural motifs. We then describe the concepts and ideas of affix arrays and show how to use them in an algorithm for fast bidirectional search for sequence-structure patterns. After presenting a detailed complexity analysis of the algorithm, we proceed with a detailed description and analysis of a novel method for computing chains of sequence-structure pattern matches. Finally, we benchmark and validate our method in several experiments.

Methods

Preliminaries

A *sequence* S of length $n = |S|$ over an alphabet $\mathcal{A}$ is a juxtaposition of n elements (*characters*) from the set $\mathcal{A}$. $S[i]$, $0 \leq i < n$ denotes the *character of* S *at position* i. Let ε denote the empty sequence, the only sequence of length 0. By $\mathcal{A}^n$ we denote the set of sequences of length $n \geq 0$ over $\mathcal{A}$. The set of all possible sequences over $\mathcal{A}$ including the empty sequence ε is denoted by $\mathcal{A}^*$.

For a sequence $S = S[0]S[1]\dots S[n-1]$ and $0 \leq i \leq j < n$, $S[i..j]$ denotes the *substring* $S[i]S[i+1]\dots S[j]$ of S. We denote the *reverse sequence* of S with $S^{-1} = S[n-1]S[n-2]\dots S[0]$. For $S = uv$, u and $v \in \mathcal{A}^*$, u is a *prefix* of S, and v is a *suffix* of S. The k–th suffix of S starts at position k, while the k–th prefix of S ends at k. Note that the 0-th suffix of S is S itself and that $S[0]$ is the 0-th prefix of S. The k–th *reverse prefix* of S is the k–th suffix of S^{-1}. For $0 \leq k < n$, S_k denotes the k–th suffix of S, and $S_k^{-1} = (S^{-1})_k$, denotes the k–th reverse prefix of S.

Let $\mathcal{A}$ denote the *RNA alphabet* $\{A, C, G, U\}$. Its characters code for the nucleotides adenine (A), cytosine (C), guanine (G), and uracil (U). In the following we fix a sequence S over the RNA alphabet $\mathcal{A}$. For stating the space requirements of our index structures, we assume that $|S| < 2^{32}$, such that sequence positions and lengths can be stored in 4 bytes.

RNA structural motifs

RNA molecules can form complex secondary structures consisting of different structural elements like stem-loops with or without bulges or internal loops. See Figure 1 for an overview of some secondary structure elements. Such elements are often important for the function of the molecule and are structurally conserved throughout evolution. The secondary structure is formed by Watson-

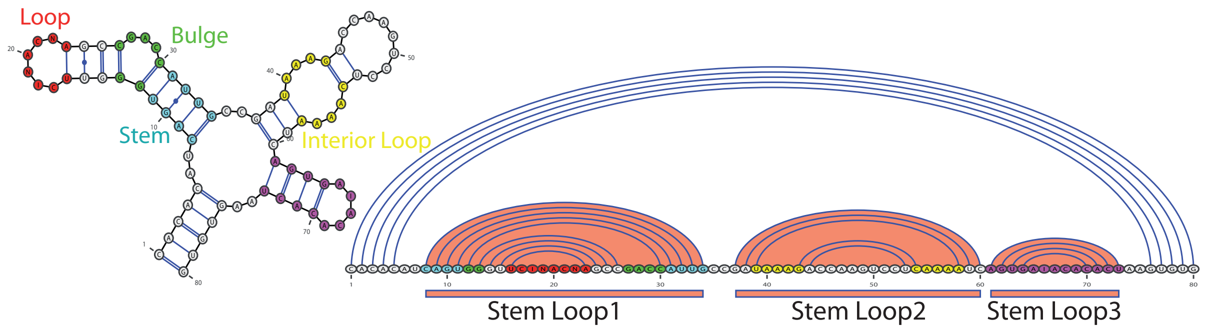


Figure 1: Secondary structure elements of an RNA molecule represented by a base-pair graph (left) and as arc-annotated sequence (right). The depicted structure contains three stem-loop substructures. Observe that all arcs representing base pairings are *non-crossing* and stem-loop substructures can contain interior loops and bulges. Hence this molecule forms a *non-crossing* secondary structure that does not contain higher order structural elements like pseudoknots. Secondary structure drawings were generated with the *VARNA* program [30].

Crick pairing of complementary bases and also by the slightly weaker wobble pairs. We say that two bases $(c, d) \in \mathcal{A} \times \mathcal{A}$ are *complementary* and can form a *base pair* if and only if $(c, d) \in \mathcal{C} = \{(A, U), (U, A), (C, G), (G, C), (G, U), (U, G)\}$. A *non-crossing RNA structure* R *of length* m is a set of *base pairs* (i, j), $0 \leq i < j < m$, stating that the base at position i pairs with the base at position j, such that for all $(i, j), (i', j') \in R$: $i < i' < j' < j$ or $i' < i < j < j'$ or $i < j < i' < j'$ or $i' < j' < i < j$. For the algorithms and methods presented in this paper we only consider this class of structures. For an example of such an RNA secondary structure see Figure 1. An important structural motif occurring in many RNA molecules is the *stem-loop* structure. We call R a *stem-loop* RNA structure if and only if for all $(i, j), (i', j') \in R : i < i' < j' < j$ or $i' < i < j < j'$. Note that due to our definition a stem-loop can contain bulges and interior loops (see Figure 1). We equivalently call such a structure *non-branching*. In Figure 1, such stem-loop structures occur as substructures.

A *structure string* H is a sequence over the alphabet $\{., (,)\}$ with an equal number of characters (and). There is a bijection between the set of (non-crossing) RNA structures R and the set of structure strings H, both of length m, such that for each base pair $(i, j) \in R$, $H[i] =$ (and $H[j] =$), and $H[r] =$. for positions r, $0 \leq r < m$, that do not occur in any base pair of R, i.e. $r \neq i \wedge r \neq j$ for all $(i, j) \in R$. Due to this equivalence we identify both representations.

Let $\Phi = \{$R, Y, M, K, W, S, B, D, H, V, N$\}$ be a set of characters. The IUPAC nucleotide base code introduces the characters in Φ to code nucleotide ambiguity and assigns a specific character class $\varphi(x) \subseteq \mathcal{A}$ to each $x \in \Phi \uplus \mathcal{A}$. In particular, for $x \in \mathcal{A} : \varphi(x) = \{x\}$ and $\varphi(\text{N}) = \mathcal{A}$. A *sequence pattern* is a sequence $P \in (\mathcal{A} \cup \Phi)^*$. Let m denote its length $|P|$. An *occurrence* of P in a sequence S is a position i, $0 \leq i < n$, such that $P[k] = S[i + k]$ with $S[i + k] \in \varphi(P[k])$ for all $0 \leq k < m$. An *RNA sequence-structure pattern (RSSP)* $\mathcal{Q} = (P, R)$ of length m is a pair of a *sequence pattern* P and a *structure string* R, both of length m. A *match* or *occurrence* of $\mathcal{Q}$ of length m in an RNA sequence S is an occurrence i of P in S, such that for all base pairs $(l, r) \in R$: $S[i + l]$ and $S[i + r]$ are complementary. Furthermore, define $\mathcal{CS}$ as a mapping of a character $c \in \Phi \cup \mathcal{A}$ to the set of its complementary characters in $\mathcal{A}$, i.e. $\mathcal{CS}(c) = \{d \in \mathcal{A} | \, \exists e \in \varphi(c) : d$ and e are complementary$\}$.

In this paper, structures described by RSSPs are non-branching.

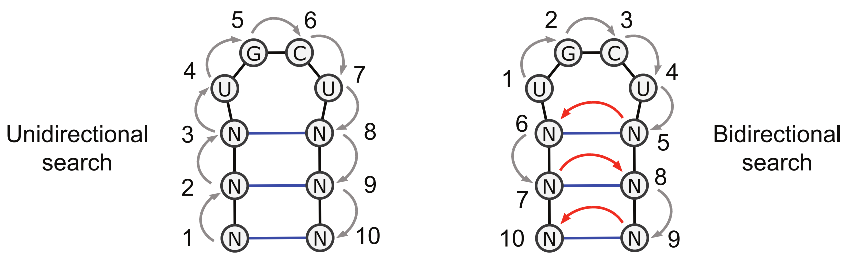


Figure 2: Unidirectional (left) and bidirectional (right) searches for the RNA sequence-structure pattern (RSSP) $\mathcal{Q} = (P, R)$ with P = NNNUGCUNNN and R = `(((....)))`, which represents a stem-loop structure of length $m = 10$. The numbers indicate the order in which the pattern characters are matched against the target sequence. In the unidirectional search, the characters are matched in a single direction, beginning (ending) with a character in $\varphi(P[0])$ ($\varphi(P[m-1])$). In the bidirectional search, the loop region of the pattern can be matched first. Then, pairing bases are matched consecutively by switching the search direction, represented by the red arrows.

The affix array data structure

In [27] the theoretical concept of an index data structure called *affix array* is described. This index structure supports efficient unidirectional as well as bidirectional searches and is more space efficient than the affix tree [29, 31]. The term *unidirectional search* refers to the search for occurrences of a sequence pattern where the pattern characters are compared with sequence characters in a left-to-right (right-to-left) order, i.e. the already compared (matched) prefix (suffix), of the pattern is extended to the right (left). Notably, a change of the direction is not possible.

When searching for occurrences of sequence-structure patterns, however, unidirectional search cannot exploit the complementarity condition on base paired pattern positions. To utilize this condition as effectively as possible, both positions of a base pair need to be accessed immediately after each other. This is enabled by *bidirectional search*, which refers to methods where the direction of the match extension can be changed freely. Figure 2 illustrates the order of the character comparisons of a sequence-structure pattern in the unidirectional and bidirectional searches.

Until now, affix arrays have received little attention in bioinformatics. Presumably, this has been due to the lack of an open and robust implementation. As a consequence, their potential for efficient database search with RSSPs has hardly been recognized and the details of this data structure are not widely known in the field. Therefore, we briefly recall the basic ideas of the affix array, which constitutes the central component of our *Structator* approach.

For notational convenience, we define $S^{\mathrm{F}} = S$ and $S^{\mathrm{R}} = S^{-1}$. We use S^X for statements that apply to S^{F} and S^{R}. The subscript X is used for other notions depending on S^{F} and S^{R} in an analogous way. Furthermore, we introduce the notation $\overline{\mathrm{F}} = \mathrm{R}$ and $\overline{\mathrm{R}} = \mathrm{F}$. We reserve a character $\$ \notin \mathcal{A}$, called *terminator symbol*, for marking the end of a sequence. $\$$ is lexicographically larger than all the characters in $\mathcal{A}$.

The affix array data structure of a sequence S is composed of six tables, namely $\mathsf{suf}_{\mathsf{F}}$ and $\mathsf{suf}_{\mathsf{R}}$, $\mathsf{lcp}_{\mathsf{F}}$ and $\mathsf{lcp}_{\mathsf{R}}$, and $\mathsf{aflk}_{\mathsf{F}}$ and $\mathsf{aflk}_{\mathsf{R}}$. They are called *suffix, longest common prefix*, and *affix link arrays* of S^{F} and S^{R}, respectively. Table $\mathsf{suf}_{\mathsf{R}}$ is also known as *reverse prefix array*. suf_X is an array of integers in the range 0 to n specifying the lexicographic order of the $n+1$ suffixes of the string $S^X\$$. That is, $S^X_{\mathsf{suf}_X[0]}, S^X_{\mathsf{suf}_X[1]}, \ldots, S^X_{\mathsf{suf}_X[n]}$ is the sequence of suffixes of $S^X\$$ in ascending lexicographic order. Each of the tables $\mathsf{suf}_{\mathsf{F}}$ and $\mathsf{suf}_{\mathsf{R}}$ requires $4n$ bytes and can be constructed in $\mathcal{O}(n)$ time and space [32]. In practice non-linear time [33, 34] construction algorithms are often used as they are faster and require less space.

lcp_X is a table in the range 0 to n such that $\mathsf{lcp}_X[0] = 0$, and $\mathsf{lcp}_X[i]$ is the length of the longest common prefix between $S^X_{\mathsf{suf}_X[i-1]}$ and $S^X_{\mathsf{suf}_X[i]}$ for $1 \leq i \leq n$. Each of the tables lcp_F and lcp_R requires n bytes and store entries with value up to 255, whereas occasional larger entries are stored in an exception table using 8 bytes per entry [35]. More space efficient representations of the lcp table are possible (see [36]). The construction of lcp_F and lcp_R can be accomplished in $\mathcal{O}(n)$ time and space given suf_F and suf_R [37]. In contrast to [27] where affix arrays were described using a terminology derived from tree-like data structures, we explain the underlying concepts of this data structure in terms of intervals in the suffix array suf_X. Two important concepts of affix arrays are suffix-intervals and lcp-intervals. An interval $[i..j]$ representing the set of suffixes $S^X_{\mathsf{suf}_X[i]}, ..., S^X_{\mathsf{suf}_X[j]}$, $0 \leq i \leq j \leq n$, of *width* $j-i+1$, is a *suffix-interval* in suf_X with *depth (prefix length)* $\ell \in \{0, \ldots, n\}$, or *$\ell$-suffix-interval,* denoted $\ell - [i..j]$, if and only if the following three conditions hold:

1. $\mathsf{lcp}_X[i] < \ell$;

2. $\mathsf{lcp}_X[j+1] < \ell$; and

3. $\mathsf{lcp}_X[k] \geq \ell$ for all $k \in \{i+1, \ldots, j\}$.

We call a suffix-interval $\ell - [i..j]$ in suf_X *lcp-interval* in suf_X with *lcp-value* $\ell \in \{0, \ldots, n\}$, or *$\ell$-interval,* if and only if $i < j$ and $\mathsf{lcp}_X[k] = \ell$ for at least one $k \in \{i+1, \ldots, j\}$.

For a suffix-interval $\ell - [i..j]$ in suf_X, we denote the common prefix of length ℓ of its suffixes $S^X_{\mathsf{suf}_X[i]}, \ldots, S^X_{\mathsf{suf}_X[j]}$ by $\delta_X(\ell - [i..j]) = S^X[\mathsf{suf}_X[i]..\mathsf{suf}_X[i] + \ell - 1]$. In case of an lcp-interval $\ell - [i..j]$ in suf_X, $\delta_X(\ell - [i..j])$ is the longest common prefix of all suffixes in this interval.

In summary, a suffix-interval $\ell - [i..j]$ in suf_X describes simultaneously:

- A location in the index structure suf_X by interval borders i and j and depth ℓ. For an example, see the yellow marked region in Figure 3 which corresponds to the suffix-interval $4 - [4..6]$ in suf_F.

- A (lexicographically ordered) sequence of suffixes $S^X_{\mathsf{suf}_X[i]}, \ldots, S^X_{\mathsf{suf}_X[j]}$. For an example, consider the lexicographically ordered sequence $S^\mathrm{F}_{\mathsf{suf}_\mathsf{F}[4]} = \mathrm{CUGCA}, \ldots, S^\mathrm{F}_{\mathsf{suf}_\mathsf{F}[6]} =$ CUGCUGCUGCA of suffixes in the suffix-interval $4 - [4..6]$ in suf_F in Figure 3.

- A substring of S^X of length ℓ, namely $\delta_X(\ell - [i..j])$. That is, for the suffix-interval $4 - [4..6]$ in suf_F in Figure 3, $\delta_\mathsf{F}(4 - [4..6]) =$ CUGC.

- The occurrences of this substring in S^X, namely at positions $\mathsf{suf}_X[i], \ldots, \mathsf{suf}_X[j]$. To give an example, consider Figure 3 and observe that substring CUGC occurs at positions $\mathsf{suf}_\mathsf{F}[4] = 10$, $\mathsf{suf}_\mathsf{F}[5] = 7$, and $\mathsf{suf}_\mathsf{F}[6] = 4$ in S^F = AUAGCUGCUGCUGCA.

For unidirectional left-to-right search of some pattern in S it is sufficient to process lcp-intervals only in suf_F. For bidirectional pattern search using affix arrays, described in detail in the next section, we employ information from table suf_F as well as suf_R. Therefore, we need to associate information of one table to the other. This is done by linking intervals via tables aflk_F and aflk_R. We observe that there exists a mapping between lcp-intervals in suf_F and suf_R. This is stated by the following proven lemma [27].

Lemma 1 *For every lcp-interval $q = \ell - [i..j]$ in table suf_X there is exactly one lcp-interval $q^{-1} = \ell' - [i'..j']$ in table $\mathsf{suf}_{\overline{X}}$ called reverse lcp-interval of q, such that $\ell' \geq \ell$ and the $\ell - 1$-th prefix of $\delta_{\overline{X}}(q^{-1})$ equals $(\delta_X(q))^{-1}$. The number of suffixes (prefixes) represented by q and q^{-1} are the same, i.e., $j - i = j' - i'$.*

We note that the equivalence $q = (q^{-1})^{-1}$ is not necessarily true. This is stated by the next lemma.

Lemma 2 *If the lcp-interval q^{-1} with depth ℓ' in $\mathsf{suf}_{\overline{X}}$ is the reverse of the lcp-interval q with depth ℓ in suf_X and $\ell = \ell'$, then $q = (q^{-1})^{-1}$. Otherwise, if $\ell' > \ell$, then $q \neq (q^{-1})^{-1}$.*

The mapping between intervals in S^{F} and S^{R} is encoded in tables $\mathsf{aflk_F}$ and $\mathsf{aflk_R}$ as follows. Tables $\mathsf{aflk_F}$ and $\mathsf{aflk_R}$ store, for each lcp-interval in $\mathsf{suf_F}$ and $\mathsf{suf_R}$ respectively, a pointer to the reverse interval in the reverse tables $\mathsf{suf}_{\overline{F}}$ and $\mathsf{suf}_{\overline{R}}$. The position in the tables where the pointers are stored is determined by the function home_X, defined as

$$\mathsf{home}_X([i..j]) = \begin{cases} i, & \text{if } \mathsf{lcp}_X[i] \geq \mathsf{lcp}_X[j+1], \\ j, & \text{otherwise}, \end{cases} \tag{1}$$

where $\ell - [i..j]$ is an lcp-interval in suf_X. Hence, the home position is one of two boundary positions. Strothmann [27] shows that $\mathsf{home}_X([i..j]) \neq \mathsf{home}_X([i'..j'])$ for different lcp-intervals $\ell - [i..j]$ and $\ell' - [i'..j']$.

Table aflk_X of string $S^X\$$ with total length $n+1$ can now be defined as a table in the range 0 to n such that $\mathsf{aflk}_X[\mathsf{home}_X(q)] = i'$, where q is an lcp-interval in suf_X and i' is the left border of the reverse interval $q^{-1} = [i'..j']$ in $\mathsf{suf}_{\overline{X}}$. We refer to the entries in table aflk_X as *affix links*. Tables $\mathsf{aflk_F}$ and $\mathsf{aflk_R}$ occupy $4n$ bytes each. They can be computed by traversing the lcp-intervals in suf_X while simultaneously looking for the corresponding reverse lcp-intervals in $\mathsf{suf}_{\overline{X}}$. Locating reverse lcp-intervals can be accelerated by skp-tables. These tables, introduced in Beckstette *et al.* [38] and hereinafter referred to as $\mathsf{skp_F}$ and $\mathsf{skp_R}$, can be constructed in linear time [39] and allow one to quickly skip intervals in suf_X (for details, see [38]). The construction of tables $\mathsf{aflk_F}$ and $\mathsf{aflk_R}$ takes $\mathcal{O}\left(n^2\right)$ time. Although the use of skp-tables requires additional $2 \times 4n$ bytes of memory, they considerably reduce the construction times of tables $\mathsf{aflk_R}$ and $\mathsf{aflk_R}$ in practice. We note that Strothmann [27] describes a linear time construction algorithm for tables $\mathsf{aflk_F}$ and $\mathsf{aflk_R}$, which employs suffix link and child-tables [35] and an additional table. Altogether these tables require together at least additional $7n$ bytes of space. Moreover, even without applying the skp-table based acceleration, Strothmann states that the quadratic time construction algorithm is fast in practice.

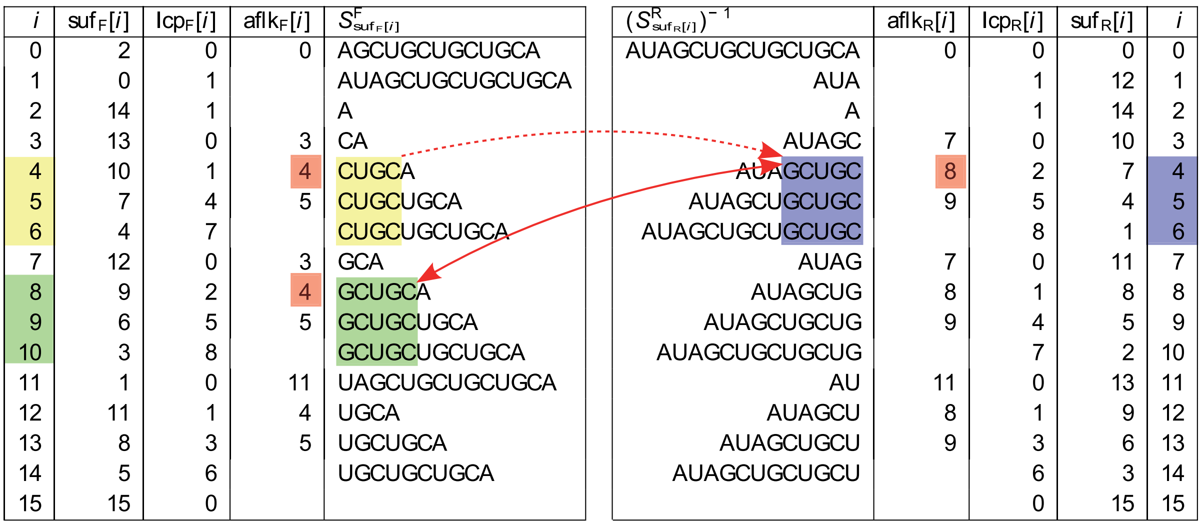
An example of the affix array for sequence S = AUAGCUGCUGCUGCA highlighted with some of its lcp-intervals connected to the respective reverse interval via the aflk_X table is shown in Figure 3.

Because affix links in table aflk_X are only defined for lcp-intervals but not suffix-intervals in general, which we require in bidirectional search, we introduce the concept of *affix-intervals*. Affix-intervals are similar to affix nodes as defined in [27]. An affix-interval in suf_X is a triple $v = \langle k, q, X \rangle$, where k is an integer designated *context* of v and q is a suffix-interval in suf_X.

An affix-interval $v = \langle k, q, X \rangle$ in suf_X, with $q = \ell - [i..j]$, $\ell > 0$, $-m < k < \ell$, describes a substring $\omega_X(v)$ of S^X of length $\ell - k$, defined as the k-th suffix of $\delta_X(q)$, i.e. $\omega_X(v) = S^X[\mathsf{suf}_X[i] + k..\mathsf{suf}_X[i] + \ell - 1]$. At the same time v identifies all occurrences of $\omega_X(v)$ in S^X, namely the positions $\mathsf{suf}_X[i] + k, \ldots, \mathsf{suf}_X[j] + k$. For $v = \langle k, q, X \rangle$, we therefore also use the notation $\overrightarrow{v} = \omega_F(v)$ if $X = \mathsf{F}$ and $\overrightarrow{v} = \omega_R(v)^{-1}$ if $X = \mathsf{R}$. As an example, consider the affix-interval $v = \langle 1, 4 - [4..6], \mathsf{F} \rangle$ in $\mathsf{suf_F}$ of the affix array shown in Figure 3. In this case, $k = 1$, $q = 4 - [4..6]$, and $X = \mathsf{F}$. v identifies all occurrences of substring $\overrightarrow{v}$ = UGC in S^{F} at positions $\mathsf{suf_F}[4] + 1 = 11$, $\mathsf{suf_F}[5] + 1 = 8$, and $\mathsf{suf_F}[6] + 1 = 5$. Observe that $\overrightarrow{v}$ = UGC is the first suffix of $\delta_{\mathsf{F}}(q)$ = CUGC due to context $k = 1$.

Searching RNA databases for RSSPs with affix arrays

Pattern matching using affix arrays means the sequential processing of characters in the pattern guiding the traversal of the data structure. This can be performed in either a traditional left-to-right order



i	$\mathsf{suf_F}[i]$	$\mathsf{lcp_F}[i]$	$\mathsf{aflk_F}[i]$	$S^{\mathsf{F}}_{\mathsf{suf_F}[i]}$
0	2	0	0	AGCUGCUGCUGCA
1	0	1		AUAGCUGCUGCUGCA
2	14	1		A
3	13	0	3	CA
4	10	1	4	CUGCA
5	7	4	5	CUGCUGCA
6	4	7		CUGCUGCUGCA
7	12	0	3	GCA
8	9	2	4	GCUGCA
9	6	5	5	GCUGCUGCA
10	3	8		GCUGCUGCUGCA
11	1	0	11	UAGCUGCUGCUGCA
12	11	1	4	UGCA
13	8	3	5	UGCUGCA
14	5	6		UGCUGCUGCA
15	15	0		

$(S^{\mathsf{R}}_{\mathsf{suf_R}[i]})^{-1}$	$\mathsf{aflk_R}[i]$	$\mathsf{lcp_R}[i]$	$\mathsf{suf_R}[i]$	i
AUAGCUGCUGCUGCA	0	0	0	0
AUA		1	12	1
A		1	14	2
AUAGC	7	0	10	3
AUAGCUGC	8	2	7	4
AUAGCUGCUGC	9	5	4	5
AUAGCUGCUGCUGC		8	1	6
AUAG	7	0	11	7
AUAGCUG	8	1	8	8
AUAGCUGCUG	9	4	5	9
AUAGCUGCUGCUG		7	2	10
AU	11	0	13	11
AUAGCU	8	1	9	12
AUAGCUGCU	9	3	6	13
AUAGCUGCUGCU		6	3	14
		0	15	15

Figure 3: Affix array for S = AUAGCUGCUGCUGCA. Some lcp-intervals are marked by rectangles and the affix links from an lcp-interval to its reverse interval are represented by arcs. The solid arc points in two directions, from the the lcp-interval $q = 5 - [8..10]$ in $\mathsf{suf_F}$ (on the left-hand side) to its reverse interval $q^{-1} = 5 - [4..6]$ in $\mathsf{suf_R}$ (on the right-hand side) and vice versa. That is, $q = (q^{-1})^{-1}$ (see Lemma 2). The dotted arc points in only one direction, from the lcp-interval $q = 4 - [4..6]$ in $\mathsf{suf_F}$ to its reverse interval $q^{-1} = 5 - [4..6]$ in $\mathsf{suf_R}$. In this case, the reverse of q^{-1} is $(q^{-1})^{-1} = 5 - [8..10]$, and $q \neq (q^{-1})^{-1}$.

resulting in a unidirectional search or in a bidirectional way where character comparison is started at any position of the pattern extending the already matched substring of the pattern to the left or to the right. We will see that bidirectional search using alternating series of left and right extensions is very well suited for fast database search with RNA sequence-structure patterns (RSSPs) containing both paired and unpaired bases. In the following we will explain the two different traversal strategies underlying unidirectional and bidirectional search using affix arrays.

Unidirectional traversal

Let $P = P[0] \dots P[m-1] \in (\mathcal{A} \cup \Phi)^m$ be a sequence pattern to be searched in S in a unidirectional left-to-right way using information from table $\mathsf{suf_F}$ only. To search for P, we call the procedure *unidir-search* of Figure 4 by *unidir-search*$([0..|S|], P, 0)$. Therefore, in step 0 we start searching for the characters in $\varphi(P[0])$ in the suffix-interval $q_0 = 0 - [0..n]$ in $\mathsf{suf_F}$, which represents all suffixes of $S\$$. In each step k, $k \geq 0$, we locate the $k+1$-suffix-intervals q_k of maximal width, such that $P[0..k-1]d$ matches $\delta_{\mathsf{F}}(q_k)$. For each $d \in \varphi(P[k])$, this step is performed by a binary search in the suffix-interval $q_{k-1} = \ell - [i..j]$ for $q_k = (\ell+1) - [i'..j']$, $i \leq i' \leq j' \leq j$, $j' - i'$ maximal, and $S[\mathsf{suf_F}[i'] + k] = d$.

After m steps, if all q_k could be located, $\delta_{\mathsf{F}}(q_m)$, $q_m = m - [r..s]$, matches the pattern P and the occurrences $\mathsf{suf_F}[r], \mathsf{suf_F}[r+1], \dots, \mathsf{suf_F}[s]$ of $\delta_{\mathsf{F}}(q_m)$ are reported as occurrences of P in S. Note that in this approach the matched substring of S is extended only to the right and at each step k the occurrences of the already matched prefix are represented by a suffix-interval.

Algorithm 1: *unidir-search*(suffix-interval $q = [i..j]$, pattern P, position k)

```
1 if $k = |P|$ then
2     report match at positions $\mathsf{suf}_\mathsf{F}[i], ..., \mathsf{suf}_\mathsf{F}[j]$
3     return
4 else
5     foreach $q'$ such that $d \in \varphi(P[k])$ and $\delta_\mathsf{F}((k+1) - [i..j]) = \delta_\mathsf{F}(k - [i..j])d$ do
6         unidir-search($q', P, k+1$)
7     end
8 end
```

Figure 4: Unidirectional search algorithm for searching for a sequence pattern $P \in (\mathcal{A} \cup \Phi)^*$. Given the suffix array suf_F of S, the procedure enumerates all occurrences of P in S when called by *unidir-search*$([0..|S|], P, 0)$. In line 5, the suffix-interval q' is located by binary search in $\mathcal{O}(\log n)$.

Bidirectional traversal

For the bidirectional search, we start at some position in $P \in (\mathcal{A} \cup \Phi)^m$ and then compare the pattern P character by character to the text, where we can freely switch between extending to the left or to the right. Note that as in the case of unidirectional search, ambiguous nucleotides x in the pattern can be handled by enumerating all characters c in the corresponding character class $\varphi(x)$. We can focus on the situation in the search, where

- a range $r..r'$ ($0 \leq r \leq r' < m$) of the pattern P is already compared,
- the occurrences of a substring $u \in \mathcal{A}^m$ of S matching $P[r..r']$ are represented by an affix-interval $v = \langle k, \ell - [i..j], X\rangle$ in suf_X, and
- we want to extend $\overrightarrow{v}$ either to the left or to the right by a sequence character $c \in \mathcal{A}$ (that matches the respective pattern character $P[r-1]$ or $P[r'+1]$). This will result in a new, extended affix-interval v_x.

Switch of the search direction. Like its suffix-interval, an affix-interval directly supports extension of the represented substring in only one direction, namely searching to the left for $X = \mathsf{F}$ and to the right for $X = \mathsf{R}$. However, there are "corresponding" affix-intervals representing the same substring of S but allowing extension to the opposite direction.

If the new search direction differs from the supported search direction of v, this *switch of the search direction* requires determining the corresponding affix-interval v' in $\mathsf{suf}_{\overline{X}}$ unless $i = j$ or v has non-empty context $k \neq 0$. There are these two exceptions, since first if $i = j$, independently of the value of k, $\omega_X(v)$ is already a unique substring of S^X. Second, for a non-empty context $k \neq 0$, all occurrences of substring $\omega_X(v)$ in S^X are followed (if $k > 0$) or preceded (if $k < 0$) by the same substring $u \in \mathcal{A}^k$.

Let $k = 0$ and $i < j$. The affix-interval $v' = \langle k', \ell' - [i'..j'], \overline{X}\rangle$ in $\mathsf{suf}_{\overline{X}}$ is called the *reverse affix-interval* of $v = \langle k, \ell - [i..j], X\rangle$ if and only if $j' - i' = j - i$, $\ell' \geq \ell$, and $\omega_X(v)^{-1} = \omega_{\overline{X}}(v')$. The interval boundaries i' and j' of v' are determined via a lookup in table aflk_X. We set $i' = \mathsf{aflk}_X[\mathsf{home}_X([i..j])]$ and $j' = i' + (j - i)$. Observe that ℓ is not necessarily the length of the longest common prefix of all suffixes in $[i..j]$. For this reason we define $\ell_{\text{lcp}} = min\{\mathsf{lcp}_X[k] \mid i < k \leq j\} \geq \ell$ and compute the context of v' as $k' = \ell_{\text{lcp}} - \ell$. Further, we set $\ell' = \ell_{\text{lcp}}$. Hence the reverse affix-interval $v' = \langle k', \ell' - [i'..j'], \overline{X}\rangle$ is well defined and v' is the required corresponding interval of v.

Right/left c-extension of an affix-interval In our situation, $\overrightarrow{v} = u$ represents the occurrences of a substring u of S matching $P[r..r']$.

The *right (left) extension of v by a character $c \in \mathcal{A}$*, also called *c-extension of v*, is an operation that computes the affix-interval v_x representing all occurrences of a substring uc (cu). It fails, if there is no such substring. We elaborate the cases for right extension. The cases for left extension are symmetric and therefore omitted. For right c-extension of $v = \langle k, \ell - [i..j], X\rangle$, we determine the interval $v_x = \langle k_x, \ell_x - [i_x..j_x], X_x\rangle$ with $\overrightarrow{v_x} = \overrightarrow{v}c$. The first two cases do not require switching the search direction.

- Case $X = \mathsf{F}$ and $i = j$. u is a unique substring $\overrightarrow{v}$ of S. If $S[\mathsf{suf_F}[i] + \ell] = c$, then $v_x = \langle k, (\ell + 1) - [i..j], \mathsf{F}\rangle$.
- Case $X = \mathsf{F}$ and $i < j$. We determine the minimal $i_x \geq i$ and maximal $j_x \leq j$ in $\mathsf{suf_F}$ such that $S[\mathsf{suf_F}[i_x] + \ell] = c$ and $S[\mathsf{suf_F}[j_x] + \ell] = c$ by binary search in the suffix-interval $\ell - [i..j]$. If i_x and j_x exist, we set $v_x = \langle k, (\ell + 1) - [i_x..j_x], \mathsf{F}\rangle$.

The following cases require switching the search direction.

- Case $X = \mathsf{R}$, $i = j$. We evaluate $S^{\mathsf{R}}[\mathsf{suf_R}[i] + k - 1]$. If $S^{\mathsf{R}}[\mathsf{suf_R}[i] + k - 1] = c$, set $v_x = \langle k - 1, \ell - [i..j], \mathsf{R}\rangle$.
- Case $X = \mathsf{R}$, $i < j$, and $k = 0$. We first determine the reverse affix-interval $v' = \langle k', \ell' - [i'..j'], \mathsf{F}\rangle$ of v via a switch of the search direction as described above. Then we compute the minimal $i_x \geq i'$ and maximal $j_x \leq j'$ via binary search, such that $S[\mathsf{suf_F}[i_x] + \ell'] = c$ and $S[\mathsf{suf_F}[j_x] + \ell'] = c$. If i_x and j_x exist, we set $v_x = \langle k', (\ell' + 1) - [i_x..j_x], \mathsf{F}\rangle$.
- Case $X = \mathsf{R}$, $i < j$, and $k > 0$. We evaluate the $(k - 1)$–th character of $\delta_{\mathsf{R}}(\ell - [i..j])$. That is, if $\delta_{\mathsf{R}}(\ell - [i..j])[k - 1] = c$, then we consume the context k by setting $v_x = \langle k - 1, \ell - [i..j], \mathsf{R}\rangle$.

The operation fails if v_x cannot be determined.

RSSP matching using affix arrays

Searching a sequence S with an RNA sequence-structure pattern (RSSP) $\mathcal{Q} = (P, R)$ means to find the occurrences of P in S under the complementarity constraints imposed by the structure string R (cf. our definition of RSSP-occurrence). We introduce a search algorithm that checks for complementarity constraints as early as possible in bidirectional search to maximally reduce the search time due to this restriction.

For further considerations, we will assume a special 'canonical' form for RSSPs, which we define in the following. Independently of a sequence S, each RSSP describes a set of pattern instances, i.e. the set of potential subsequences matching the pattern. Often, there are several patterns that describe the same set of instances. For example, the pattern (UNUACACGNR, `((((....))))`) describes the same set of instances as (UNUACACGNR, `(((......)))`) since the additional base pair $(2, 7)$ in `((((....))))` does not make the pattern more specific. We will define a pattern to be structure minimal if there is no, in this sense, equivalent pattern containing a true subset of the base pairs. An RSSP $\mathcal{Q} = (P, R)$ is *structure minimal* if and only if for all base pairs $(i, j) \in R$ it holds that

$$\begin{aligned} &\varphi(P[i]) \cap \mathcal{CS}(P[j]) \times \varphi(P[j]) \cap \mathcal{CS}(P[i]) \\ &\neq \varphi(d) \times \varphi(e), \text{ for all } d, e \in (\mathcal{A} \cup \Phi). \end{aligned}$$

Furthermore, a general pattern is called *inconsistent* if it does not have any instance. Formally, a pattern is *consistent* if and only if for each base pair (i, j) it holds that $\varphi(P[i]) \cap \mathcal{CS}(P[j]) \neq \emptyset$ and $\varphi(P[j]) \cap \mathcal{CS}(P[i]) \neq \emptyset$. An example of an inconsistent RSSP is $\mathcal{Q} = (P, R)$ with P = UAUACACGAN and

$R =$ `((......))`. $\mathcal{Q}$ is not consistent because there is a base pair $(1, 8) \in R$ but the bases $P[1] = \text{A}$ and $P[8] = \text{A}$ are not complementary. An example of a structure minimal and consistent RSSP is (UNUACACGNR, `((......))`). Note that a pattern can be transformed into an equivalent structure minimal pattern and checked for consistency in $\mathcal{O}(m)$ time. For complexity considerations, we can therefore safely assume that patterns are consistent and structure minimal.

In this case, one can restrict the search space by comparing the two positions of each base pair immediately after each other. Due to this, the enumeration of characters matching the pattern symbols at each base pair can be restricted to the smaller number of complementary ones. In the search for a sequence-structure pattern this can reduce the number of enumerated combinations of matching characters exponentially. Thus, for structure minimal patterns (P, R), the non-branching structure R suggests a search strategy, i.e. an order of left and right extensions, which requires switching the search direction at every base pair but makes optimal use of the complementarity constraints due to the base pairs.

Following this idea, Mauri and Pavesi [28] presented an algorithm for matching RNA stem-loop structures using affix trees. This algorithm explores the search space in a breadth-first manner, so memory use grows exponentially with increasing depth. Instead of an affix tree, we employ the more space efficient affix array data structure and use a depth-first search algorithm which only requires space for the search proportional to the length of the substring searched. The depth-first search for all occurrences of a stem-loop RSSP $\mathcal{Q} = (P, R)$ is performed by calling procedure *bidir-search* of Algorithm 2 (see Figure 5). Note that we explicitly support bulges and internal loops in the stem-loop pattern, i.e. we do not require perfect stacking of the base pairs but allow general non-branching structures.

In our algorithm, we switch the search direction only once per base pair when matching the stem region of the pattern, thus halving the number of lookups in the affix link tables compared to a naive algorithm without this optimization. This was also observed by Strothmann [27] whose algorithm did not support RSSPs containing bulges and internal loops.

To match $\mathcal{Q}$ we call procedure *bidir-search* initially as *bidir-search*$(\langle 0, 0 - [0..n], \mathsf{F}\rangle, r_0 - 1, r_0)$, where $\langle 0, 0 - [0..n], \mathsf{F}\rangle$ is an affix-interval and r_0 is any position in the loop region of the RSSP or any position of a completely unpaired pattern. Then, the procedure traverses the affix-intervals by performing right and left extensions, while at the same time checking base complementarity of paired positions. This verification takes constant time by using a binary table of size $|\mathcal{A}| \times |\mathcal{A}|$ containing all valid base pairings. Matching positions are reported whenever the boundaries of the RSSP are reached.

In principle, we are free to choose any loop position r_0 (or any position if R is empty) for starting our bidirectional search algorithm. However, in order to reduce the combinatorial explosion of the search space due to ambiguous IUPAC characters, it is preferable to match non-ambiguous pattern characters first. To keep the selection simple, we set r_0 to the position of the first character c in the possible range such that $|\varphi(c)|$ is minimal. That is, we start the search with the most specific (least ambiguous) character.

A detailed example of bidirectional RSSP search along with the underlying affix array traversal is provided in Additional file 1, Section S1. We remark that procedure *bidir-search* can be extended to support variable-length RSSPs. Such an extended version of *bidir-search* is provided in Additional file 1, Section S3.

Analysis

We analyze the complexity for searching in a sequence S of length n for an RSSP $\mathcal{Q}$ of length $m < n$, where the index structures for S are already computed.

The bidirectional search algorithm requires tables suf_F and suf_R, lcp_F and lcp_R, and aflk_F and aflk_R.

Algorithm 2: $\textit{bidir-search}$(affix-interval $v = \langle k, \ell - [i..j], X\rangle$, pos r, pos r')

```
if $r < 0$ and $r' \geq m$ then
    report match at positions $\mathsf{suf}_X[i] + k, ..., \mathsf{suf}_X[j] + k$
    return
else if $r \geq 0$ and $r' < m$  and  $R[r] =$ '(' and $R[r'] =$ ')' then
    if $X = \mathrm{R}$ then
        // perform left extension first
        foreach $v'$ such that $d \in \varphi(P[r])$ and $\overrightarrow{v}' = d\overrightarrow{v}$ do
            foreach $v''$ such that $e \in \varphi(P[r'])$ and $(d, e)$ complementary and $\overrightarrow{v}'' = \overrightarrow{v}'e$ do
                bidir-search($v''$, $r - 1$, $r' + 1$)
            end
        end
    else
        // perform right extension first
        foreach $v'$ such that $e \in \varphi(P[r'])$ and $\overrightarrow{v}' = \overrightarrow{v}e$ do
            foreach $v''$ such that $d \in \varphi(P[r])$ and $(d, e)$ complementary and $\overrightarrow{v}'' = d\overrightarrow{v}'$ do
                bidir-search($v''$, $r - 1$, $r' + 1$)
            end
        end
    end
else if $r' < m$  and  $R[r'] =$ '.' and ($X = \mathrm{F}$ or $r < 0$ or $R[r] \neq$ '.') then
    foreach $v'$ such that $d \in \varphi(P[r'])$ and $\overrightarrow{v}' = \overrightarrow{v}d$ do
        bidir-search($v'$, $r$, $r' + 1$)
    end
else if $r \geq 0$  and  $R[r] =$ '.' then
    foreach $v'$ such that $d \in \varphi(P[r])$ and $\overrightarrow{v}' = d\overrightarrow{v}$ do
        bidir-search($v'$, $r - 1$, $r'$)
    end
end
```

Figure 5: Bidirectional recursive RSSP matching using an affix array. Procedure *bidir-search* finds all matches of a given RSSP (P, R), beginning the pattern extensions from any position in the loop region or any position in a completely unpaired pattern. In each call, parameter v denotes the affix-interval representing matches of the pattern substring $P[r+1..r'-1]$, $0 \leq r \leq r' < m$ satisfying the structural constraints imposed by $R[r+1..r'-1]$. The procedure takes care to change the search direction only as often as necessary, in particular it changes the direction only once per base pair.

Under our assumption that $n < 2^{32}$, each of the four tables suf_X and aflk_X consumes $4n$ bytes, and the two tables lcp_X are each stored in n bytes ($X \in \{\mathsf{F}, \mathsf{R}\}$). This amounts to a space consumption of $18n$ bytes for the index structures. The algorithm performs a depth first search, where the depth is limited by m, and therefore requires $O(m)$ space. The total space complexity is therefore $O(n)$.

We assume that $\mathcal{Q} = (P, R)$ is structure minimal. Such a pattern $\mathcal{Q}$ without ambiguity, i.e. $P \in \mathcal{A}^m$, does not contain base pairs and the search for $\mathcal{Q}$ does not profit from bidirectional search. Although such a pattern is processed by Algorithm 2, it can be handled by Algorithm 1 using only a suffix array and saving some overhead.

Algorithm 1 accomplishes the search for a non-ambiguous pattern $\mathcal{Q}$ on the suffix array suf_F using binary search for locating intervals in $\mathcal{O}(m \log n + z)$ time, where z is the number of occurrences of P in S. We remark that this time bound can be lowered at the price of higher memory consumption to $\mathcal{O}(m + \log n + z)$ [25] or even $\mathcal{O}(m + z)$ [35,40] time by using additional precomputed information.

Notably, if there is ambiguity but no base pair in $\mathcal{Q}$, bidirectional search can still be beneficial in practice. This is the case when searching for a pattern in which a string of unambiguous characters is surrounded on both sides by ambiguous IUPAC characters, because the comparison can start at the most specific part of the pattern. The time complexities for searching ambiguous patterns with Algorithm 1 can be estimated as $\mathcal{O}(n \log n)$ in the worst case of searching for the sequence pattern P consisting only of Ns. Furthermore, note that our Algorithm 2 behaves exactly like Algorithm 1 on patterns without base pairs if we invoke the search procedure with $r = -1$ and $r' = 0$.

For a pattern $\mathcal{Q} = (P, R)$ of length m, let $p \geq 0$ be the number of base pairs in R. In the worst case P consists only of Ns. Moreover, all possible strings of length m satisfying the complementarity constraints specified in R occur in the text S. Recall that, since we allow (G, U) pairs, there are $|\mathcal{C}|$=6 possible complementary base pairs. Thus, there are $|\mathcal{A}|^{m-2p}|\mathcal{C}|^p$ such strings and Algorithm 2 spans a virtual tree with $E_{m,p} = |\mathcal{A}|^{m-2p}|\mathcal{C}|^p$ paths from the root to a leaf. At each leaf, it reports the occurrences of the respective matched substring.

On each path from the root to the leaf the algorithm performs $m - 2p$ c-extensions and at most one switch of the search direction for matching the $m - 2p$ unpaired characters. Then, it performs $2p$ c-extensions and p switches of the direction for matching the base paired positions. Therefore, we count the total number of c-extensions as

$$\begin{aligned} &\sum_{i=1}^{m-2p} |\mathcal{A}|^i \; + \; |\mathcal{A}|^{m-2p} \sum_{j=1}^{2p} 2|\mathcal{C}|^j \\ &= \frac{|\mathcal{A}|^{m-2p+1} - |\mathcal{A}|}{|\mathcal{A}| - 1} + 2|\mathcal{A}|^{m-2p} \frac{|\mathcal{C}|^{p+1} - |\mathcal{C}|}{|\mathcal{C}| - 1}, \end{aligned}$$

which is in $\mathcal{O}(E_{m,p})$.

The cost of each c-extension consists of the cost of locating the suffix-interval of the new affix-interval, which is performed by binary search in $\mathcal{O}(\log n)$, and the cost for potentially computing the reverse affix-interval when switching the search direction.

Instead of performing the binary search over the suffix tables, one can use the child-tables introduced by Abouelhoda *et al.* in [35] to determine the child intervals and switch the search direction in constant time. The child-tables, however, add at least $2n$ bytes to the index and require additional involved index construction. As the child-tables improve the worst case behavior but, on the other hand, require more space, we analyze the complexity with and without these tables (i.e. with tables suf_X, lcp_X, and aflk_X only).

First, we analyze the time required for performing a single switch of the search direction. Therefore we assume that the current affix-interval is $v = \langle k, \ell - [i..j], X \rangle$. Consider the following two cases.

(1) Case $i = j$ or $k \neq 0$. If $i = j$, $\overrightarrow{v}$ represents a unique substring of S, or, if $k \neq 0$, all occurrences of substring $\overrightarrow{v}$ in S are followed (if $k > 0$) or preceded (if $k < 0$) by the same substring of length $|k|$ (known as context). Switching the search direction does not require locating the reverse interval of v, because the algorithm can perform the c-extension in the new search direction by consuming context. Therefore, this case requires constant time.

(2) Case $i < j$ and $k = 0$. The algorithm needs to locate the reverse affix-interval $v' = \langle k', \ell' - [i'..j'], \overline{X} \rangle$ of v. Interval boundaries $i' = \mathsf{aflk}_X[\mathsf{home}_X([i..j])]$ and $j' = i' + (j - i)$ of v' are computed in constant time.

   By definition, computing the reverse affix-interval of v requires knowing ℓ_{lcp}. Then, $\ell' = \ell_{\text{lcp}}$ and $k' = \ell' - \ell$. Without child-tables, we determine ℓ_{lcp} by computing the length of the longest common prefix between $S^X_{\mathsf{suf}_X[i]}$ and $S^X_{\mathsf{suf}_X[j]}$. It suffices to perform $\ell_{\text{lcp}} - \ell + 1 = k' + 1$ character comparisons only, since both suffixes $S^X_{\mathsf{suf}_X[i]}$ and $S^X_{\mathsf{suf}_X[j]}$ share a common prefix of at least length ℓ. With the help of child-tables, ℓ_{lcp} is determined in constant time [35].

Due to the following lemma, the computation of all reverse affix-intervals on one path of our virtual tree is in $\mathcal{O}(n)$ if child-tables are not used.

Lemma 3 *Using tables* suf_X, lcp_X, *and* aflk_X, *the computation of all contexts on a path in the recursion of Algorithm 2 is in* $\mathcal{O}(n)$.

Proof. Let $v_1, v_2, v_t \ldots, v_C$ be the sequence of reverse intervals processed when matching $\mathcal{Q}$, and let k_t denote the context of v_t for $1 \leq t \leq C$.

To show $\sum_{t=1}^{C} k_t \leq n$, let $v = \langle k, \ell - [i..j], X \rangle$, with $k = 0$, $i < j$, and $X = \mathsf{F}$ ($X = \mathsf{R}$), be the current affix-interval. We assume without loss of generality that we perform a left (right) c-extension of v and thus locate the reverse interval $v_t = \langle k_t, \ell_t - [i_t..j_t], \overline{X} \rangle$. Then the following statements hold: $k_t \geq 0$, $\ell_t = \ell + k_t$, and $j_t - i_t = j - i$ (see Lemma 1). Observe that $k_t = 0$ implies $\omega_{\overline{X}}(v_t) = \delta_{\overline{X}}(\ell_t - [i_t..j_t])$ and $k_t > 0$ implies that substring $\delta_{\overline{X}}(\ell_t - [i_t..j_t])$ has a non-empty prefix of length k_t, namely $S^{\overline{X}}[\mathsf{suf}_{\overline{X}}[i_t]..\mathsf{suf}_{\overline{X}}[i_t] + k_t - 1]$. Note that v_t is only located if $k = 0$, otherwise the context k has to be consumed. Hence there is no reverse interval $v_s = \langle k_s, \ell_s - [i_s..j_s], \overline{X} \rangle$, with $1 \leq s \leq C$, $s \neq t$, and $k_s > 0$, such that the $(k_s - 1)$-th prefix of $\delta_{\overline{X}}(\ell_s - [i_s..j_s])$ overlaps with $S^{\overline{X}}[\mathsf{suf}_{\overline{X}}[i_t]..\mathsf{suf}_{\overline{X}}[i_t] + k_t - 1]$ for the same positions in $S^{\overline{X}}$. From this, $\sum_{t=1}^{C} k_t \leq n$ follows. Since a single context k_t can be determined by performing exactly $k_t + 1$ character comparisons, this implies $\mathcal{O}(n)$ time to compute all these contexts. With this, we conclude that all switches of the search direction performed while finding one substring w in S that matches $\mathcal{Q}$ take up to $\mathcal{O}(n)$ time. □

Therefore, when searching for $\mathcal{Q}$ without child-tables, the total time for switching search directions is coarsely estimated by multiplying the complexity for one path with the number of paths as $\mathcal{O}(E_{m,p} n)$. The use of child-tables removes the linear factor.

For the worst case that all strings matching the pattern actually occur as substrings in S, the sequence S must have a certain minimal length. In the case of $p = 0$, the possible matches are the words in $\mathcal{A}^m$ and a sequence that contains all these matches is called $|\mathcal{A}|$-ary *de Bruijn* sequence of order m [41] without wrap-around, i.e. a *de Bruijn* sequence with its first $m - 1$ characters concatenated to its end. Such a sequence was shown to have a length of $n_0 = |\mathcal{A}|^m + m - 1$. As a consequence, the worst case requires $n \geq n_0$.

We summarize the worst-case time complexities for Algorithm 2 as follows. 1.) From determining new suffix-intervals, we get a contribution of $\mathcal{O}(E_{m,p} \log n)$. For $n \geq n_0$, this is in $\mathcal{O}(n \log n)$. Child-tables reduce this time further to $\mathcal{O}(n)$. 2.) Switching directions without child-tables is in $\mathcal{O}(E_{m,p} n)$

worst-case time, which is reduced to $\mathcal{O}(E_{m,p})$ when using child-tables. For $n \geq n_0$, $E_{m,p}$ is in $\mathcal{O}(n)$. Finally, Algorithm 2 runs in $\mathcal{O}(E_{m,p}(n + \log n))$, which is reduced to $\mathcal{O}(E_{m,p})$ using child-tables (i.e. $\mathcal{O}(n)$ for $n \geq n_0$).

One should note that the worst-case time complexity of bidirectional search for sequence-structure pattern is only in the order of online search algorithms. In our implementation, we use a minimal set of tables in order to keep the implementation simple and save space.

However, it can be clearly seen from this analysis that the worst case is based on extremely pessimistic assumptions that are almost contrary to the expected application. 1.) It is assumed that a pattern consists of wildcards N only. In the expected application, however, patterns will often specify bases in the loop region, which is of particular benefit for our algorithm. 2.) Sequences, like the *de Bruijn* sequence, that contain all possible matches of an average sized pattern will be rare in practice. E.g. it could be assumed that a sequence that contains all possible matches of a pattern Q with p base pairs (and $P =$N. . . N) is at least as long as the $|\mathcal{A}|$-ary *de Bruijn* sequence of order m, since one expects no significant bias for the specific complementarity due to R over all substrings of length m. However, $E_{m,p} = |\mathcal{A}|^{m-p}|\mathcal{C}|^p = 4^{m-2p}6^p = 4^m/(16/6)^p$ is even for small p much smaller than $n_0 = 4^m + m - 1$. For example, four base pairs (i.e., $p = 4$) reduce the time bound by a factor of $(16/6)^4 \approx 50$ and eight base pairs reduce time by a factor of about 2500.

RNA secondary structure descriptors based on multiple ordered RSSPs

Obviously RNAs with complex, branching structures cannot be described completely by a single RSSP. Describing an RNA by only a single unbranched fragment is often inappropriate, since searching a large sequence database or a complete genome for structurally conserved RNAs (RNA homology search) with a single RSSP will likely generate many spurious matches. However, larger RNAs can often adequately be described by a sequence of RSSPs. This holds for $1{,}247$ out of $1{,}446$ RNA families in Rfam 10.0 which have a structure containing several stem-loops but no multi-loop. Only 199 out of $1{,}446$ (13.76%) RNA families in Rfam 10.0 containing multi-loops cannot be modeled completely this way. Still, the consensus structures of these 199 families contain on average 4.06 stem-loops (standard deviation 2.08, median 3) which can be modeled as RSSPs. In consequence, we can use a sequence of RSSPs that consist of at least one pattern per stem-loop (and potentially also unstructured patterns) for the description of those families. This allows to accurately identify members even of those families containing multi-loops.

We address search for complex structured RNA families with the new concept of RNA secondary structure descriptors (SSD for short). SSDs use the information of multiple ordered RSSPs derived from the decomposition of an RNA's secondary structure or from the consensus secondary structure of a multiple sequence-structure alignment of related RNAs into stem-loop-like structural elements. Such consensus secondary structures for multiple RNAs can be computed with a variety of programs following one of the three strategies introduced in [42]. Namely: (A) alignment of the sequences followed by joint folding [43–46], (B) Sankoff style [8] simultaneous alignment and folding [10, 12, 47, 48], and (C) individual folding of the sequences followed by alignment of their structures [7, 49, 50]. In the following we make the concept of SSDs more precise. Let $A = A_1, A_2, \ldots, A_L$ be a sequence of non-overlapping alignment blocks. These alignment blocks are excised from a multiple sequence(-structure) alignment and represent regions of the molecule that fold into stem-loop-like structures or remain unfolded. The indexing from 1 to L reflects their order of occurrence in the alignment. Hence A represents a sequential decomposition of the molecule's secondary structure (in $5' \rightarrow 3'$ direction) into regions, each of which can be described by an RSSP. See Figure 6 (A) for an example.

An SSD $\mathcal{R}$ of length L is a sequence of L RSSPs $\mathcal{R} = \mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_L$ where $\mathcal{Q}_i$ denotes the RSSP

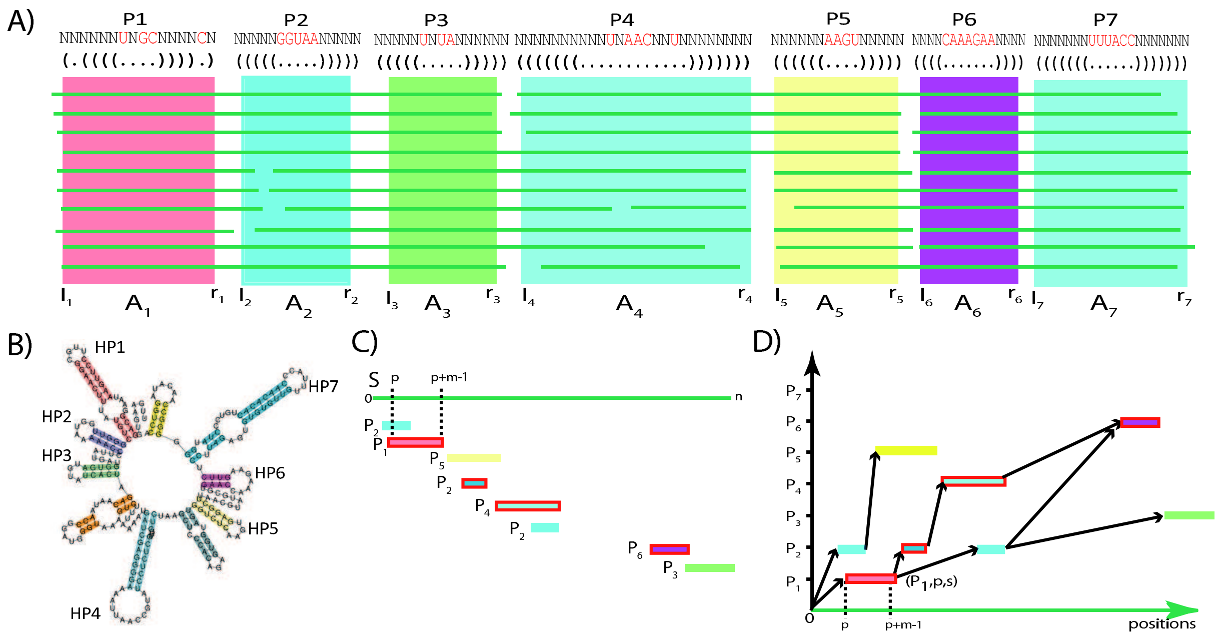

A)
P1
P2
P3
P4
P5
P6
P7
l_1 A_1 r_1 l_2 A_2 r_2 l_3 A_3 r_3 l_4 A_4 r_4 l_5 A_5 r_5 l_6 A_6 r_6 l_7 A_7 r_7
B)
HP1
HP2
HP3
HP4
HP5
HP6
HP7
C)
S p p+m-1
0 n
P_2 P_1 P_5 P_2 P_4 P_2 P_6 P_3
D)
P_7 P_6 P_5 P_4 P_3 P_2 P_1
(P_1,p,s)
0 p p+m-1
positions

Figure 6: (A) Non-overlapping alignment blocks of stem-loop regions excised from a multiple sequence-structure alignment and derived sequence-structure patterns. Since $l_i \le r_i < l_j \le r_j$ and sequence regions $S[l_i \dots r_i]$ fold into stem-loop structures for $1 \le i \le j \le 7$, $A = A_1, A_2, A_3, A_4, A_5, A_6, A_7$ is an ordered sequence of non-overlapping alignment blocks suitable to construct an RNA secondary structure descriptor $\mathcal{R} = \mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4, \mathcal{Q}_5, \mathcal{Q}_6, \mathcal{Q}_7$. The sequence-structure patterns $\mathcal{Q}_i, i \in [1, 7]$ of $\mathcal{R}$ given on top of their underlying alignment blocks describe the seven marked stem-loops shown in the RNA secondary structure (B) of the Citrus tristeza virus replication signal (Rfam: RF00193). (C) Matches of RSSPs $\mathcal{Q}_i, i \in [1, 7]$, on sequence S, sorted in ascending order of their start position. (D) Graph-based representation of the matches of $\mathcal{Q}_i, i \in [1, 7]$. An optimal chain of collinear non-overlapping matches is determined by computing an optimal path in the directed acyclic graph. Observe that not all edges in the graph are shown in this example and that the optimal chain (indicated here by their red marked members) is not necessarily the longest possible chain.

describing A_i, $i \in [1, L]$. The order $\ll$ of the RSSPs in $\mathcal{R}$ is imposed by the order of the corresponding alignment blocks. By l_i and r_i we denote the start and end positions of A_i in the multiple alignment, respectively. In practice, $\mathcal{R}$ can be obtained from multiple sequence-structure alignments of related RNA sequences (i.e., of an RNA family) as they are available in databases like Rfam [3,4]. A match to $\mathcal{R}$ is a non-overlapping sequence of matches for some or all of the RSSPs in $\mathcal{R}$ in their specified order. We will now make this more precise.

Consider an RNA SSD $\mathcal{R}$ with total order $\ll$. Let $\mathcal{MS}$ be the set of all matches for all RSSP from $\mathcal{R}$ in sequence S of length n. A match is represented by a pair $(\mathcal{Q}, p)$ such that $\mathcal{Q}$ matches at position p in S. With each $\mathcal{Q}$ in $\mathcal{R}$ we associate a positive weight $\alpha(\mathcal{Q})$ which can be defined by the user. This weight allows to quantify the expressiveness of $\mathcal{Q}$ and/or its significance. For example, $\alpha(\mathcal{Q})$ can be the length of $\mathcal{Q}$ or it might be derived from the number of non-ambiguous nucleotides in $\mathcal{Q}$ or the probability of obtaining a match for $\mathcal{Q}$ just by chance assuming a certain (mono-)nucleotide background distribution.

We say that matches $(\mathcal{Q}, p)$ and $(\mathcal{Q}', p')$ are *collinear*, written as $(\mathcal{Q}, p) \ll (\mathcal{Q}', p')$ if $\mathcal{Q} \ll \mathcal{Q}'$ and $p + |\mathcal{Q}| - 1 < p'$. A *chain* $\mathcal{C}$ for an SSD $\mathcal{R}$ is a sequence of matches

$$\mathcal{C} = \langle (\mathcal{Q}_{j_1}, p_1), (\mathcal{Q}_{j_2}, p_2), \ldots, (\mathcal{Q}_{j_k}, p_k) \rangle,$$

all from $\mathcal{MS}$, such that $(\mathcal{Q}_{j_i}, p_i) \ll (\mathcal{Q}_{j_{i+1}}, p_{i+1})$ for all i, $1 \le i \le k - 1$.

There are two modes to score chains, depending on the nature of the search problem. If the multiple sequence-structure alignment our SSD is derived from and the searched sequences have comparable length, we want the chain to cover as much as possible of the sequence and we define the *global chain score* for chain $\mathcal{C}$ as follows:

$$gcsc\,(\mathcal{C}) = \sum_{i=1}^{k} \alpha(\mathcal{Q}_{j_i}). \tag{2}$$

Then, the global chaining problem is to find a chain $\mathcal{C}$ with maximum global chain score.

If we are searching in a whole genome or chromosome for a relatively short structural RNA, we are interested in local chains covering only parts of the genome or chromosome. Then we have to penalize gaps using a penalty function g and thus the *local chain score* is defined by

$$\begin{aligned} lcsc\,(\mathcal{C}) = \sum_{i=1}^{k-1} (\alpha(\mathcal{Q}_{j_i}) - \\ g\left((\mathcal{Q}_{j_i}, p_i), (\mathcal{Q}_{j_{i+1}}, p_{i+1})\right)) + \alpha(\mathcal{Q}_{j_k}) \end{aligned} \tag{3}$$

where

$$\begin{aligned} g\left((\mathcal{Q}_{j_i}, p_i), (\mathcal{Q}_{j_{i+1}}, p_{i+1})\right) \\ = \left|(p_{i+1} - p_i) - (l_{j_{i+1}} - r_{j_i})\right|. \end{aligned} \tag{4}$$

To solve the local chaining problem we use our own implementation of a fast local chaining algorithm described in [51] with modified gap costs. While the algorithm of [51] penalizes gaps by the sum of their lengths, our solution is based on the difference between their observed lengths (in the chain of matches) and their expected lengths (as given by the multiple alignment of the family), confer Equation 4. This algorithm runs in $O(q \log q)$ time where q is the size of $\mathcal{MS}$.

To solve the global chaining problem we have developed a new efficient chaining algorithm described next.

An improved method for global RSSP match chaining

So far our description was based on a single sequence. However, the results described below are based on a large set of sequences $S_1, \ldots, S_k$ as it occurs when searching a large sequence database. I.e. in case of databases like Rfam k can be in the range of millions. To handle these, we concatenate the single sequences with separator symbols and construct the affix array for the concatenation. For a given SSD $\mathcal{R} = \mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_L$, all RSSPs $\mathcal{Q}_i$, $1 \leq i \leq L$, are matched one after the other using fast bidirectional search on the affix array. This results in match sets $\mathcal{MS}(\mathcal{Q}_i)$ for RSSP $\mathcal{Q}_i$. L is typically in the range of tens while the number of RSSP matches for a particular sequence S_j is in the order of hundreds to thousands if S_j is an mRNA or complete genome sequence. For each match f the following information is recorded:

- The ordinal number i of the RSSP $\mathcal{Q}_i$ involved in f. This is denoted by $f.rssp$.
- The length of the RSSP involved in f. This is denoted by $f.length$.
- The number j of the sequence S_j f occurs in. This is denoted by $f.seqnum$.
- The starting position of f in S_j. This is denoted by $f.pos$.
- The weight $\alpha(\mathcal{Q}_{f.rssp})$ of f. The weight of f is denoted by $f.weight$.

In an initial sorting step the union $\mathcal{MS}$ of all match sets $\mathcal{MS}(\mathcal{Q}_i)$, $1 \leq i \leq L$, is sorted in ascending order of $f.seqnum$. Matches with identical sequence numbers are sorted in ascending order of the ordinal number of the RSSP, i.e., by $f.rssp$. Suppose that b^* is the size of $\mathcal{MS}$. As there are at most b^* sequences with at least one RSSP match, the sorting according to the sequence numbers can be done in $\mathcal{O}(k^* + b^*)$ time and $\mathcal{O}(k^*)$ space using the counting sort algorithm [52]. Here, k^* is the number of sequences with at least one RSSP match. As $k^* \leq b^*$, the sorting requires $\mathcal{O}(b^*)$ time and space. We obtain disjoint subsets $\mathcal{MS}(S_j)$, $1 \leq j \leq k$, where $\mathcal{MS}(S_j)$ is the set of all matches in $\mathcal{MS}$ matching a substring of S_j. As $\mathcal{MS}$ is ordered by the ordinal number of the RSSP and the counting sort algorithm is stable, the sets $\mathcal{MS}(S_j)$ are also sorted by the ordinal number of the RSSPs. Let $\mathcal{MS}(S_j, \mathcal{Q}_i)$ denote the matches $f \in \mathcal{MS}(S_j)$ such that $f.rssp = i$. In a second sorting step, each $\mathcal{MS}(S_j, \mathcal{Q}_i)$ is sorted according to the starting position of the matches. As this is a typical integer sorting problem, it requires $\mathcal{O}(b_{j,i} \log b_{j,i})$ time, where $b_{j,i}$ is the size of $\mathcal{MS}(S_j, \mathcal{Q}_i)$. Altogether, the two initial sorting steps can be performed in $\mathcal{O}\left(b^* + \sum_{j=1}^{k} \sum_{i=1}^{L} b_{j,i} \log b_{j,i}\right)$ time.

For all $S_1, S_2, \ldots, S_k$ one now solves independent chaining problems for sets $\mathcal{MS}(S_j)$, $1 \leq j \leq k$, of matches sorted according to the ordinal number of the RSSP and the starting position of the matches in S_j. Let j be fixed, but arbitrary. For each match $f \in \mathcal{MS}(S_j)$, the weight $f.weight$ is positive. Hence, an optimal chain ends with a match f such that there is no match f' satisfying $f \ll f'$. Similarly, an optimal chain begins with a match f' such that there is no match f satisfying $f \ll f'$.

The chaining problem is solved by a dynamic programming algorithm which tabulates for all matches $f' \in \mathcal{MS}(S_j)$ the maximum score $f'.score$ of all chains ending with f'. In addition, it computes the predecessor $f'.prec$ of f' in a chain with maximum score ending with f'. To obtain $f'.score$, one has to maximize over all matches f such that $f.rssp < f'.rssp$ and $f.pos + f.length - 1 < f'.pos$. This is a two dimensional search problem. As the matches in $\mathcal{MS}(S_j)$ are already sorted according to the first dimension (i.e., by the ordinal number of the RSSP), one can reduce it to a one dimensional sorting problem. This has already been observed in [51], and led to the development of an algorithm solving the chaining problem in $\mathcal{O}(b \log b)$, where b is the number of matches in $\mathcal{MS}(S_j)$. However, the algorithm of [51] was developed for chaining pairwise sequence matches. The RSSP chaining problem is a special instance of this problem: the first "sequence" consists of the positions $1, \ldots, L$, and a match for RSSP $\mathcal{Q}_i$ is a match of length one to position i. Moreover, matches at position i in the first sequence can be treated as being of equal length because they are matches to the same RSSP $\mathcal{Q}_i$. In addition to this, our initial sorting step delivers, for all i, $1 \leq i \leq L$, the matches in $\mathcal{MS}(S_j, \mathcal{Q}_i)$ in sorted order according to the starting position in S_j. All these properties allow us to simplify and improve the algorithm of [51] in the following aspects:

- While the algorithm of [51] requires a dictionary data structure with insert, delete, predecessor, and successor operations running in logarithmic time (e.g., an AVL-tree or a red-black tree [52]), our approach only needs a linear list, which is much easier to implement and requires less space.
- While the algorithm of [51] requires an initial sorting step using $\mathcal{O}(b^* \log b^*)$ time, our method only needs $\mathcal{O}\left(b^* + \sum_{j=1}^{k} \sum_{i=1}^{L} b_{j,i} \log b_{j,i}\right)$ time for this step. Note that the $b_{j,i}$ satisfy $\sum_{j=1}^{k} \sum_{i=1}^{L} b_{j,i} = b^*$.
- While the algorithm of [51] solves the chaining problem for $\mathcal{MS}(S_j)$ in $\mathcal{O}(b \log b)$ time, our approach runs in $\mathcal{O}(b \cdot L)$ time. If L is considered to be a constant, the running time becomes linear in b, where $b = |\mathcal{MS}(S_j)|$.

To explain our algorithm, let i, $1 \leq i \leq L$ be arbitrary but fixed and assume that all match sets $\mathcal{MS}(S_j, \mathcal{Q}_{i'})$, $i' < i$ have been processed. In a first loop over the sorted matches in $\mathcal{MS}(S_j, \mathcal{Q}_i)$ one determines the score of the matches. In a second loop, one inserts them into a linear list if necessary. The linear list contains a subset of the previously processed and scored matches. This split of the computation into two loops is different from the algorithm of [51] where the scoring and insertions are interweaved in one loop, requiring an extra array of length $2b$ containing references to the matches. The separation into two loops allows us to get rid of this extra array.

Now consider the first loop over all elements in $\mathcal{MS}(S_j, \mathcal{Q}_i)$ in sorted order of the match position in S_j. Let f' be the current element. At this point, all matches f such that $f.rssp < f'.rssp$ have been processed already. In particular, the score $f.score$ and the previous match (if any) in an optimal chain ending with f has been determined. Among the processed matches we only have to consider those matches f satisfying $f.pos + f.length - 1 < f'.pos$. If there is such a match, one takes the one with maximal score, say f. Then, the optimal chain ending with f' contains the previous match f, and the score is $f'.score = f'.weight + f.score$. If there is no such match, then the optimal chain ending with f' only consists of f' and $f'.score = f'.weight$.

Now consider the second loop over all elements in $\mathcal{MS}(S_j, \mathcal{Q}_i)$ for which the scores and predecessor matches (if any) are already determined. Let f' be the current element to be inserted. As explained in the previous case, one has to make sure that, among the processed matches, one can efficiently determine the match f with the maximum score such that $f.pos + f.length - 1$ is smaller than some value depending on f'. The processed matches are stored in a linear list which is sorted in ascending order of the position of the matches in S_j. Let $\prec_{pos}$ denote this order, that is $f \prec_{pos} f''$ if and only if $f.pos + f.length < f''.pos + f''.length$ for any matches f and f''. If for two processed matches f and f'' one has $f.pos < f''.pos$ and $f.score > f''.score$, then an optimal chain does not include f''. Each chain that uses f'' can also use f and increase the chain score. As a consequence, one has to take care that f'' is not inserted into the linear list or it is deleted if it was inserted earlier. In this way, $f \prec_{pos} f''$ always implies $f.score \leq f''.score$ for two matches f and f'' in the linear list. As the elements to be scored in the first loop and to be inserted in the second loop are ordered in the same way as the elements in the linear list, one can perform the scoring and the insertion loop (which also may involve deletions) by merging two lists of length l_1 and l_2 in $\mathcal{O}(l_1 + l_2)$ time where l_1 is the number of matches to be scored and inserted and l_2 is the length of the linear list involved. Let $b = |\mathcal{MS}(S_j)|$. As $l_1 + l_2 \leq b$, one obtains a running time of $\mathcal{O}(b)$ for each set $\mathcal{MS}(S_j, \mathcal{Q}_i)$. As there are L such sets, the running time is $\mathcal{O}(b \cdot L)$.

Results

Implementation and computational results

We implemented (1) the algorithms necessary for affix array construction, (2) the fast bidirectional search of RSSPs using affix arrays as sketched in Algorithm 2 (hereinafter called *BIDsearch*), (3) an on-line variant operating on the plain sequence (hereinafter called *ONLsearch*) for validation of *BIDsearch* and reference benchmarking, and (4) the efficient global and local chaining algorithms. Algorithm *ONLsearch* shifts a window of length $m = |RSSP|$ along the sequence of length n to be searched and compares the substring inside the window with the RSSP from left to right until a mismatch occurs. Hence, it runs in $\mathcal{O}(nm)$ time in the worst and $\mathcal{O}(n)$ time in the best case. Algorithms *BIDsearch* and *ONLsearch* were implemented in the program *afsearch*. The *afconstruct* program makes use of routines from the *libdivsufsort2* library (see http://code.google.com/p/libdivsufsort/) for computing the suf_F and suf_R tables in $\mathcal{O}(n \log n)$ time. For the construction of the lcp_F and lcp_R tables we employ our own implementation of the linear time algorithm of [37]. Tables aflk_F and aflk_R are constructed in $\mathcal{O}(n^2)$ worst-case time with fast practical construction time due to the use of the skip tables skp_F and skp_R [38]. The programs were compiled with the GNU C compiler (version 4.3.2, optimization option -O3) and all measurements were performed on a Quad Core Xeon E5410 CPU running at 2.33 GHz, with 64 GB main memory (using only one CPU core). To minimize the influence of disk subsystem performance the reported running times are user times averaged over 10 runs. Allowed base pairs were canonical Watson-Crick (A, U), (U, A), (C, G), (G, C), and wobble (G, U), (U, G), unless stated otherwise.

Affix array construction times

In a first experiment we constructed the affix array for genomes of selected model organisms of different sizes and stored it on disk. We measured the total running times needed by *afconstruct* to construct each table comprising the affix array. See Figure 7 for the results of this experiment. The total size for each table is given in Additional file 1, Table S2. Construction times were in the range of 25 minutes for the *C.elegans* genome containing ~ 100 megabases to 15.7 hours for the ~ 2 gigabase genome of the megabat *P.vampyrus*.

We also measured the running time of *afconstruct* to construct the affix array for a set of $3{,}192{,}599$ RNA sequences with a total length of ~ 622 MB compiled from the full alignments of all Rfam release 10.0 families. The construction and storage on disk required 126 minutes. In the following we refer to this dataset as RFAM10 for short.

Influence of loop length on search performance

In a second experiment we investigated the influence of the loop length and the number of non-ambiguous characters in the loop of an RSSP on the running time of *BIDsearch* and *ONLsearch*. For this experiment we constructed artificial RSSPs with a fixed stem length of 7 and a loop length l varying from 3 to 20. For each loop length, we also varied the number of consecutive non-ambiguous characters q from 0 to 4. For $q = 0$ this means that the RSSP contains structural constraints only. That is, for $q = 0$ and $l = 5$ the used RSSP matches all substrings that are able to fold into a stem-loop structure with loop length 5 and stem length 7. Such a pattern is written in dot-bracket notation as `(((((((.....)))))))`. Allowed base pairs were (A, U), (U, A), (C, G), and (G, C). We measured the time needed by *BIDsearch* and *ONLsearch* to search for these patterns in the RFAM10 dataset.

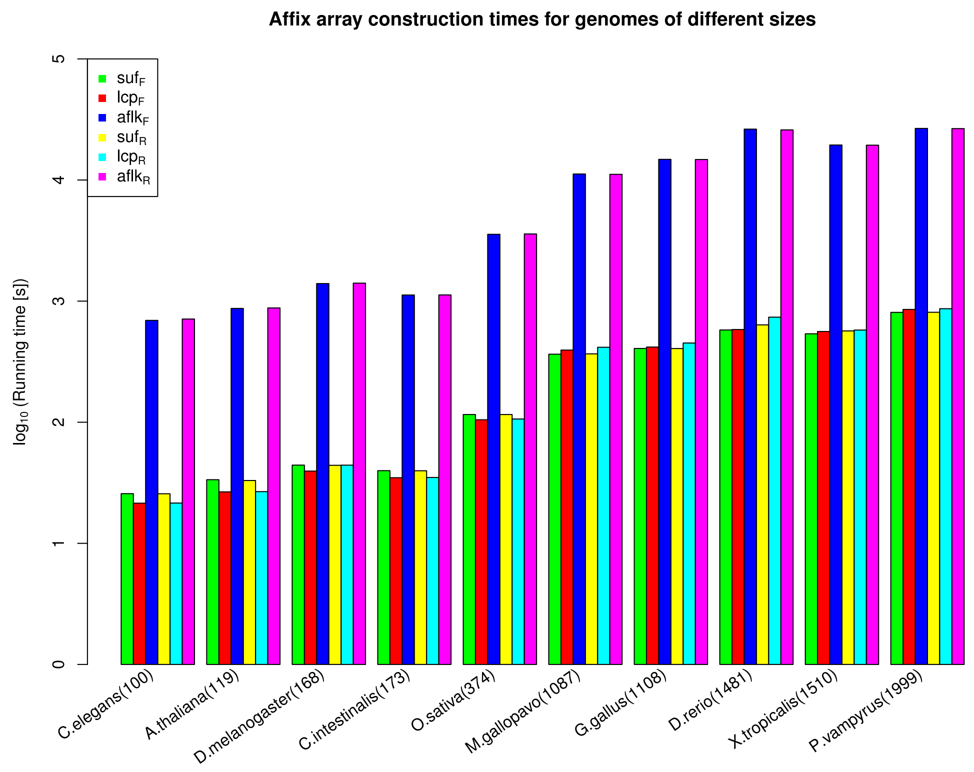


Figure 7: Experiment 1: Running times for affix array construction for genomes of different model organisms. Genome sizes are given for each organism in megabases in brackets. We measured the running time in seconds for all tables the affix array consists of (y-axis, $\log_{10}$ scale). Total construction times were in the range of ~ 25 minutes for *C.elegans* up to 15.7 hours for *P.vampyrus*.

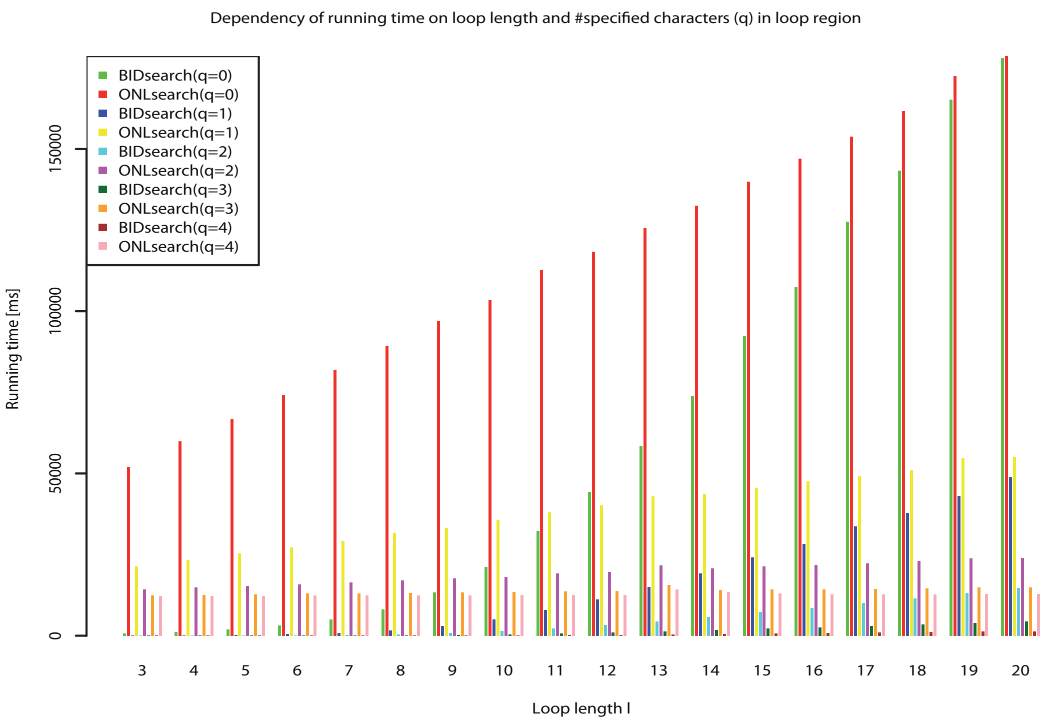


Figure 8: Experiment 2: Influence of loop length and number of non-ambiguous characters in loop region on total running time of *BIDsearch* and *ONLsearch*. We measured the running time in milliseconds to search with artificial RSSPs with loops of varying length $l \in \{3, \dots, 20\}$ on ~ 622MB of RNA sequence data. For each loop length l we also varied the number $q \in \{0, \dots, 4\}$ of non-ambiguous nucleotides in the loop. The used RSSPs had a fixed stem length of 7. For more details on this experiment see corresponding text.

Results are given in Figure 8. In this experiment *BIDsearch* performed very well and was faster than *ONLsearch* for all parameter combinations. We also investigated the influence of different stem length (data not shown here) and found that the impact on the total running time is negligible. We observe that the advantage of *BIDsearch* over *ONLsearch* decreases with increasing loop length l for fixed q. We explain this behavior with the increasing number of affix-intervals that have to be processed for finding all different substrings of the sequences that match the RSSP. However, even for an RSSP with loop length $l = 20$ containing only structural constraints ($q = 0$), *BIDsearch* is still faster than *ONLsearch*. We further notice that the number of non-ambiguous characters in the loop region has a strong influence on the running time of *BIDsearch*. That is, by specifying only a few conserved nucleotides in the RSSP's loop region, the running time of *BIDsearch* is reduced dramatically. For an example of this effect, see the running times of *BIDsearch* in Figure 8 for parameters $l = 15$ and $q \in \{2, 3, 4\}$. This renders *BIDsearch* in particular useful for searching with RSSPs with moderate loop length or existing sequence conservation in the loop region. The speedup factors measured in this experiment were in the range from 1.001 to 78.1 for $q = 0$ and from 9.28 to 11×10^3 for $q = 4$. Table 1 gives more details on the speedups of *BIDsearch* over *ONLsearch* for all investigated combinations of q and l.

Searching large sequence databases

To measure the performance of *BIDsearch* for non-artificial real-world RSSPs, we manually compiled a set of 397 RSSPs describing 42 highly structured RNA families taken from the RFAM10 database.

l	3	4	5	6	7	8	9	10	11
$q = 0$	78.10	48.64	35.42	23.55	16.35	11.01	7.31	4.89	3.48
$q = 1$	329.81	180.45	105.67	57.41	33.75	19.20	11.30	7.14	4.81
$q = 2$	749.94	418.65	227.45	121.80	67.81	36.99	21.44	12.73	8.41
$q = 3$	2,345.17	1,169.53	653.31	353.49	188.34	103.34	56.59	33.08	20.79
$q = 4$	11,045.75	3,638.14	2,144.8	1,132.53	610.63	338.77	184.56	106.11	64.93
l	12	13	14	15	16	17	18	19	20
$q = 0$	2.67	2.15	1.79	1.51	1.37	1.20	1.13	1.07	1.00
$q = 1$	3.58	3.13	2.28	1.89	1.68	1.46	1.35	1.27	1.12
$q = 2$	5.96	4.88	3.64	2.94	2.57	2.19	2.02	1.82	1.63
$q = 3$	14.27	11.88	8.25	6.50	5.53	4.74	4.19	3.76	3.34
$q = 4$	43.09	35.23	25.74	19.52	15.91	13.25	11.75	10.32	9.28

Table 1: Experiment 2: Obtained speedup of *BIDsearch* over *ONLsearch* for different loop length $l \in \{3, \dots, 20\}$ and number of non-ambiguous characters in the loop region $q \in \{0, \dots, 4\}$. For the parameter combination $l = 3, q = 4$ also one character of the stem was specified.

BIDsearch	*ONLsearch*	*RNAMotif*	*RNABOB*
$46.1(1)$	6,203(134.5)	11,745(254.7)	9,061(196.5)

Table 2: Experiment 3 (A): Running times in seconds needed by the programs to search for 397 RSSPs describing 42 RFAM10 families in $\sim$ 622 megabases of RNA sequence data. For each program the speedup factor of *BIDsearch* over the particular program is given in brackets.

These were all families with a consensus secondary structure containing at least 5 stem-loop substructures. We measured the running time needed by *BIDsearch*, *ONLsearch*, and the widely used tools *RNAMotif* [13] and *RNABOB* [15] to search for these 397 RSSPs in the RFAM10 dataset. As expected, all tools delivered identical results. However, while it took *BIDsearch* less than 50 seconds to search for the 397 patterns as shown in Table 2, *RNABOB* and *RNAMotif* needed more than 2.5 and 3.2 hours respectively to complete the same task. This made for a speedup factor of 196.5 (254.7) for *BIDsearch* over *RNABOB* (*RNAMotif*). Even if we include the time needed for affix array construction, *BIDsearch* is still faster than *RNABOB* and *RNAMotif*.

We also investigated the distribution of speedup factors obtained by *BIDsearch* when searching for the 397 RSSPs. We observed that *BIDsearch* is more than $50{,}000$ times faster than *RNABOB* and *RNAMotif* for the majority of the patterns and that the total search time required by *BIDsearch* is dominated by only a small number of patterns. These patterns describe large unconserved loop regions. See Figure S3 in Additional file 1 for a graphical visualization of the distribution of speedup factors.

Scaling behavior of bidirectional pattern search using affix arrays

In a further experiment we investigated the scaling behavior of *BIDsearch* and *ONLsearch* for an increasing size of sequences to be searched. For this, we searched with different RSSPs on random subsets of RFAM10 of different sizes and measured the running time for both algorithms. The results are given in Figure 9. Here pattern1 is an RSSP containing only structural constraints. It describes a stem-loop with loop length 4, stem length 10 and no specified nucleotides in the loop region. The RSSP pattern2 (pattern3) only differ from pattern1 by containing one (two consecutively) non-ambiguous nucleotides in the loop region.

In this experiment *BIDsearch* clearly showed a sublinear scaling behavior, whereas *ONLsearch* scaled only linearly. It took *BIDsearch* only 566.8 (pattern1), 133.8 (pattern2), and 37.1 (pattern3) milliseconds to search the whole RFAM10 dataset. The obtained speedups of *BIDsearch* over *ONLsearch* were in the range from 4.63 (*1MB subset*) to 104.79 (*full* RFAM10) for pattern1, from

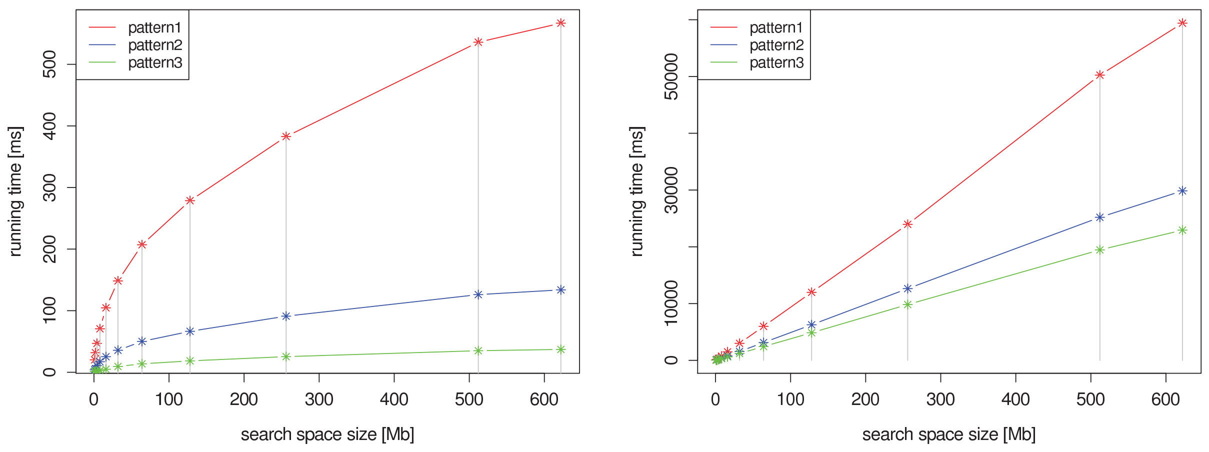


Figure 9: Scaling behavior *BIDsearch* (left) and *ONLsearch* (right). We measured the running time needed to search with three different patterns on random subsets of RFAM10 of different sizes. For details, see main text.

12.23 (*1MB subset*) to 223.18 (*full* RFAM10) for pattern2, and from 35.0 (*1MB subset*) to 618.37 (*full* RFAM10) for pattern3. We observe again that the specification of only one or two nucleotides in an RSSP's loop region considerably reduces the running time of the *BIDsearch* algorithm.

RNA family classification by global chaining of RSSP matches

To demonstrate the effect of global chaining of RSSP matches, we searched with an SSD built for the Rfam family of OxyS RNAs (Acc.: RF00035). OxyS is a small 109-nucleotide long non-coding RNA which is included in response to oxidative stress in *E.coli* [53]. Members of this family fold into a characteristic secondary structure consisting of three stem-loop substructures, referred to as HP1, HP2, and HP3 in Figure 10 (C). From the three stem-loops we derived three descriptors called RSSP1, RSSP2, and RSSP3, which constitute the SSD describing this family. We note that in this experiment the RSSPs were constructed to guarantee high specificity and thus to minimize the number of false positives. For the SSD specified in *Structator* syntax, see Figure 10 (A). Searching for this SSD in RFAM10, *Structator* delivers 8,619 matches for RSSP1, 1,699 matches for RSSP2, and 142,219 matches for RSSP3. Instead of reporting these matches, *Structator* computes high-scoring global chains for each sequence containing matches to all three RSSPs. The chains and the sequences they occur in are reported in descending order of the chain score. This procedure resulted in 61 sequences, all belonging to the OxyS family which contains 115 members in total. Hence, by considering only high-scoring chains all the spurious RSSP matches were eliminated. We also described the same three stem-loops in a format compatible with *RNAMotif* (see Figure 10 (B)). A search on RFAM10 with this descriptor returned exactly the same 61 sequences. However, *Structator* operating in *BIDsearch* (*ONLsearch*) mode with subsequent global chaining of RSSP matches needed only 3.9 (122.5) seconds to identify all family members, whereas *RNAMotif* needed 84.7 seconds. The search times for *Structator* include 0.05 seconds required for the chaining.

We also employed global chaining to detect members of the structurally more complex family of Citrus tristeza virus replication signal (Rfam Acc.: RF00193). Therefore we built an SSD comprising 8 RSSPs, describing 8 of 10 stem-loops the molecule is predicted to fold into. For more information on the molecule's secondary structure and the used descriptor, see Additional file 1, Figure S4. Using *Structator* operating in *BIDsearch* (*ONLsearch*) mode and global chaining of RSSP matches it took

(A)

```
>RSSP1|maxrightloopextent=1|maxleftloopextent=1|maxmispair=6|weight=1
NNNNNNNNNNNNNNNNNNNNNNACCCNUNANNNNNNNNNNNNNNNN
(((((((.((((.((.((..........)).))..)))))))))))
>RSSP2|maxrightloopextent=5|weight=1
GNNNNNCUCACNN
((((.....))))
>RSSP3|maxmispair=2|maxrightloopextent=2|weight=1
NNGGANCUNNNNNNNNNNNN
(((((((.....)))))))
```

(B)

```
parms
           wc +=gu;
descr
 h5(len=7)
 ss(len=1)
 h5(len=4)
 ss(len=1)
 h5(len=2)
 ss(len=1)
 h5(len=2)
   ss(seq="N\{0,1\}NNNNACCCNUN\{0,1\}",minlen=10,maxlen=12)
 h3(seq="NA",len=2)
 ss(len=1)
 h3(len=2)
 ss(len=2)
 h3(len=4)
 h3(len=7)
 ss(minlen=2,maxlen=3)  #single strand between HP1 and HP2
 h5(len=4)
   ss(seq="NNCUCN\{0,5\}",minlen=5,maxlen=10)
 h3(len=4)
 ss(minlen=27,maxlen=31)#single strand between HP2 and HP3
 h5(len=7, seq="NNGGANC",mispair=2,ends='mm')
   ss(seq="UN\{4,6\}",minlen=5,maxlen=7)
 h3(len=7)
```

(C)

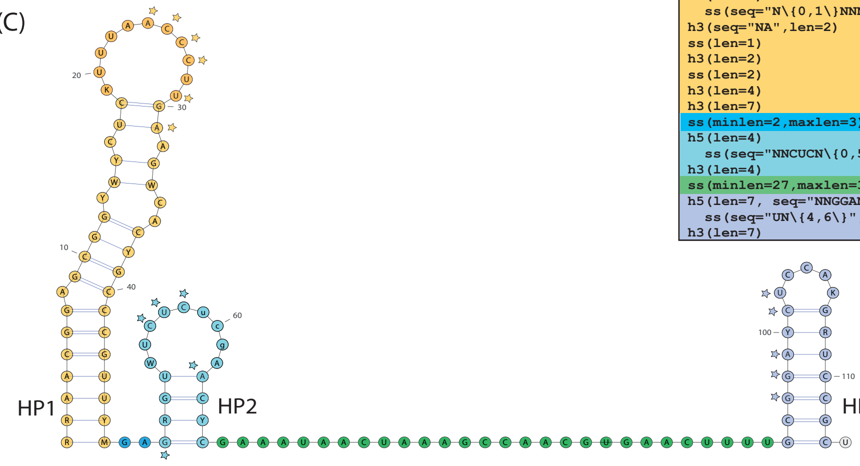


Figure 10: (A) Secondary structure descriptor for the family of OxyS RNAs in *Structator* syntax. The SSD consists of RSSPs RSSP1, RSSP2, and RSSP3 describing the three stem-loop structures (HP1, HP2, and HP3, see (C)) of this small non-coding RNA. (B) *RNAMotif* descriptor for the same structural elements. (C) Consensus secondary structure of the OxyS RNA family as drawn by *VARNA* [30]. Sequence information (non-wildcard nucleotides) used in both descriptors are marked with an asterisk. Observe that both descriptors use predominantly structure and very little sequence information.

only 1.3 (138.7) seconds to search RFAM10 with this SSD, where 0.06 seconds were required for the chaining. The computed global chains with a minimum length of 5, computed from the $184{,}199$ single RSSP matches, were ranked according to their global chain score. We observe that the sequences containing the 37 highest scoring chains are exactly all 37 members of the family.

In addition we measured the performance of *Structator* using global chaining for RNA family classification with manually compiled SSDs for 42 Rfam families. For the results of this experiment see Additional file 1, Table S4.

Searching whole genomes using local chains of RSSP matches

As an example of searching a complete genome or whole chromosomes for non-coding RNAs, we searched for the RNA gene Human accelerated region 1F (HAR1F) on both strands of the human genome sequence. HAR1F is one of 49 regions in the human genome that differ significantly from highly conserved regions of the chimpanzee [54]. The consensus structure of the HAR1F family in Rfam (Acc.: RF00635) contains three stem-loop regions, denoted HP1, HP2, and HP3 in Figure 11 (A). From these regions, we built an SSD for the family with RSSPs RSSP1, RSSP2, and RSSP3, shown in Figure 11 (B). Since we were searching on complete chromosomes, we only wanted to consider RSSP matches that occurred at a similar distance to each other w.r.t. to the distances of the corresponding descriptors in the SSD. Therefore, unlike in the previous experiment where we searched for global chains of RSSP matches, we now computed high-scoring local chains. Gap costs were computed according to Equation (4) and we used an RSSP weight $\alpha(\mathsf{RSSP}_i) = 10$, for $1 \leq i \leq 3$. Affix array construction for all human chromosomes was accomplished in 12.6 hours by *afconstruct*.

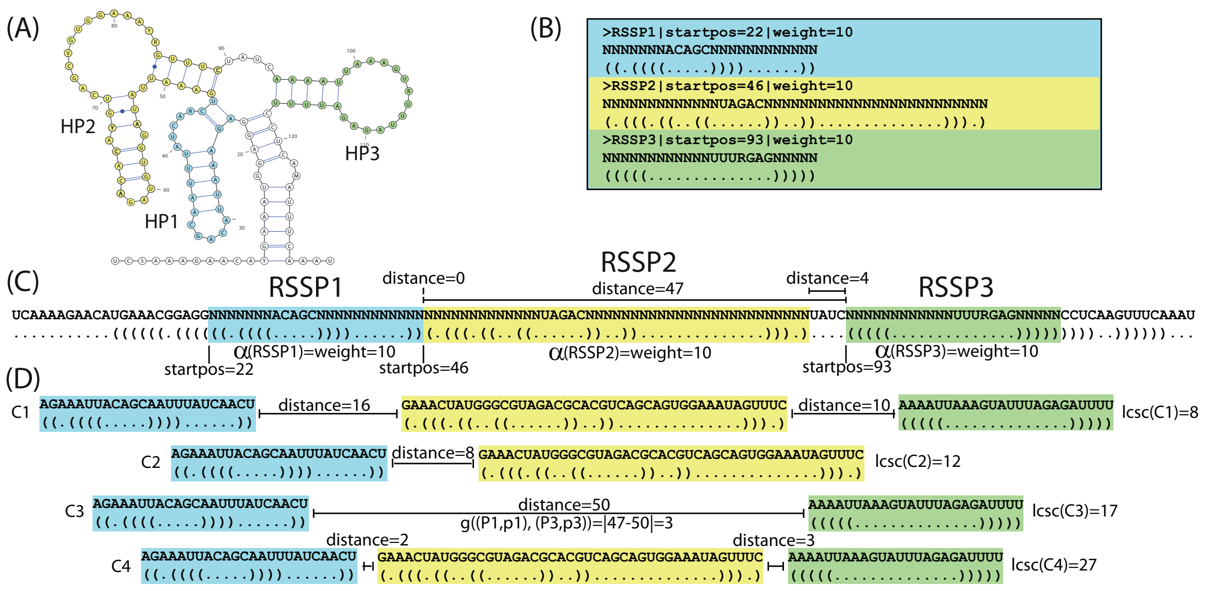


Figure 11: (A) Consensus secondary structure visualized with the *VARNA* program of the HAR1F RNA family showing stem-loops HP1, HP2, and HP3. (B) SSD consisting of RSSP1, RSSP2, and RSSP3 in *Structator* syntax describing the three stem-loop regions of HAR1F. (C) Regions of HAR1F described by the RSSPs, including distances $l_{i+1} - r_i$, $1 \leq i < 3$, between neighbored RSSPs and RSSP weights $\alpha(\mathsf{RSSP}_i)$, $1 \leq i \leq 3$. (D) Examples of local chains $\mathcal{C}_i$, $1 \leq i \leq 4$ found with the SSD, showing, in each chain, the distance between RSSP matches and their local chain score $lcsc\,(\mathcal{C}_i)$. Gap cost computation according to Equation (4) is shown exemplary for the two RSSP matches of chain $\mathcal{C}_3$.

We searched with *Structator* for the three RSSPs and found $15{,}090$, $1{,}578$, and $14{,}491$ matches for RSSP1, RSSP2, and RSSP3, respectively. For these RSSP matches we computed local high-scoring chains (see Figure 11 (D)). Chains $\mathcal{C}$ were ranked according to their local chain score $lcsc\,(\mathcal{C})$. We observed that the highest-scoring chain corresponds to the correct location of the gene on chromosome 20. Using *BIDsearch* (*ONLsearch*) this task needed 3.1 (633.4) seconds only, including 0.02 seconds for the chaining. *RNAMotif* also found a single match corresponding to the correct location of the gene, but needed 274.7 seconds. See Figure S5 in Additional file 1 for the used *RNAMotif* descriptor.

Comparison of implementations of bidirectional pattern search

In the last experiments we compared *Structator*'s running time using using *BIDsearch* with the time needed by a recently published bidirectional pattern search implementation for the same task. The implementation of [55], to which we refer as *BWI*, uses a compressed data structure called bidirectional wavelet index. We remark that *BWI* can only search with a small set of hard-coded patterns, i.e., the user cannot use it to search with his/her own patterns. Moreover, unlike *Structator*, which provides a full command line interface with many configurable options (see section about the software package), *BWI* reports neither matching substrings nor matching positions (which is known to be the most time consuming part when querying compressed index structures [26]). It only outputs the search time of individual patterns and the number of matches. Thus, it serves rather as a prototype implementation of the concepts introduced in [55]. Nevertheless, since it also makes use of bidirectional search, we compared *BWI* with *Structator* using *BWI*'s hard-coded patterns. See Table 3 for the results. Details of the database and patterns are as previously described [55]. We noticed that *BIDsearch* was faster than

	hairpin1	hairpin2	hairpin4	hloop(5)	acloop(5)	acloop(10)
BWI	10,484	64	612	26,413	896	420
BIDsearch	8,325	32	330	16,768	511	295
BIDsearch vs. *BWI*	1.26	2	1.85	1.58	1.75	1.42

Table 3: Search time comparison between *Structator*'s *BIDsearch* and an implementation, here called *BWI*, of bidirectional search using the wavelet tree data structure described in [55]. Search times are in milliseconds. The last row shows the speedup of *BIDsearch* over *BWI*.

BWI for matching all patterns by up to factor 2, hence making it preferable when speed is most important. However, we note that *BWI*'s compressed wavelet index consumes significantly less memory than *Structator*'s affix array index, which would make *BWI* preferable in cases where space consumption is critical. See Table S3 in Additional file 1 for the memory required by *BWI*'s index for different genomes.

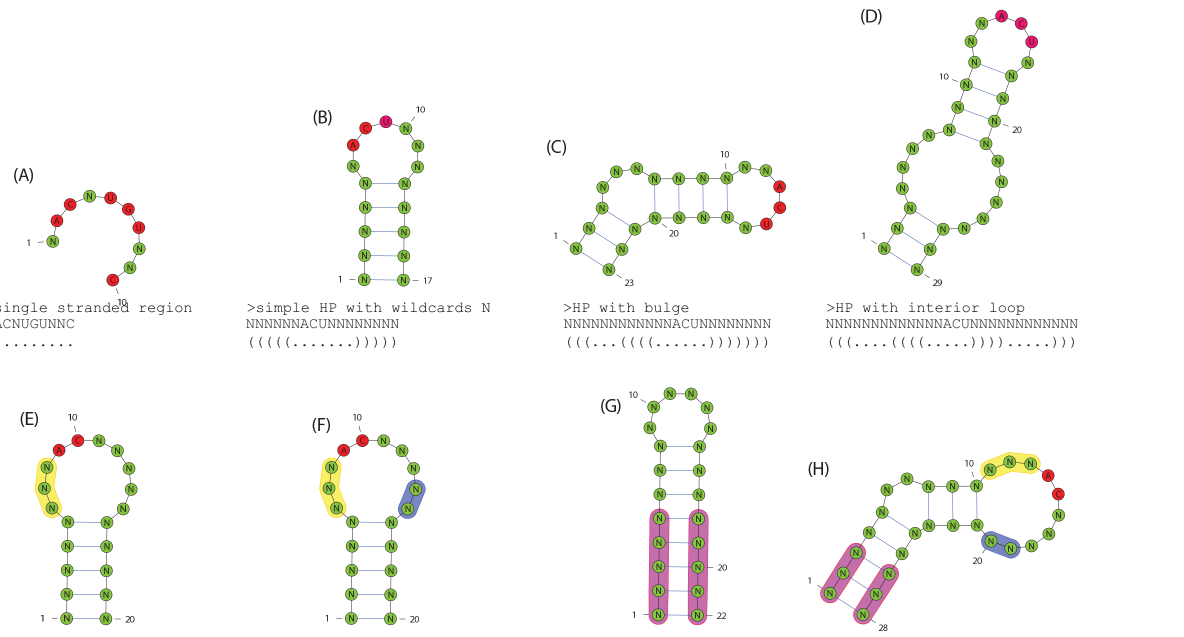
We also measured the speedup of *Structator* running in *BIDsearch* mode over *ONLsearch* and compared the results with previously reported measurements [27]. Because the implementation used there is not available (personal communication with the author), we calculated relative speedups based on the reported absolute running times. Details on this experiment are given in Additional file 1, Section S2.

***Structator* software package**

Structator is an open-source software package for fast database search with RNA structural patterns implementing the algorithms and ideas presented in this work. It consists of the command line programs *afconstruct* and *afsearch*.

afconstruct implements all algorithms necessary for affix array construction, namely a lightweight suffix sorting algorithm for construction of the suffix arrays suf_F and suf_R, the algorithm for construction of tables lcp_F and lcp_R [37], and the algorithm for computation of the affix link tables aflk_F and aflk_R. The program constructs all or if necessary only some of the tables of the affix array for a target database provided in FASTA format and stores them on disk. Therefore the program can also be used to compute only the tables needed for a traditional enhanced suffix array [35]. *afconstruct* can handle RNA as well as DNA sequences. Moreover, it supports the transformation of input sequences according to user-defined (reduced) alphabets and allows the index construction for transformed sequences. Such personalized alphabets are easily specified in a text file.

afsearch is the program for performing structural pattern matching. That is, it searches (ribo)nucleic acid sequence databases for entries that can adopt a particular secondary structure. For an overview of the supported RNA sequence-structure patterns (RSSPs), see Figure 12. The simplest RSSP describes a single-stranded region, where ambiguous (not well-conserved) nucleotides can be specified with IUPAC characters. All ambiguous IUPAC characters are hard-coded in *afsearch*, e.g. N standing for nucleotides A, C, G, and U (and T) and R standing for A and G. Besides fixed-length RSSPs with or without ambiguous characters (Figure 12 (A) until (D)), also RSSPs describing loop or stem regions of variable size (Figure 12 (E) until (H)) are supported. More precisely, one can specify with parameters *maxleftloopextent (mllex)* and *maxrightloopextent (mrlex)* a variable number of allowed extensions to the left (nucleotides marked in yellow in Figure 12 (E)) and/or to the right (nucleotides marked in blue in Figure 12 (F)) for the specified loop pattern. Variable stem sizes can be addressed with parameter *maxstemlength (msl)* (see regions marked in pink in Figure 12 (G)). Also supported is the combination of variable loop and stem size (see Figure 12 (H)) and a maximal number of allowed mispairings in the stem region. All these different RSSPs can be specified by the user in a text file which use, as shown in



```
>single stranded region
NACNUGUNNC
..........
```

```
>simple HP with wildcards N
NNNNNNACUNNNNNNNN
(((((.......)))))
```

```
>HP with bulge
NNNNNNNNNNNNNACUNNNNNNN
(((...((((......)))))))
```

```
>HP with interior loop
NNNNNNNNNNNNNNACUNNNNNNNNNNNN
(((....((((.....)))).....)))
```

```
>HP with variable loop|mllex=3
NNNNNACNNNNNNNNNN
(((((.......)))))
```

```
>HP with var. loop|mllex=3|mrlex=2
NNNNNACNNNNNNNNN
(((((.....)))))
```

```
>HP with var. stem|msl=8
NNNNNNNNNNNN
(((......)))
```

```
>HP with var. loop and stem|mllex=3|mrlex=2|msl=8
NNNNNNNNNNACNNNNNNNNNN
((...(((.........)))))
```

Figure 12: Supported structural patterns and corresponding pattern definitions in *Structator* syntax. Non-ambiguous nucleotides are marked in red. Positions containing ambiguous nucleotides, denoted here with character N, are marked in green and can contain any nucleotide from $\mathcal{A}$. Maximal allowed left and right extensions of the loop region of a pattern as specified by parameters *maxleftloopextent (mllex)* and *maxrightloopextent (mrlex)* are marked in yellow and blue, respectively. Allowed possible extensions of a pattern's stem region as specified by parameter *maxstemlength (msl)* are marked in purple. As an example for the semantics of the parameter *msl* consider pattern (G): it matches all substrings of the searched sequence that are able to fold into a stem-loop structure with loop length 6 and stem length between 3 and 8. For further details see corresponding text.

Figure 12, an expressive but easy to understand pattern syntax. For additional details on the supported patterns see the corresponding section in the *Structator* user manual. *afsearch* also permits user-defined base pairing rules. That is, the user can define an arbitrary subset from $\mathcal{A} \times \mathcal{A}$ as valid pairings. This ensures a maximum of flexibility. For example, the standard canonical Watson-Crick pairings as well as non-standard pairings such as G-U can be specified.

The search is performed efficiently on a pre-computed affix array. *afsearch* implements the bidirectional index-based search algorithms *BIDsearch* and the online algorithm *ONLsearch* operating on the plain sequence, both extended to support patterns with variable loop size and/or stem length. Further, it implements the methods for fast global and local chaining of RSSP matches. The search with RSSPs can be performed on the forward and, in case of nucleotide sequences, also on the reverse strand. Searching on the reverse strand is implemented by reversal of the RSSP and transformation according to Watson-Crick base pairing. Hence it is sufficient to build the affix array for one strand only.

RSSP matches can be reported directly by *afsearch* or can be used as input for the computation of high-scoring global or local chains of matches. Computed chains resemble the order of the RSSPs given in the pattern file and are reported in descending order of their chain score. This allows the description of complex secondary structures with our new concept of secondary structure descriptors (SSDs). This is done by simply specifying a series of RSSPs in the pattern file describing the stem-loop substructures the RNA molecule is composed of in the order of their occurrence in 5' to 3' direction. To incorporate different levels of importance or significance of an RSSP into SSD models and subsequently in the computation of chain scores, RSSP specific weights can be defined in the pattern file. This is particularly useful in the context of RNA family classification where the used SSD may be derived from a multiple sequence-structure alignment or a consensus structure-annotated multiple sequence alignment. Here, it permits the assignment of higher weights to RSSPs describing highly conserved functionally important structural elements occurring in a family of RNAs, and lower weights to RSSPs describing less conserved substructures that occur only in certain members of the family.

The output format of *afsearch* contains all available information of a match or chain of matches, either in a human-readable, or a tab-delimited format. Moreover, *afsearch* can also report matches in BED format. This allows a direct visualization of the results in e.g. the UCSC genome browser.

Discussion and conclusion

We have presented a method for fast index-based search of RNA sequence-structure patterns (RSSPs), implemented in the *Structator* software. As part of the software, we give the first publicly available implementation of bidirectional pattern search using the affix array data structure. For the majority of biologically relevant RSSPs, our implementation of *BIDsearch* shows superior performance over previous programs. In a benchmark experiment on the Rfam database, *BIDsearch* was faster than *RNAMotif* and *RNABOB* by up to two orders of magnitude. Furthermore, in a comparison between *BIDsearch* and the program of [55], which works on compressed index data structures, *BIDsearch* was faster by up to 2 times. We observed that for RSSPs with long unconserved loop regions, the advantage of *BIDsearch* over *ONLsearch* decreases. For such cases, *Structator* can also employ *ONLsearch* on the plain sequence data. As a further contribution, we presented for the first time a detailed complexity analysis of bidirectional search using affix arrays. While bidirectional search does not does not improve the worst-case time complexity compared to online search, in practice it runs much faster than online search algorithms and the running time scales sublinearly with the length n of the searched sequences.

Our implementation of the affix array data structure requires only $18n$ bytes of space. This is a significant space reduction compared to the $\sim 45n$ bytes needed for the affix tree. With the program *afconstruct* we present for the first time a command line tool for the efficient construction and persistent storage of affix arrays that can also be used as a stand-alone program for index construction.

With the new concept of RNA secondary structure descriptors (SSDs) combined with fast global and local chaining algorithms, all integrated into *Structator*, we also introduce a powerful technique to describe RNAs with complex secondary structures. This even allows to effectively describe RNA families containing branching substructures like multi-loops, by decomposition into sequences of non-branching substructures that can be described with RSSPs. Compared to programs like *RNAMotif*, *Structator*'s pattern description language for RSSP formulation is simple but powerful, in particular in combination with the SSD concept. Beyond the algorithmic contributions, we provide with the *Structator* software distribution a robust, well-documented, and easy-to-use software package implementing the ideas and algorithms presented in this manuscript.

Availability

The *Structator* software package including documentation is available in binary format for different operating systems and architectures and as source code under the GNU General Public License Version 3. See http://www.zbh.uni-hamburg.de/Structator for details.

Authors' contributions

F.M. implemented the presented algorithms and wrote parts of the manuscript and the *Structator* manual. S.K. developed and implemented the RSSP chaining algorithms and contributed to the manuscript. S.W. provided supervision and wrote parts of the manuscript. M.B. initiated the project, provided supervision and guidance, designed/performed the experiments and wrote large parts of the manuscript. R.B. contributed to the introduction. All authors read and approved the final manuscript.

Acknowledgments

This work was supported by the German Research Foundation (grant WI 3628/1-1). We also thank the anonymous referees, especially referee 2, for their valuable comments and suggestions.

References

1. Mattick J: **RNA regulation: a new genetics?** *Nat Rev Genet* 2004, **5**(4):316–323.
2. Mattick J, Taft R, Faulkner G: **A global view of genomic information - moving beyond the gene and the master regulator**. *Trends Genet.* 2009.
3. Gardner P, Daub J, Tate J, Moore B, Osuch I, Griffiths-Jones S, Finn R, Nawrocki E, Kolbe D, Eddy S, Bateman A: **Rfam: Wikipedia, clans and the "decimal" release**. *Nucl. Acids Res.* 2010.
4. Gardner P, Daub J, Tate J, Nawrocji E, Kolbe D, Lindgreen S, Wilkinson A, Finn R, Griffith-Jones S, Eddy S, Bateman A: **Rfam: updates to the RNA families database**. *Nucl. Acids Res.* 2008, **37**:D136–D140.
5. Gardner PP, Wilm A, Washietl S: **A benchmark of multiple sequence alignment programs upon structural RNAs**. *Nucl. Acids Res.* 2005, **33**(8):2433–9.
6. Höchsmann M, Voss B, Giegerich R: **Pure multiple RNA secondary structure alignments: a progressive profile approach**. *IEEE/ACM Trans Comput Biol Bioinform* 2004, **1**:53–62.
7. Siebert S, Backofen R: **MARNA: multiple alignment and consensus structure prediction of RNAs based on sequence structure comparisons**. *Bioinformatics* 2005, **21**(16):3352–3359.
8. Sankoff D: **Simultaneous solution of the RNA folding, alignment and protosequence problem**. *SIAM Journal on Applied Mathematics* 1985, **45**:810–825.

9. Gorodkin J, Heyer LJ, Stormo GD: **Finding the most significant common sequence and structure motifs in a set of RNA sequences**. *Nucl. Acids Res.* 1997, **25**(18):3724–32.
10. Havgaard J, Lyngso R, Stormo G, Gorodkin J: **Pairwise local structural alignment of RNA sequences with sequence similarity less than 40%.** *Bioinformatics* 2005, **21**:1815–1824.
11. Mathews DH, Turner DH: **Dynalign: an algorithm for finding the secondary structure common to two RNA sequences**. *Journal of Molecular Biology* 2002, **317**(2):191–203.
12. Will S, Reiche K, Hofacker IL, Stadler PF, Backofen R: **Inferring noncoding RNA families and classes by means of genome-scale structure-based clustering**. *PLoS Comput. Biol.* 2007, **3**(4):e65.
13. Macke T, Ecker D, Gutell R, Gautheret D, Case D, Sampath R: **RNAMotif – A new RNA secondary structure definition and discovery algorithm**. *Nucl. Acids Res.* 2001, **29**(22):4724–4735.
14. Gautheret D, Major F, Cedergren R: **Pattern searching/alignment with RNA primary and secondary structures: an effective descriptor for tRNA**. *Comput Appl Biosci* 1990, **6**(4):325–31.
15. **RNABOB: a program to search for RNA secondary structure motifs in sequence databases** [http://selab.janelia.org/software.html].
16. Chang T, Huang H, Chuang T, Shien D, Horng J: **RNAMST: efficient and flexible approach for identifying RNA structural homologs**. *Nucl. Acids Res.* 2006, **34**:W423–W428.
17. Dsouza M, Larsen N, Overbeek R: **Searching for patterns in genomic data**. *Trends Genet.* 1997, **13**(12):497–8.
18. Grillo G, Licciulli F, Liuni S, Sbisà E, Pesole G: **PatSearch: A program for the detection of patterns and structural motifs in nucleotide sequences**. *Nucl. Acids Res.* 2003, **31**(13):3608–12.
19. Nawrocki E, Eddy S: **Query-dependent banding (QDB) for faster RNA similarity searches**. *PLoS Comput. Biol.* 2007, **3**(56).
20. Nawrocki E, Kolbe D, Eddy S: **Infernal 1.0: inference of RNA alignments**. *BMC Bioinformatics* 2009, **25**:1335–1337.
21. Klein R, Eddy S: **RSEARCH: finding homologs of single structured RNA sequences**. *BMC Bioinformatics* 2003, **4**:44.
22. Sakakibara Y: **Pair hidden markov models on tree structures**. *BMC Bioinformatics* 2003, **19**:i232–40.
23. Gautheret D, Lambert A: **Direct RNA motif definition and identification from multiple sequence alignments using secondary structure profiles**. *J Mol Biol* 2001, **313**:1003–11.
24. Gusfield D: *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press 1997.
25. Manber U, Myers E: **Suffix arrays: a new method for on-line string searches**. *SIAM Journal on Computing* 1993, **22**(5):935–948.
26. Ferragina P, Manzini G: **Indexing compressed text**. *Journal of the ACM* 2005, **52**(4):552–581.
27. Strothmann D: **The affix array data structure and its applications to RNA secondary structure analysis**. *Theor. Comput. Sci.* 2007, **389**(1-2):278–294.
28. Mauri G, Pavesi G: **Algorithms for pattern matching and discovery in RNA secondary structure**. *Theor. Comput. Sci.* 2005, **335**:29–51.
29. Maaß MG: **Linear bidirectional on-line construction of affix trees**. *Algorithmica* 2003, **37**:43–74.
30. Darty K, Denise A, Ponty Y: **VARNA: Interactive drawing and editing of the RNA seondary structure**. *Bioinformatics* 2009, **25**(15):1974–1975.
31. Mauri G, Pavesi G: **Pattern discovery in RNA secondary structures using affix trees**. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, Volume 2676*, Springer 2003:278–294.

32. Kärkkäinen J, Sanders P: **Simple linear work suffix array construction**. In *Proceedings of the 13th International Conference on Automata, Languges and Programming,* Springer 2003.

33. Puglisi SJ, Smyth W, Turpin A: **The performance of linear time suffix sorting algorithms**. In *DCC '05: Proceedings of the Data Compression Conference*, Washington, DC, USA: IEEE Computer Society 2005:358–367.

34. Manzini G, Ferragina P: **Engineering a lightweight suffix array construction algorithm**. *Algorithmica* 2004, **40**:33–50.

35. Abouelhoda M, Kurtz S, Ohlebusch E: **Replacing suffix trees with enhanced suffix arrays**. *Journal of Discrete Algorithms* 2004, **2**:53–86.

36. Fischer J: **Wee LCP**. *Information Processing Letters* 2010, **110**(8-9):317–320.

37. Kasai T, Lee G, Arimura H, Arikawa S, Park K: **Linear-time longest-common-prefix computation in suffix arrays and its applications**. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching* 2001:181–192.

38. Beckstette M, Homann R, Giegerich R, Kurtz S: **Fast index based algorithms and software for matching position specific scoring matrices**. *BMC Bioinformatics* 2006, **7**:389.

39. Beckstette M, Homann R, Giegerich R, Kurtz S: **Significant speedup of database searches with HMMs by search space reduction with PSSM family models**. *Bioinformatics* 2009, **25**(24):3251–3258.

40. Abouelhoda MI, Ohlebusch E, Kurtz S: **Optimal exact string matching based on suffix arrays**. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval, Volume 2476,* Springer 2002:31–43.

41. de Bruijn N: **A combinatorial problem**. *Koninklijke Nederlandse Akademie v. Wetenschappen* 1946, **49**:758–764.

42. Gardner P, Giegerich R: **A comprehensive comparison of comparative RNA structure prediction approaches**. *BMC Bioinformatics* 2004, **5**(140).

43. Hofacker I, Fekete M, Stadler P: **Secondary structure prediction for aligned RNA sequences**. *Journal of Molecular Biology* 2002, **319**(5):1059–66.

44. Knudsen B, Hein J: **Pfold: RNA secondary structure prediction using stochastic context-free grammars**. *Nucl. Acids Res.* 2003, **31**(13):3423–8.

45. Hofacker I: **RNA consensus structure prediction with RNAalifold**. *Methods Mol Biol* 2007, **395**:527–544.

46. Bremges A, Schirmer S, Giegerich R: **Fine-tuning structural RNA alignments in the twilight zone**. *BMC Bioinformatics* 2010, **11**(222).

47. Torarinsson E, Havgaard J, Gorodkin J: **Multiple structural alignment and clustering of RNA sequences**. *Bioinformatics* 2007, **23**:926–932.

48. Harmanci A, Sharma G, Mathews D: **Efficient pairwise RNA structure prediction using probabilistic alignment constraints**. *BMC Bioinformatics* 2007, **8**(130).

49. Reeder J, Giegerich R: **Consensus shapes: an alternative to the Sankoff algorithm for RNA consensus structure prediction**. *Bioinformatics* 2005, **21**(17):3516–23.

50. Wilm A, Higgins D, Notredame C: **R-Coffee: a method for multiple alignment of non-coding RNA**. *Nucl. Acids Res.* 2008, **36**(9).

51. Abouelhoda M, Ohlebusch E: **Chaining algorithms for multiple genome comparison**. *J. Discrete Algorithms* 2005, **3**(2-4):321–341.

52. Cormen T, Leiserson C, Rivest R: *Introduction to algorithms*. Cambridge, MA: MIT Press 1990.

53. Altuvia S, Zhang A, Argaman L, Tiwari A, Storz G: **The Escherichia coli OxyS regulatory RNA represses fhlA translation by blocking ribosome binding**. *EMBO* 1998, **15**(20):6069–75.

54. Pollard K, Salama S, Lambert N, Lambot M, Coppens S, Pedersen J, Katzman S, King B, Onodera C, Siepel A, Kern A, Dehay C, Igel H, Ares M, Vanderhaeghen P, Haussler D: **An RNA gene expressed during cortical development evolved rapidly in humans**. *Nature* 2006, **443**(7108):167–172.

55. Schnattinger T, Ohlebusch E, Gog S: **Bidirectional search in a string with wavelet trees**. In *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching*, *Volume 6129*, Springer 2010:40–50.

Supplemental material for the paper: Structator: fast index-based search for RNA sequence-structure patterns

Fernando Meyer Stefan Kurtz Rolf Backofen Sebastian Will
Michael Beckstette

1 An example of bidirectional RSSP search

As an example of bidirectional search for RSSPs using affix arrays, we search for the RSSP $\mathcal{Q}$ in the sequence S of Figures 2 and 3, respectively, of the main document. We recall that $\mathcal{Q} = (P, R)$ with P = NNNUGCUNNN and R = `(((....)))` represents a stem-loop structure of length $m = 10$ and S = AUAGCUGCUGCUGCA has length 15. We start matching P in S by calling procedure *bidir-search* of Algorithm 2 as *bidir-search*$(\langle 0, 0 - [0..15], \mathsf{F}\rangle, 2, 3)$. That is, the algorithm matches the first position $P[3] = $ U of the loop region in left-to-right direction. Given that $\mathsf{X} = \mathsf{F}$ and $i < j$ (i.e. $0 < 15$) hold, it locates interval $v_x = \langle 0, 1 - [11..14], \mathsf{F}\rangle$ with $\overrightarrow{v_x} = $ U via binary search in the interval $0 - [0..15]$ of $\mathsf{suf_F}$. Analogously, the following recursive calls of *bidir-search* perform right c-extensions of $u = $ U $= P[3..3]$ with characters $P[4] = $ G, $P[5] = $ C, and $P[6] = $ U, by searching in the intervals $1 - [11..14]$, $2 - [12..14]$, and $3 - [12..14]$, respectively. After these extensions, the algorithm has located the affix-interval $v_x = \langle 0, 4 - [13..14], \mathsf{F}\rangle$ representing all occurrences of $\overrightarrow{v_x} = $ UGCU in S such that $\overrightarrow{v_x}$ matches $u = P[3..6]$. We set $v = v_x$. Next, the algorithm performs a right c-extension of u with the pairing position $c \in \varphi(P[7] = $ N$)$. Therefore, it enumerates all possible v_x such that $\overrightarrow{v_x} = \overrightarrow{v}d$ for some $d \in \varphi(c)$. We observe that $v_x = \langle 0, 5 - [13..14], \mathsf{F}\rangle$ with $\overrightarrow{v_x} = $ UGCUG is the only interval satisfying these conditions and conclude $d = $ G. As an additional structural constraint, matches to positions 2 and 7 of P shall form a base pair. To fulfill this constraint the algorithm first switches the search direction by locating the reverse interval v' of v_x. The left boundary of v' is determined with a lookup in table $\mathsf{aflk_F}$ as $\mathsf{aflk_F}[\mathsf{home_F}([13..14])] = 5$ and the right boundary as 6. Further, we set $\ell_{\mathsf{lcp}} = min\{\mathsf{lcp_F}[r] \mid 13 < r \leq 14\} = 6$ and calculate the context of v' as $6 - 5 = 1$. Hence, the reverse interval of v_x is determined as $v' = \langle 1, 6 - [5..6], \mathsf{R}\rangle$ with $\overrightarrow{v}' = $ UGCUG and we set $v = v'$. Now the only interval satisfying (1) $\overrightarrow{v_x} = e\overrightarrow{v}, e \in \varphi(P[2])$, and (2) the complementarity condition between positions 2 and 7 of P, as required by the structure string R, is the interval $v_x = \langle 1, 7 - [5..6], \mathsf{R}\rangle$ with $\overrightarrow{v_x} = $ CUGCUG representing occurrences of substrings matching $P[2..7]$. Observe that $\overrightarrow{v_x}[0] = $ C and $\overrightarrow{v_x}[5] = $ G can form a base pair as demanded by $R[2]$ and $R[7]$. Consequently, $\overrightarrow{v_x}$ matches $(P[2..7], R[2..7])$ and therefore we set $v = v_x$. In the next step the algorithm performs another left c-extension of $\overrightarrow{v}$ by some $c \in \varphi(P[1] = $ N$)$ leading to interval $v_x = \langle 1, 8 - [5..6], \mathsf{R}\rangle$ with $\overrightarrow{v_x} = $ GCUGCUG representing occurrences of substrings matching $P[1..7]$. We set $v = v_x$. To match a character $d \in \varphi(c)$ that is complementary to $\overrightarrow{v}[0] = $ G the algorithm performs a right c-extension of $\overrightarrow{v}$ using a character $c \in \varphi(P[8])$. Because the context of v is larger than zero, it consumes the context and remains in table $\mathsf{suf_R}$. That is, $\mathsf{X} = \mathsf{R}$. The resulting interval after performing the right c-extension is $v_x = \langle 0, 8 - [5..6], \mathsf{R}\rangle$ with $\overrightarrow{v_x} = $ GCUGCUGC. Observe that $\overrightarrow{v_x}[0] = $ G and $\overrightarrow{v_x}[7] = $ C can form a base pair and thus v_x represents occurrences of substrings of S matching $(P[1..8], R[1..8])$. We set $v = v_x$. The next operation is a left c-extension by some $c \in \varphi(P[0] = \mathrm{N})$. Hence, the algorithm enumerates all intervals v_x such that $\overrightarrow{v_x} = \overrightarrow{v}d, d \in \varphi(c)$. There are two intervals satisfying these conditions. Namely, $v_{x1} = \langle 0, 9 - [5..5], \mathsf{R}\rangle$ with $\overrightarrow{v_{x1}} =$ AGCUGCUGC and $v_{x2} = \langle 0, 9 - [6..6], \mathsf{R}\rangle$ with $\overrightarrow{v_{x2}} =$ UGCUGCUGC. We set $v_1 = v_{x1}$ and $v_2 = v_{x2}$ and continue by processing v_1, which represents occurrences of $\overrightarrow{v_1} =$ AGCUGCUGC in S. Because $\overrightarrow{v_1}$ is a unique substring of S, for the following right c-extension by some $c \in \varphi(P[9] = \mathrm{N})$ we can directly evaluate $S^{\mathrm{R}}[\mathsf{suf}_{\mathsf{R}}[5] - 1] = \mathrm{U}$. Bases $(\overrightarrow{v_1}[0] = A, U)$ are complementary, hence we set $v_x = \langle -1, 9 - [5..5], \mathsf{R}\rangle$ and observe that occurrences of substring $\overrightarrow{v_x} =$ AGCUGCUGCU of S match $(P[0..9], R[0..9])$ and that the boundaries of $\mathcal{Q}$ have been reached. With this, in the following recursion the algorithm reports a matching position of $\mathcal{Q}$ via a lookup in table $\mathsf{suf}_{\mathsf{R}}$ as $\mathsf{suf}_{\mathsf{R}}[5] + (-1) = 4 - 1 = 3$, where -1 is the context of v_x that has to be added to $\mathsf{suf}_{\mathsf{R}}[5]$. Note that, because $\mathsf{X} = \mathsf{R}$, 3 is a position in S^{R}. Now the algorithm backtracks to interval $\langle 0, 8 - [5..6], \mathsf{R}\rangle$ and continues to perform a right c-extension of interval v_2 by some $c \in \varphi(P[9])$. Again, $\overrightarrow{v_2} =$ UGCUGCUGC is a unique substring of S and we can directly evaluate $S^{\mathrm{R}}[\mathsf{suf}_{\mathsf{R}}[6] - 1] = \mathrm{A}$. Since bases $(\overrightarrow{v_2}[0] = U, A)$ can pair, we set $v_x = \langle -1, 9 - [6..6], \mathsf{R}\rangle$ with $\overrightarrow{v_x} =$ UGCUGCUGCA representing occurrences of substrings of S matching $(P[0..9], R[0..9])$. The boundaries of $\mathcal{Q}$ have been reached again and in the following recursion the algorithm reports another matching position of $\mathcal{Q}$, precisely $\mathsf{suf}_{\mathsf{R}}[6] + (-1) = 1 - 1 = 0$. There are no further intervals to process and the search ends. In summary, *bidir-search* has found two occurrences of $\mathcal{Q}$ in S.

2 Comparison of two implementations of bidirectional pattern search

We measured the speedup of *Structator* running in *BIDsearch* mode over *ONLsearch* and compared the results with previously reported measurements [1]. Because the implementation used by Strothmann [1] is not available (personal communication), we calculated relative speedups based on the absolute running times reported in [1]. We note that the measurements of [1] were performed on different hardware. This can, according to our experiments, significantly influence the performance of *BIDsearch*. See Table S1 for the results of the comparison of *BIDsearch* with the method of [1]. For a description of the used RSSPs see [1]. The search was performed in the genomes of *P. horikoshii* (GenBank Acc.: NC_000961, 1.7 MB) and *E. coli* (GenBank Acc.: AC_000091, 4.5 MB), which were also used in [1]. Additionally we included with *P. vampyrus* (GenBank Acc.: ABRP00000000, 1.9 GB) a larger eukaryotic genome in this experiment.

Surprisingly, with the RSSPs ACloop(5), ACloop(10), and ACloop(15) taken from [1], which describe a loop consisting of 5 (10 and 15) repetitions of AC the speedup of the affix array based method of [1] decreased with increasing loop length. This is a behavior which is opposite to our observations (see Figure 8 of the main document). We also noticed that *BIDsearch* obtained a higher speedup when searching for RSSP Hpin2 in *E. coli* than the method of [1] but not when searching in the smaller genome of *P. horikoshii*. This observation remains unclear and cannot be further investigated due to unavailability of the implementation used in [1].

3 A bidirectional search algorithm supporting variable-length RSSPs

Algorithm 2 of the main document matches fixed-length RSSPs. We here present an extension of it also capable of matching RSSPs with loop region allowing a variable number of additional extensions with ambiguous characters N to the left and to the right. In combination, also stem region of variable length is supported. We observe that this extended version is as efficient as the original algorithm supporting fixed-length RSSPs. Additional computation time is only required for the traversal of additional affix-intervals due to the increased sensitivity.

	P. horikoshii (1.7 MB)				*E. coli K12* (4.5 MB)				*P. vampyrus* (1.9 GB)			
RSSP	*ONL*	*BID*	*B*vs.*O*	*STR*	*ONL*	*BID*	*B*vs.*O*	*STR*	*ONL*	*BID*	*B*vs.*O*	*STR*
Hpin1	169.61	65.59	2.59	10.26	432.94	141.84	3.05	12.17	172,913.36	9,520.39	18.16	-
Hpin2	33.34	0.27	123.48	155	88.61	0.45	196.91	99.25	34,702.63	48.85	710.39	-
Hloop(5)	214.8	166.94	1.29	14.6	552.67	372.57	1.48	18.09	219,547.76	23,958.41	9.16	-
Hloop(10)	331.96	1,412.64	0.23	2.13	842.32	3,235.11	0.26	2.43	335,928.97	248,711.65	1.35	-
ACloop(5)	59.07	4.43	13.33	182	152.87	9.91	15.43	815	64,053.16	825.79	77.57	-
ACloop(10)	58.71	1.37	42.85	4	152.12	3.45	44.09	7.24	64,136.82	391.56	163.8	-
ACloop(15)	58.67	0.89	65.92	1.3	152.01	1.86	81.73	1.38	64,199.98	278.76	230.31	-

Table S1: Comparison of speedup of *Structator*'s *BIDsearch* over *ONLsearch* (column *B*vs.*O*) and the speedup of affix array based search over searching on the plain text as reported in [1] (column *STR*). The respective search times of *BIDsearch* (column *BID*) and *ONLsearch* (column *ONL*) are shown in milliseconds. For *P. vampyrus* only measurements for *Structator* are available.

Before describing the algorithm, we define this extension of RSSPs. A *variable-length RSSP* $\mathcal{Q}$ consists of an RSSP (P, R) and parameters *maxleftloopextent (mllex)*, *maxrightloopextent (mrlex)*, and *maxstemlength (msl)*. *mllex* and *mrlex* denote the maximum number of respective left and right extensions of the loop region specified in R and msl denotes the maximum number of base pairs in the stem. The minimum length of occurrences of $\mathcal{Q}$ is $m = |P| = |R|$. For examples of variable-length RSSPs, see Figure 12 (E) until (H) of the main document.

To keep the code simple, we split the original algorithm into two procedures. (i) First the loop region of a given variable-length RSSP $\mathcal{Q}$ is matched with procedure *bidir-search-loop* (see Algorithm 3, Figure S1). (ii) Next, the stem region is matched with procedure *bidir-search-stem* (see Algorithm 4, Figure S2). Note that *bidir-search-stem* is very similar to Algorithm 2 of the main document. Prior to the search for $\mathcal{Q}$, the following variables are set: *loopstart*, *minloopstart*, *loopend*, *maxloopend*, *minbps*, and *maxbps*. These variables store the following information. *loopstart* (*loopend*) stores the position of the base occurring in the left-most (right-most) position of the loop described by the structure string R in 5' to 3' direction, *minloopstart* $=$ *loopstart* $-$ *mllex*, *maxloopend* $=$ *loopend* $+$ *mrlex*, and *minbps* (*maxbps* $=$ *msl*) is the minimum (maximum) number of base pairs occurring in $\mathcal{Q}$. It holds: *minloopstart* $\leq$ *loopstart* $\leq$ *loopend* $\leq$ *maxloopend*. Note that *minloopstart* can be negative. As an example, let $R =$ `(((....)))`, *mllex* $= 4$, and *mrlex* $= 1$. Then *loopstart* $= 3$, *minloopstart* $= -1$, *loopend* $= 6$, *maxloopend* $= 7$, and *minbps* $= 3$. To match $\mathcal{Q}$, procedure *bidir-search-loop* is initially called as *bidir-search-loop*$(\langle 0, 0 - [0..n], \mathsf{F}\rangle, r_0 - 1, r_0, \text{true})$, where $\langle 0, 0 - [0..n], \mathsf{F}\rangle$ is an affix-interval, r_0 is any position in the loop region of $\mathcal{Q}$, and parameter true states that the pattern can be extended to the right. Procedure *bidir-search-loop* makes a call to *bidir-search-stem* whenever substrings of minimum length *loopend* $-$ *loopstart* $+ 1$ matching the loop in the searched database are found. If $\mathcal{Q}$ has no base pairs, i.e. $msl = 0$, it instead immediately reports the matching positions. The call to *bidir-search-stem* is made as *bidir-search-stem*$(v', loopstart - 1, loopend + 1, 0)$, where v' is the affix-interval representing all occurrences of substring $\overrightarrow{v}'$ in the searched database matching the loop region of $\mathcal{Q}$, positions *loopstart* $- 1$ and *loopend* $+ 1$ denote the inner-most base pair $($*loopstart* $- 1$, *loopend* $+ 1)$ of the pattern, and 0 is the number of currently matched base pairs. Procedure *bidir-search-stem* reports matching positions of $\mathcal{Q}$ whenever the boundaries of the RSSP are reached or *minbps* $<$ *bpcount* $<$ *maxbps* holds.

References

[1] D. Strothmann. The affix array data structure and its applications to RNA secondary structure analysis. *Theor. Comput. Sci.*, 389(1-2):278–294, 2007.

Algorithm 3: *bidir-search-loop*(affix-interval $v = \langle k, \ell - [i..j], X\rangle$, pos r, pos r', *allowrightext*)

```
1  if $r' \leq maxloopend$ and $allowrightext$ = true then
       // perform right extension
2      if $r' > loopend$ then
3          $chr' =$ 'N'
4      else
5          $chr' = P[r']$
6      end
7      foreach $v'$ such that $d \in \varphi(chr')$ and $\overrightarrow{v}' = \overrightarrow{v}d$ do
8          if $r < loopstart$ and $r' + 1 > loopend$ then
9              if $msl = 0$ then // if entire pattern is single stranded
10                 report match at positions $\mathsf{suf}_X[i] + k, ..., \mathsf{suf}_X[j] + k$
11                 return
12             else // otherwise loop of length $r' - r + 1$ was matched
                   // so extend stem region
13                 $bidir\text{-}search\text{-}stem(v', loopstart - 1, loopend + 1, 0)$
14             end
15         end
16         if $r' + 1 \leq maxloopend$ then
17             $bidir\text{-}search\text{-}loop(v', r, r' + 1, \text{true})$
18         end
19         if $r' + 1 > loopend$ then
20             $bidir\text{-}search\text{-}loop(v', r, r' + 1, \text{false})$
21         end
22     end
23 else if $r \geq minloopstart$ then
       // perform left extension
24     if $r < loopstart$ then
25         $chr =$ 'N'
26     else
27         $chr = P[r]$
28     end
29     foreach $v'$ such that $d \in \varphi(chr)$ and $\overrightarrow{v}' = d\overrightarrow{v}$ do
30         if $r - 1 < loopstart$ and $r' > loopend$ then
31             if $msl = 0$ then // if entire pattern is single stranded
32                 report match at positions $\mathsf{suf}_X[i] + k, ..., \mathsf{suf}_X[j] + k$
33                 return
34             else // otherwise loop of length $r' - r + 1$ was matched
                   // so extend stem region
35                 $bidir\text{-}search\text{-}stem(v', loopstart - 1, loopend + 1, 0)$
36             end
37         end
38         $bidir\text{-}search\text{-}loop(v', r - 1, r', allowrightext)$
39     end
40 end
```

Figure S1: Bidirectional recursive matching of the loop region of a variable-length RSSP using an affix array. Procedure *bidir-search-loop* searches for an RSSP (P, R) defined with additional variables *maxleftloopextent (mllex)* and *maxrightloopextent (mrlex)* denoting the maximum number of left and right extensions of the loop specified in R, respectively, and *maxstemlength (msl)* denoting the maximum number of base pairs. Used variables *loopstart*, *minloopstart*, *loopend*, and *maxloopend* are preset according to structure string R, *mllex*, and *mrlex* (see text). *bidir-search-loop* calls procedure *bidir-search-stem* (see Algorithm 4) whenever substrings of minimum length *loopend* $-$ *loopstart* $+ 1$ matching the loop are found.

Algorithm 4: *bidir-search-stem*(affix-interval $v = \langle k, \ell - [i..j], X\rangle$, pos r, pos r', *bpcount*)

```
if $(r < 0$ and $r' \geq m)$ or $(minbps < bpcount < maxbps)$ then
    report match at positions $\mathsf{suf}_X[i] + k, ..., \mathsf{suf}_X[j] + k$
end
if $(minbps < bpcount < maxbps$ or $(r \geq 0$ and $r' < m$ and $R[r] =$ '(' and $R[r'] =$ ')') then
    if $minbps < bpcount < maxbps$ then
        $chr\ =$ 'N'
        $chr' =$ 'N'
    else
        $chr\ = P[r]$
        $chr' = P[r']$
    end
    if $X = \mathrm{R}$ then
        // perform left extension first
        foreach $v'$ such that $d \in \varphi(chr)$ and $\overrightarrow{v}' = d\overrightarrow{v}$ do
            foreach $v''$ such that $e \in \varphi(chr')$ and $(d, e)$ complementary and $\overrightarrow{v}'' = \overrightarrow{v}'e$ do
                bidir-search-stem($v''$, $r - 1$, $r' + 1$, $bpcount + 1$)
            end
        end
    else
        // perform right extension first
        foreach $v'$ such that $e \in \varphi(chr')$ and $\overrightarrow{v}' = \overrightarrow{v}e$ do
            foreach $v''$ such that $d \in \varphi(chr)$ and $(d, e)$ complementary and $\overrightarrow{v}'' = d\overrightarrow{v}'$ do
                bidir-search-stem($v''$, $r - 1$, $r' + 1$, $bpcount + 1$)
            end
        end
    end
else if $r' < m$ and $R[r'] =$ '.' and $(X = \mathrm{F}$ or $r < 0$ or $R[r] \neq$ '.') then
    foreach $v'$ such that $d \in \varphi(P[r'])$ and $\overrightarrow{v}' = \overrightarrow{v}d$ do
        bidir-search-stem($v'$, $r$, $r' + 1$, $bpcount$)
    end
else if $r \geq 0$ and $R[r] =$ '.' then
    foreach $v'$ such that $d \in \varphi(P[r])$ and $\overrightarrow{v}' = d\overrightarrow{v}$ do
        bidir-search-stem($v'$, $r - 1$, $r'$, $bpcount$)
    end
end
```

Figure S2: Bidirectional recursive matching of the stem region of a variable-length RSSP using an affix array. Procedure *bidir-search-stem* is called by procedure *bidir-search-loop* (see Algorithm 3) and extends substrings $\overrightarrow{v}$ matching the loop region of the RSSP (P, R) to substrings matching also the stem. Used variables *minbps* and *maxbps* are preset according to structure string R and variable *maxstemlength (msl)* (see text).

[2] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees. In *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching*, volume 6129, pages 40–50. Springer, 2010.

[3] K. Darty, A. Denise, and Y. Ponty. VARNA: Interactive drawing and editing of the RNA seondary structure. *Bioinformatics*, 25(15):1974–1975, 2009.

Organism	Genome size (n)	suf_F (4n)	lcp_F (n)	lcpe_F	aflk_F (4n)	suf_R (4n)	lcp_R (n)	lcpe_R	aflk_R (4n)
C.elegans	100.29	401.14	100.29	6.29	401.14	401.14	100.29	6.29	401.14
A.thaliana	119.67	478.67	119.67	8.85	478.67	478.67	119.67	8.85	478.67
D.melanogaster	168.74	674.95	168.74	94.34	674.95	674.95	168.74	94.34	674.95
C.intestinalis	173.52	694.02	173.50	28.03	694.02	694.02	173.50	28.03	694.02
O.sativa	374.33	1,497.33	374.33	71.05	1,497.33	1,497.33	374.33	71.05	1,497.33
M.gallopavo	1,087.50	4,349.99	1,087.50	2.01	4,349.99	4,349.99	1,087.50	2.01	4,349.99
G.gallus	1,108.48	4,433.93	1,108.48	98.86	4,433.93	4,433.93	1,108.48	98.86	4,433.93
D.rerio	1,481.32	5,925.08	1,481.27	457.26	5,925.08	5,925.08	1,481.27	457.26	5,925.08
X.tropicalis	1,510.98	6,043.63	1,510.91	310.89	6,043.63	6,043.63	1,510.91	310.89	6,043.63
P.vampyrus	1,999.71	7,998.82	1,999.71	170.84	7,998.82	7,998.82	1,999.71	170.84	7,998.82

Table S2: Sizes in megabytes of the different tables the affix array consists of for the genomes used in Experiment 1. lcpe_F and lcpe_R are the exception tables storing entries with value larger than 255 which cannot be stored in tables lcp_F and lcp_R, respectively. In tables lcpe_F and lcpe_R, each entry consumes 8 bytes.

Organism	Genome size	BWI
C.elegans	100.29	157.96
A.thaliana	119.67	188.59
D.melanogaster	168.74	295.37
C.intestinalis	173.52	279.83
O.sativa	374.33	602.21
M.gallopavo	1,087.50	1,800.88
G.gallus	1,108.48	1,757.84
D.rerio	1,481.32	2,424.81
X.tropicalis	1,510.98	2,309.24
P.vampyrus	1,999.71	3,282.55

Table S3: Size in megabytes of the bidirectional wavelet index (BWI) [2] for different genomes.

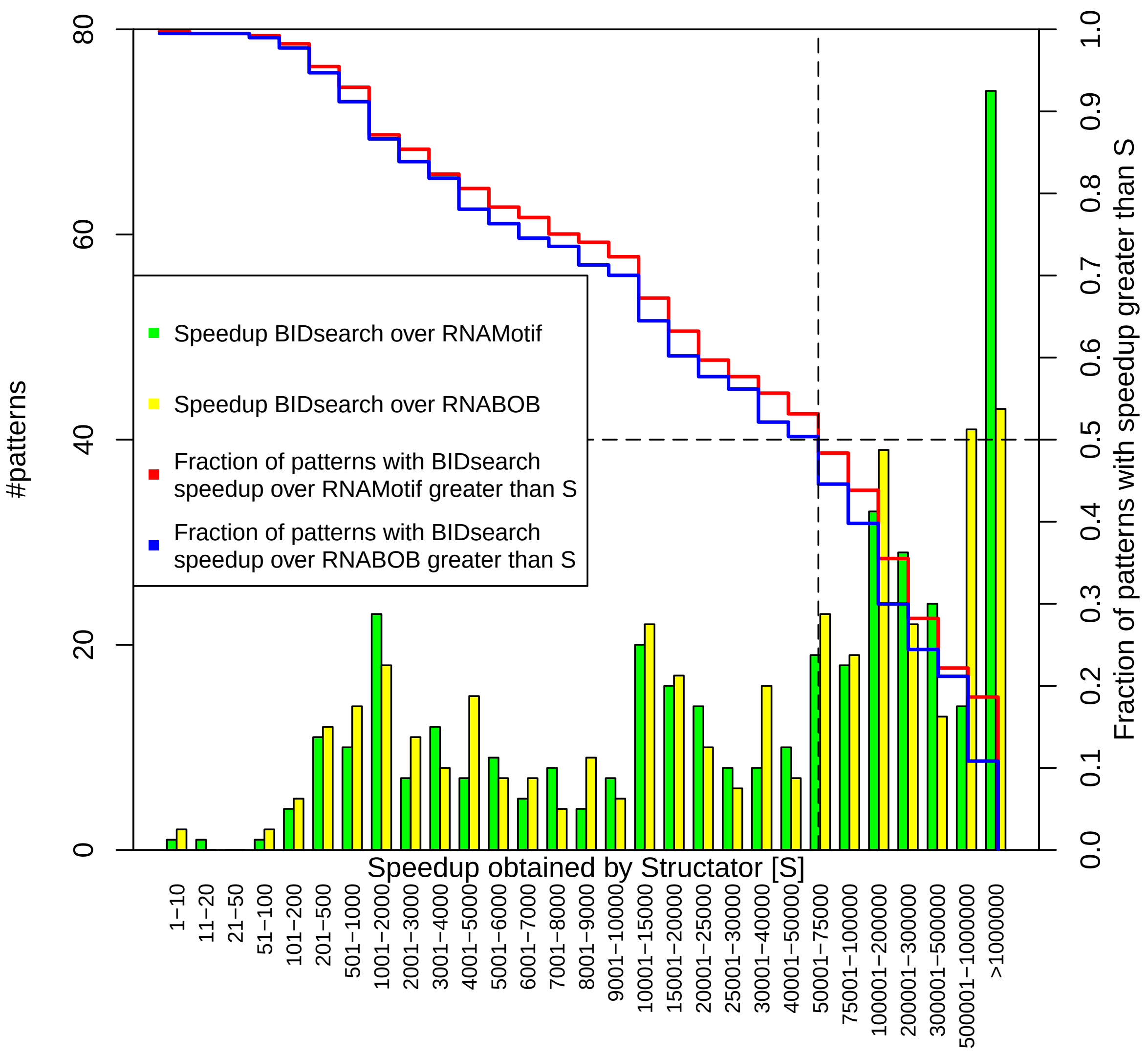
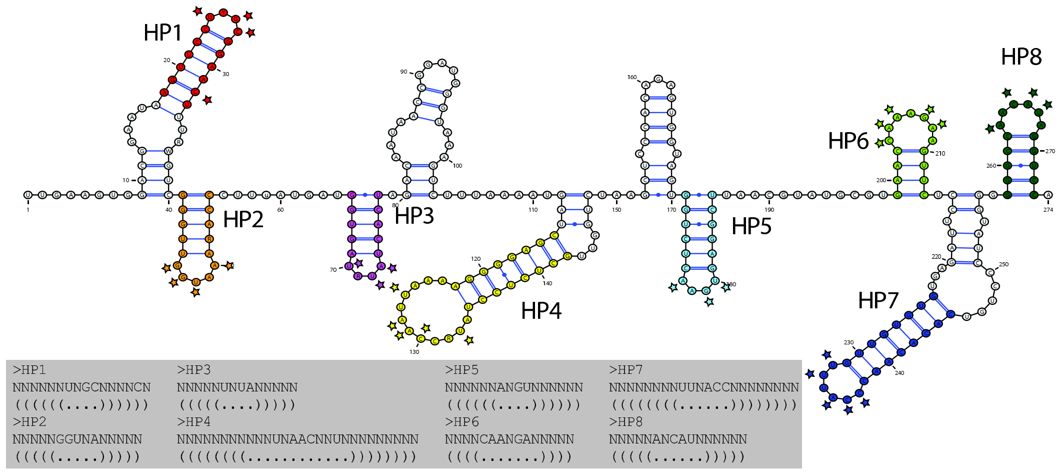
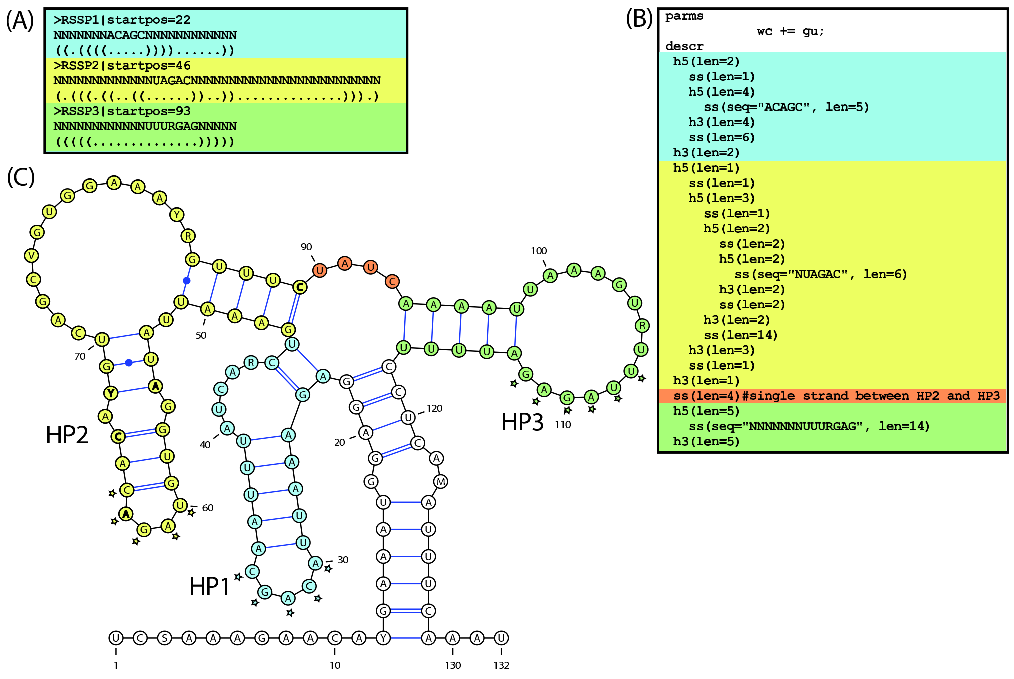


Figure S3: Distribution of speedup factors of *BIDsearch* over *RNABOB* (yellow) and *RNAMotif* (green) when searching for 397 RSSPs in RFAM10 consisting of ~ 622 megabases of RNA sequence data. The red and blue curves show the values of one minus the empirical cumulative distribution function of the speedup factors distributions. That is, for a given speedup factor S they show the fraction of RSSPs for which *BIDsearch* obtained a speedup greater than S over *RNAMotif* (red curve) and *RNABOB* (blue curve), respectively. We observed that *BIDsearch* is more than $50{,}000$ times faster than *RNABOB* and *RNAMotif* for the majority of the patterns (see intersection point of dashed lines). Moreover, the total search time required by *BIDsearch* is dominated by only a small number of patterns describing large unconserved loop regions.



```
>HP1              >HP3                          >HP5              >HP7
NNNNNNUNGCNNNNCN  NNNNNUNUANNNNN                NNNNNNANGUNNNNNN  NNNNNNNNUUNACCNNNNNNNN
((((((....))))))  (((((....)))))                ((((((....))))))  ((((((((......))))))))
>HP2              >HP4                          >HP6              >HP8
NNNNNGGUNANNNNN   NNNNNNNNNNNUNAACNNUNNNNNNNNN  NNNNCAANGANNNNN   NNNNNANCAUNNNNNN
(((((.....)))))   ((((((((............))))))))  ((((.......))))   (((((......)))))
```

Figure S4: Consensus secondary structure of the CTV_rep_sig family (RFAM Acc.: RF00193) visualized with the *VARNA* program [3] and SSD in *Structator* syntax describing this family. The 8 given RSSPs correspond to the colored stem-loops HP1 - HP8. Positions at which sequence information is used in the descriptor are marked with an asterisk.



(A)

```
>RSSP1|startpos=22
NNNNNNNACAGCNNNNNNNNNNNN
((.((((.....))))......))
>RSSP2|startpos=46
NNNNNNNNNNNNNNUAGACNNNNNNNNNNNNNNNNNNNNNNNNNN
(.(((.((..((......))..))...............))).)
>RSSP3|startpos=93
NNNNNNNNNNNNNUUURGAGNNNNN
(((((...............)))))
```

(B)

```
parms
            wc += gu;
descr
 h5(len=2)
   ss(len=1)
   h5(len=4)
     ss(seq="ACAGC", len=5)
   h3(len=4)
   ss(len=6)
 h3(len=2)
 h5(len=1)
   ss(len=1)
   h5(len=3)
     ss(len=1)
     h5(len=2)
       ss(len=2)
       h5(len=2)
         ss(seq="NUAGAC", len=6)
       h3(len=2)
       ss(len=2)
     h3(len=2)
     ss(len=14)
   h3(len=3)
   ss(len=1)
 h3(len=1)
 ss(len=4)#single strand between HP2 and HP3
 h5(len=5)
   ss(seq="NNNNNNNUUURGAG", len=14)
 h3(len=5)
```

Figure S5: (A) SSD for HAR1F RNA family consisting of RSSP1, RSSP2, and RSSP3 in *Structator* syntax. RSSPs were built from stem-loops HP1, HP2, and HP3 shown in (C). (B) *RNAMotif* descriptor for the same structural elements. Secondary structure drawing shown in (C) was generated with *VARNA* [3].

Acc.	#Matching chains	#TP	#FP	#FN	Sensitivity	Specificity	Accuracy	Precision	#RSSPs	Min.chain length	$T_{BIDsearch}$[sec]	$T_{ONLsearch}$[sec]	Speedup	$T_{chaining}$[sec]
RF00044	8	8	0	0	1.000	1.000	1.000	1.000	8	2	0.964	117.359	121.742	0.001
RF00193	37	37	0	0	1.000	1.000	1.000	1.000	8	5	1.220	140.681	115.312	0.063
RF00126	106	106	0	1	0.991	1.000	1.000	1.000	6	2	1.032	128.476	124.492	0.000
RF00503	78	78	0	2	0.975	1.000	1.000	1.000	10	2	1.084	164.866	152.090	0.002
RF00209	1,511	1,493	18	58	0.963	1.000	1.000	0.988	9	2	1.056	129.372	122.511	0.006
RF00625	24	22	2	1	0.957	1.000	1.000	0.917	5	3	3.304	102.066	30.892	0.656
RF00061	6,211	6,211	0	285	0.956	1.000	1.000	1.000	7	4	1.188	119.239	100.370	0.032
RF00224	21	21	0	1	0.955	1.000	1.000	1.000	10	3	1.508	202.661	134.391	0.138
RF00084	111	111	0	7	0.941	1.000	1.000	1.000	4	2	1.180	78.669	66.669	0.050
RF00372	42	42	0	3	0.933	1.000	1.000	1.000	7	3	1.092	104.663	95.845	0.007
RF00115	58	58	0	5	0.921	1.000	1.000	1.000	9	4	1.128	167.962	148.902	0.024
RF00488	24	24	0	3	0.889	1.000	1.000	1.000	6	4	1.084	94.938	87.581	0.043
RF00294	44	44	0	9	0.830	1.000	1.000	1.000	12	3	1.124	164.814	146.632	0.008
RF00210	345	345	0	72	0.827	1.000	1.000	1.000	14	3	1.308	206.133	157.594	0.104
RF00228	348	346	2	79	0.814	1.000	1.000	0.994	13	2	1.048	225.982	215.632	0.006
RF00036	18,312	18,312	0	4,452	0.804	1.000	0.999	1.000	16	3	1.464	224.778	153.537	0.145
RF00549	39	38	1	10	0.792	1.000	1.000	0.974	10	4	1.584	154.382	97.463	0.142
RF00448	11	11	0	3	0.786	1.000	1.000	1.000	7	4	1.000	102.730	102.730	0.002
RF00177	584,748	582,839	1,909	179,250	0.765	0.999	0.946	0.997	13	3	11.004	221.798	20.156	2.414
RF00101	142	142	0	45	0.759	1.000	1.000	1.000	6	3	1.000	119.407	119.407	0.004
RF00166	54	54	0	18	0.750	1.000	1.000	1.000	8	3	1.068	127.872	119.730	0.009
RF00018	278	272	6	96	0.739	1.000	1.000	0.978	11	5	3.944	212.133	53.786	0.666
RF00252	26	26	0	10	0.722	1.000	1.000	1.000	10	3	1.260	143.709	114.055	0.057
RF00547	39	39	0	18	0.684	1.000	1.000	1.000	14	3	2.604	221.458	85.045	0.452
RF00011	355	353	2	185	0.656	1.000	1.000	0.994	10	4	2.988	183.923	61.554	0.582
RF00010	2,478	2,402	76	1,679	0.589	1.000	0.999	0.969	12	5	6.212	187.616	30.202	1.548
RF00449	33	32	1	26	0.552	1.000	1.000	0.970	9	3	1.308	154.726	118.292	0.073
RF00040	92	92	0	82	0.529	1.000	1.000	1.000	9	4	1.248	153.410	122.925	0.050
RF00023	1,362	1,362	0	1,699	0.445	1.000	0.999	1.000	11	3	2.076	193.740	93.324	0.229
RF00229	1,257	1,256	1	1,637	0.434	1.000	0.999	0.999	11	3	1.472	193.168	131.228	0.139
RF00222	26	26	0	35	0.426	1.000	1.000	1.000	12	3	1.148	201.557	175.572	0.025
RF00459	223	215	8	341	0.387	1.000	1.000	0.964	7	2	4.776	221.002	46.273	0.012
RF00028	10,647	10,229	418	28,820	0.262	1.000	0.991	0.961	13	2	1.476	203.889	138.136	0.075
RF00261	21	21	0	65	0.244	1.000	1.000	1.000	8	4	1.552	171.063	110.221	0.130
RF00373	82	75	7	247	0.233	1.000	1.000	0.915	8	4	1.692	143.645	84.897	0.166
RF00230	2,059	1,753	306	6,507	0.212	1.000	0.998	0.851	8	3	39.006	220.410	5.651	0.471
RF00226	18	18	0	73	0.198	1.000	1.000	1.000	7	4	2.664	108.687	40.798	0.449
RF00009	136	111	25	455	0.196	1.000	1.000	0.816	11	3	3.260	190.164	58.333	0.480
RF00629	6	6	0	25	0.194	1.000	1.000	1.000	8	4	1.816	153.526	84.541	0.248
RF00030	20	20	0	476	0.040	1.000	1.000	1.000	9	5	10.632	175.559	16.512	2.427
RF00100	614	614	0	15,042	0.039	1.000	0.995	1.000	13	7	1.240	198.652	160.203	0.065
RF00004	257	257	0	7,252	0.034	1.000	0.998	1.000	8	4	1.320	128.812	97.585	0.034
Average(∅):					**0.629**	**1.000**	**0.998**	**0.983**	**9.45**	**3.38**	**3.100**	**163.330**	**101.500**	**0.29**
Total(Σ):									**397**		**130.13**	**6,859.7**		**12.236**

Table S4: Results of *Structator* searches on RFAM10 ($1{,}446$ families; $3{,}192{,}599$ sequences) using SSDs describing 42 Rfam families. The manually compiled SSDs used in this experiment are available on the *Structator* website. They were designed to be highly specific and consist of 397 RSSPs in total with an average of 9.45 RSSPs per SSD. These are the same 397 RSSPs used in section "Searching large sequence databases" in the main document. Columns 2, 3, 4, and 5 show the number of sequences containing high-scoring global chains, the numbers of true positives (TP), false positives (FP), and false negatives (FN), respectively. Sensitivity is computed as $\frac{\#TP}{\#TP+\#FN}$, specificity as $\frac{\#TN}{\#TN+\#FP}$, accuracy as $\frac{\#TP+\#TN}{\#TP+\#FP+\#FN+\#TN}$, and precision as $\frac{\#TP}{\#TP+\#FP}$. Observe that these values strongly depend on the used SSD. The number of RSSPs constituting an SSD is given in column 10. Column 11 shows the minimal required length of a chain to be considered a matching chain. Total running times of *Structator* operating in *BIDsearch* and *ONLsearch* mode are given in columns 12 and 13, respectively. Column 14 shows *BIDsearch*'s speedups over *ONLsearch*. The running time required for chaining of RSSP matches is listed in column 15. Observe that the sum of running times does not match the times needed for searching with the 397 single RSSPs reported in the main document because here each SSD was searched using a separate *Structator* program call.