View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Springer - Publisher Connector

Meyer *et al. BMC Bioinformatics* 2013, **14**:226
http://www.biomedcentral.com/1471-2105/14/226

BMC Bioinformatics

METHODOLOGY ARTICLE Open Access

Fast online and index-based algorithms for approximate search of RNA sequence-structure patterns

Fernando Meyer, Stefan Kurtz and Michael Beckstette*

Abstract

Background: It is well known that the search for homologous RNAs is more effective if both sequence and structure information is incorporated into the search. However, current tools for searching with RNA sequence-structure patterns cannot fully handle mutations occurring on both these levels or are simply not fast enough for searching large sequence databases because of the high computational costs of the underlying sequence-structure alignment problem.

Results: We present new fast index-based and online algorithms for approximate matching of RNA sequence-structure patterns supporting a full set of edit operations on single bases and base pairs. Our methods efficiently compute semi-global alignments of structural RNA patterns and substrings of the target sequence whose costs satisfy a user-defined sequence-structure edit distance threshold. For this purpose, we introduce a new computing scheme to optimally reuse the entries of the required dynamic programming matrices for all substrings and combine it with a technique for avoiding the alignment computation of non-matching substrings. Our new index-based methods exploit suffix arrays preprocessed from the target database and achieve running times that are sublinear in the size of the searched sequences. To support the description of RNA molecules that fold into complex secondary structures with multiple ordered sequence-structure patterns, we use fast algorithms for the local or global chaining of approximate sequence-structure pattern matches. The chaining step removes spurious matches from the set of intermediate results, in particular of patterns with little specificity. In benchmark experiments on the Rfam database, our improved online algorithm is faster than the best previous method by up to factor 45. Our best new index-based algorithm achieves a speedup of factor 560.

Conclusions: The presented methods achieve considerable speedups compared to the best previous method. This, together with the expected sublinear running time of the presented index-based algorithms, allows for the first time approximate matching of RNA sequence-structure patterns in large sequence databases. Beyond the algorithmic contributions, we provide with *RaligNAtor* a robust and well documented open-source software package implementing the algorithms presented in this manuscript. The *RaligNAtor* software is available at http://www.zbh.uni-hamburg.de/ralignator.

Background

Due to their participation in several important molecular-biological processes, ranging from passive carriers of genetic information (tRNAs) over regulatory functions (microRNAs) to protein-like catalytic activities (Riboswitsches), non-coding RNAs (ncRNAs) are of central research interest in molceular biology [1]. NcRNAs, although synthesized as single-stranded molecules, present surprising complexity by being able to base pair with themselves and fold into numerous different structures. It is to a large extent the structure that allows them to interact with other molecules and hence to carry out various biological functions. This can also be observed in families of functionally related ncRNAs like the ones compiled in the Rfam database [2]. Here members of a family often share only few sequence features,

*Correspondence: beckstette@zbh.uni-hamburg.de
Center for Bioinformatics, University of Hamburg, Bundesstrasse 43, Hamburg 20146, Germany

BioMed Central

© 2013 Meyer et al.; licensee BioMed Central Ltd. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

but share by far more specific structural and functional properties. Consequently, methods for effective RNA homology search (i.e. finding new members of an RNA family) cannot rely on sequence similarity alone, but also have to take structural similarity into account.

In this paper, we address the problem of searching nucleotide databases for occurrences of RNA family members. Since for this task it is not sufficient to rely on pure sequence alignment, we briefly review search methods that employ sequence and structure information.

There exist various general sequence-structure alignment tools which determine structural similarities that are too diverse to be alignable at the sequence level. Such tools can roughly be divided into two classes. The first class consists of tools that align RNAs with given structures or determine a common structure during the alignment process. Tools like *MARNA* [3] and *RNAforester* [4] require an *a priori* known secondary structure for both input RNAs. However, they suffer from the low quality of secondary structure prediction. Addressing this problem, other tools implement variations of the Sankoff algorithm [5], which provides a general but computationally demanding solution to the problem of simultaneously computing an alignment and the common secondary structure of the two aligned sequences. Unfortunately, even tools with improved running times using variations of this algorithm (*LocARNA* [6], *Foldalign* [7,8], *Dynalign* [9,10]) or heuristics [11] are simply not fast enough for rapid searches in large nucleotide databases. Hence, in a second class we identify more specialized tools for searching RNA families in nucleotide databases. These tools use a model or motif descriptors (i.e. patterns) defining consensus sequence and secondary structure properties of the families to be searched for. For example, *Infernal* [12] and *RSEARCH* [13] infer a covariance model from a given multiple sequence alignment annotated with structure information. This model can then be used to search sequence databases for new family members. Another tool, *ERPIN* [14] is also based on automatically generated statistical secondary profiles. Although being very sensitive in RNA homology search, in particular *Infernal* and *RSEARCH* suffer from high computational demands. An alternative are tools like *RNAMotif* [15], *RNAMOT* [16], *RNABOB* [17], *RNAMST* [18], *PatScan* [19], *PatSearch* [20], or *Palingol* [21]. These methods use user-defined motif descriptors created from *a priori* knowledge about the secondary structure of the described RNA family. Another tool, *Locomotif* [22], generates a thermodynamic matcher program from a pattern drawn interactively by the user via a graphical interface. Although these tools based on motif descriptors are faster than the previously mentioned tools, they have a running time that scales at least linearly with the size of the target sequence database. This makes their application to large databases challenging. Previously, we addressed this problem by presenting *Structator* [23], an ultra fast index-based bidirectional matching tool that achieves sublinear running time by exploiting base pair complementarity constraints for search space reduction.

Apart from running time constraints, another major disadvantage of all current tools that search for sequence-structure patterns is their limited capacity to find approximate matches to the patterns. Although variability in length of pattern elements is often allowed, this is constrained to certain pattern positions that must be specified by the user. This limitation also holds for our *Structator* tool. Also, variations (insertions, deletions, or replacements) in the sequence that lead to small structural changes, such as the breaking of a base pair, are not supported. This often hampers the creation of patterns that are specific but generalized enough to match all family members. An algorithm presented in [24] only partially alleviates this problem by finding approximate matches of a helix in a genome allowing edit operations on single bases, but not on the structure.

To overcome these issues, we present new fast index-based and online algorithms for approximate matching of sequence-structure patterns, all implemented in an easy-to-use software package. Given one or more patterns describing any (branching, non-crossing) RNA secondary structure, our algorithms compute alignments of the complete patterns to substrings of the target sequence, i.e. semi-global alignments, taking sequence and structure into account. For this, they apply a full set of edit operations on single bases and base pairs. Matches are reported for alignments whose sequence-structure edit cost and number of insertions and deletions do not exceed user-defined thresholds. Our most basic algorithm is a scanning variant of the dynamic programming algorithm for global pairwise sequence-structure alignment of Jiang *et al.* [25], for which no implementation was available. Because its running time is too large for database searches on a large scale, we present accelerated online and index-based algorithms. All our new algorithms profit from a new computing scheme to optimally reuse the required dynamic programming matrices and a technique to save computation time by determining as early as possible whether a substring of the target sequence can contain a match. In addition, our index-based algorithms employ the suffix array data structure compiled from the search space. This further reduces the running time.

As in [23], we also support the description of an RNA molecule by multiple ordered sequence-structure patterns. In this way, the molecule's secondary structure is decomposed into a sequence of substructures described by independent sequence-structure patterns. These patterns are efficiently aligned to the target sequences using

one of our new algorithms and the results are combined with fast global and local chaining algorithms [23,26]. This allows a better balancing of running time, sensitivity, and specificity compared to searching with a single long pattern describing the complete sequence and secondary structure.

Before we describe our algorithms, we formalize the approximate search problem with the involved sequence-structure edit operations. Then we present, step by step, two efficient online and two index-based matching algorithms. We proceed with a short review of the approach for computing chains of matches. Finally, we present several benchmark experiments.

Methods

Preliminaries

An RNA *sequence* S of length $n = |S|$ over the set of bases $\mathcal{A} = \{\text{A, C, G, U}\}$ is a juxtaposition of n bases from $\mathcal{A}$. $S[i]$, $1 \leq i \leq n$, denotes the *base 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 $S = S[1]S[2]\ldots S[n]$ and $1 \leq i \leq j \leq n$, $S[i..j]$ denotes the *substring* $S[i]S[i+1]\ldots S[j]$ of S. 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. For $1 \leq k \leq n$, S_k denotes the k–th suffix of S. For stating the space requirements of our index structures, we assume that $|S| < 2^{32}$, so that sequence positions and lengths can be stored in 4 bytes.

The secondary structure of an RNA molecule is formed by Watson-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 $(c, d) \in \mathcal{C} = \{(\text{A, U}), (\text{U, A}), (\text{C, G}), (\text{G, C}), (\text{G, U}), (\text{U, G})\}$. If two bases a and b form a base pair we also say that there exists an *arc* between a and b. A *non-crossing RNA structure* $\widehat{R}$ *of length* m is a set of *base pairs* (i, j), $1 \leq i < j \leq m$, stating that the base at position i pairs with the base at position j, such that for all $(i, j), (i', j') \in \widehat{R}$: $i < i' < j' < j$ or $i' < i < j < j'$ or $i < j < i' < j'$ or $i' < j' < i < j$. A standard notation for $\widehat{R}$ is a *structure string* R over the alphabet $\{., (,)\}$ such that for each base pair $(i, j) \in \widehat{R}$, $R[i] =$ (and $R[j] =$), and $R[r] =$. for positions r, $1 \leq r \leq m$, that do not occur in any base pair of $\widehat{R}$, i.e. $r \neq i$ and $r \neq j$ for all $(i, j) \in \widehat{R}$.

Let $\Phi = \{\text{R, Y, M, K, W, S, B, D, H, V, N}\}$ be a set of characters. According to the IUPAC definition, each character in Φ denotes a specific character class $\varphi(x) \subseteq \mathcal{A}$. Each character $x \in \mathcal{A}$ can be seen as a character class $\varphi(x) = \{x\}$ of exactly one element. A *sequence pattern* is a sequence $P \in (\mathcal{A} \cup \Phi)^*$. 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. With $\mathcal{Q}[i..j]$ we denote the RSSP region $(P[i..j], R[i..j])$.

Approximate matching of RNA sequence-structure patterns

To find in a long RNA sequence S approximate matches of an RSSP $\mathcal{Q}$ describing a part of an RNA molecule, we compute alignments of the complete $\mathcal{Q}$ and substrings of S considering edit operations for unpaired bases and base pairs. That is, we compute semi-global alignments simultaneously obtaining the sequence-structure edit distance of $\mathcal{Q}$ and substrings of S.

We define the alignment of $\mathcal{Q}$ and a substring $S[p..q]$, $1 \leq p \leq q \leq n$, as set $A = A_{\text{match}} \uplus A_{\text{gap}}$. The set $A_{\text{match}} \subseteq [1..m] \times [p..q]$ of match edges satisfies that, for all different $(k, l), (k', l') \in A_{\text{match}}$, $k > k'$ implies $l > l'$. The set A_{gap} of gap edges is defined as

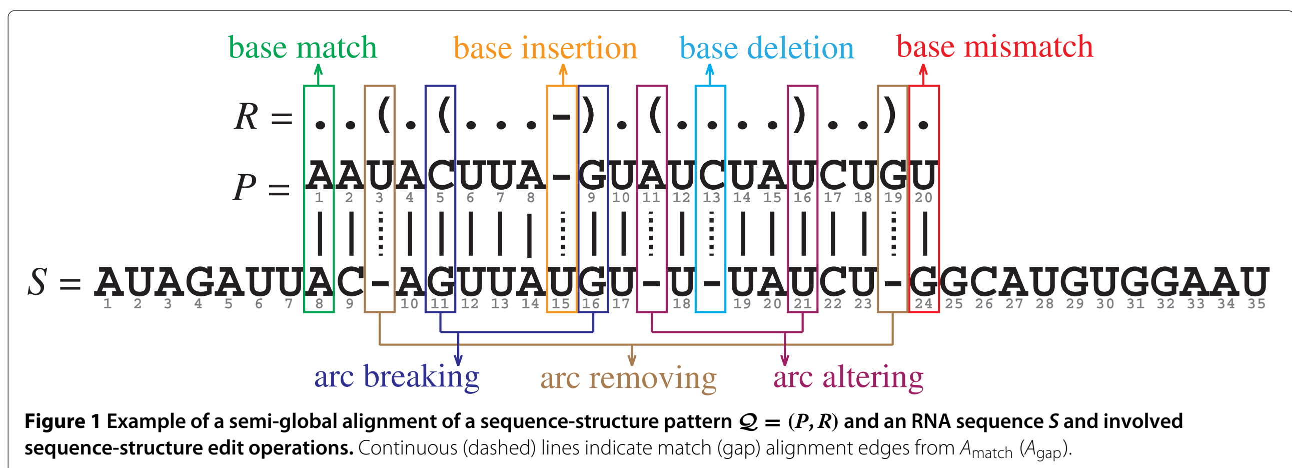

Figure 1 Example of a semi-global alignment of a sequence-structure pattern $\mathcal{Q} = (P, R)$ and an RNA sequence S and involved sequence-structure edit operations. Continuous (dashed) lines indicate match (gap) alignment edges from A_{match} (A_{gap}).

$\{(x,-) \mid x \in [1..m] \wedge \nexists y, (x,y) \in A_{\text{match}}\} \cup \{(-,y) \mid y \in [p..q] \wedge \nexists x, (x,y) \in A_{\text{match}}\}$. See Figure 1 for an example of a semi-global alignment and associated alignment edges. The alignment cost is based on a sequence-structure edit distance. The allowed edit operations on unpaired bases $P[k]$ and $S[l]$, $1 \le k \le m$, $p \le l \le q$, are *base mismatch* (*match*), with cost ω_{m} (zero), which occurs if there is an edge $(k,l) \in A_{\text{match}}$ and $S[l] \notin \varphi(P[k])$ ($S[l] \in \varphi(P[k])$), and *base deletion* (*insertion*), with cost ω_{d}, which occurs if $(k,-) \in A_{\text{gap}}$ ($(-,l) \in A_{\text{gap}}$). The possible edit operations on base pairs were first introduced by Jiang *et al.* [25] and are defined as follows. Let (k_1, k_2) be a base pair in $\widehat{R}$ and l_1 and l_2, $p \le l_1 < l_2 \le q$, be positions in S.

- An *arc breaking*, with cost ω_{b}, occurs if $(k_1, l_1) \in A_{\text{match}}$ and $(k_2, l_2) \in A_{\text{match}}$ but bases $S[l_1]$ and $S[l_2]$ are not complementary. An additional base mismatch cost ω_{m} is caused if $S[l_1] \notin \varphi(P[k_1])$ and another if $S[l_2] \notin \varphi(P[k_2])$. To give an example, consider the semi-global alignment in Figure 1. RSSP $\mathcal{Q}$ contains base pair $(5,9) \in \widehat{R}$ and there exist edges $(5,11) \in A_{\text{match}}$ and $(9,16) \in A_{\text{match}}$ but $S[11] = G$ and $S[16] = G$ are not complementary. We note a difference between our definition and the definition of Jiang *et al.*, where both aligned sequences are annotated with structure information. There, an arc breaking occurs if bases $S[l_1]$ and $S[l_2]$ are annotated as unpaired in addition to the condition of existing edges $(k_1, l_1) \in A_{\text{match}}$ and $(k_2, l_2) \in A_{\text{match}}$. Hence, because in our case sequence S has no structure annotation, our definition is based on the complementarity of bases $S[l_1]$ and $S[l_2]$.
- An *arc altering*, with cost ω_{a}, occurs if either (1) $(k_1, l_1) \in A_{\text{match}}$ and $(k_2, -) \in A_{\text{gap}}$ or (2) $(k_2, l_2) \in A_{\text{match}}$ and $(k_1, -) \in A_{\text{gap}}$. Each case induces an additional base mismatch cost ω_{m} if $S[l_1] \notin \varphi(P[k_1])$ or $S[l_2] \notin \varphi(P[k_2])$. As an example, observe in the alignment shown in Figure 1 that there exist a base pair $(11,16) \in \widehat{R}$ and edges $(11,-) \in A_{\text{gap}}$ and $(16,21) \in A_{\text{match}}$.
- An *arc removing*, with cost ω_{r}, occurs if $(k_1, -) \in A_{\text{gap}}$ and $(k_2, -) \in A_{\text{gap}}$. As an example, observe in the alignment in Figure 1 that there exist a base pair $(3,19) \in \widehat{R}$ and edges $(3,-) \in A_{\text{gap}}$ and $(19,-) \in A_{\text{gap}}$.

With this set of edit operations on the sequence and structure we can now define the cost of the alignment of $\mathcal{Q}$ and $S[p..q]$ as

$$dist(\mathcal{Q}, S[p..q]) = \min\{dist_A(\mathcal{Q}, S[p..q]) \mid A \text{ is an alignment of } \mathcal{Q} \text{ and } S[p..q]\} \quad (1)$$

where

$$\begin{aligned} dist_A(\mathcal{Q}, S[p..q]) = & \sum_{(k,l)\in A, R[k]=., S[l]\notin\varphi(P[k])} \omega_{\text{m}} && \text{base mismatch} \\ + & \sum_{(k,-)\in A, R[k]=.} \omega_{\text{d}} && \text{base deletion} \\ + & \sum_{(-,l)\in A} \omega_{\text{d}} && \text{base insertion} \\ + & \sum_{(k_1,k_2)\in\widehat{R}, (k_1,l_1)\in A, (k_2,l_2)\in A, (S[l_1],S[l_2])\notin\mathcal{C}} \omega_{\text{b}} && \text{arc breaking} \\ + & \sum_{(k_1,k_2)\in\widehat{R}, (k_1,l_1)\in A, (k_2,-)\in A} \omega_{\text{a}} && \text{arc altering} \\ + & \sum_{(k_1,k_2)\in\widehat{R}, (k_2,l_2)\in A, (k_1,-)\in A} \omega_{\text{a}} && \text{arc altering} \\ + & \sum_{(k_1,k_2)\in\widehat{R}, (k_1,-)\in A, (k_2,-)\in A} \omega_{\text{r}} && \text{arc removing.} \end{aligned} \quad (2)$$

An alignment A of minimum cost between $\mathcal{Q}$ and $S[p..q]$ is an *optimal alignment* of $\mathcal{Q}$ and $S[p..q]$.

In practice, one is often interested in finding substrings of an RNA sequence S having a certain degree of similarity to a given RSSP $\mathcal{Q}$ on both the sequence and structure levels. Therefore, we are only concerned about optimal alignments of $\mathcal{Q}$ and substrings $S[p..q]$ with up to a user-defined sequence-structure edit distance and a limited number of allowed insertions and deletions (indels). More precisely:

- the cost $dist(\mathcal{Q}, S[p..q])$ should not exceed a given threshold $\mathcal{K}$, and
- the number of indels in the alignment should be at most d.

Thus, the approximate search problem for finding occurrences of an RSSP $\mathcal{Q}$ in S, given user-defined thresholds $\mathcal{K}$ and d, is to report all intervals $[p..q]$ such that

$$dist(\mathcal{Q}, S[p..q]) \le \mathcal{K} \text{ and } m-d \le |S[p..q]| \le m+d \le n. \quad (3)$$

We call every substring $S[p..q]$ satisfying Equation (3) a *match* of $\mathcal{Q}$ in S. In the subsequent sections we present algorithms for searching for matches of an RSSP $\mathcal{Q}$ in a sequence S.

Online approximate RNA database search for RSSPs: *ScanAlign*

A straightforward algorithm to search for approximate matches of an RSSP $\mathcal{Q}$ in an RNA sequence S consists of sliding a window of length $m' = m + d$ along S while computing $dist(\mathcal{Q}, S[p..q])$ for $1 \le p \le q \le n$ and $q - p + 1 = m'$. We note that, although the length of a match can vary in the range $m-d$ to $m+d$, to find matches of all possible lengths it suffices to slide a window of length

m' along S corresponding to substrings $S[p..q]$. This holds because the alignment to a window of length m' entails all possible alignments with up to d allowed indels. In the following we present a dynamic programming algorithm computing $dist(\mathcal{Q}, S[p..q])$ for every window $S[p..q]$. Our recurrences are derived from the algorithm for global pairwise sequence-structure alignment of Jiang *et al.* [25], i.e. an algorithm for aligning sequences of similar lengths. Although Jiang's algorithm supports the sequence-structure edit operations described above, we emphasize that it is not suitable for computing semi-global alignments, which is what we are interested in.

We begin the description of our algorithm by defining three functions required by the dynamic programming recurrences. Let $T = S[p..q]$.

1. For computing base match and mismatch costs for positions i and j of the RSSP $\mathcal{Q} = (P, R)$ and substring T, respectively, we define a function $\chi : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ as:

$$\chi(i,j) = \begin{cases} 0 \ \text{if } T[j] \in \varphi(P[i]) & \text{(base match)} \\ 1 \ \text{otherwise.} & \text{(base mismatch)} \end{cases} \quad (4)$$

2. To determine whether an arc breaking operation can occur, we must also be able to check for base complementarity at positions i and j of T. Therefore, we define a function $comp : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ as:

$$comp(i,j) = \begin{cases} 0 \ \text{if } (T[i], T[j]) \in \mathcal{C} & \text{(complementary)} \\ 1 \ \text{otherwise.} & \text{(not complementary)} \end{cases} \quad (5)$$

3. For determining the correct row (of the dynamic programming matrices introduced below) where certain operation costs must be stored we introduce a function $row : \mathbb{N} \to \mathbb{N}$ defined as:

$$row(i) = \begin{cases} i' \ \text{if } (i', i) \in \widehat{R} \text{ and } 1 < i' < i < m \text{ and } R[i+1] \\ \quad = . \text{ and } R[i'-1] \neq (\\ 0 \ \text{if } (i, i') \in \widehat{R} \text{ and } R[i+1] = . \\ i \ \text{otherwise.} \end{cases} \quad (6)$$

Intuitively, function *row* satisfies the following: (1) given the right index i of a base pair (i', i), it returns the left index i' if (i', i) is preceded or followed by other structures; (2) given the left index i of a base pair (i, i'), it returns 0 if the base at position $i+1$ of $\mathcal{Q}$ is unpaired; and (3) given any other position index i, it returns i itself.

Using these three functions, our algorithm determines the sequence-structure edit distance $dist(\mathcal{Q}, T[1..m'])$ by computing a series of $m'+1$ $(m'+1) \times (m'-k+1)$ matrices DP_k, for $1 \leq k \leq m'+1$, such that $DP_1(row(m), m') = dist(\mathcal{Q}, T[1..m'])$. We remark that $DP_k(i,j)$ is not defined for every subinterval $[i..j]$. While the recurrences of Jiang's algorithm are divided in four main cases, we present a simplified recurrence relation with only two main cases. In addition, we observe that we use only three indices for a matrix entry instead of four. Our recurrences are as follows.

1. If $i = 0$ or $R[i] = .$ (unpaired base), then

$$DP_k(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ DP_k(0, j-1) + \omega_{\mathrm{d}} & \text{if } i = 0 \text{ and } j > 0 \\ DP_k(row(i-1), 0) + \omega_{\mathrm{d}} & \text{if } i > 0 \text{ and } j = 0 \\ \min \begin{cases} DP_k(row(i-1), j) + \omega_{\mathrm{d}} \\ DP_k(i, j-1) + \omega_{\mathrm{d}} \\ DP_k(row(i-1), j-1) + \chi(i,j)\omega_{\mathrm{m}} \end{cases} & \text{if } i > 0 \text{ and } j > 0 \end{cases} \quad (7)$$

2. If $R[i] \neq .$ (paired base), then
   (a) If $R[i] =)$ where i forms base pair $(i', i) \in \widehat{R}$,

$$DP_k(i,j) = \begin{cases} DP_k(row(i-1), 0) + \omega_{\mathrm{r}} & \text{if } j = 0 \\ \min \begin{cases} DP_k(row(i-1), j-1) + \chi(i, j+k)\omega_{\mathrm{m}} + \omega_{\mathrm{a}} \\ DP_{k+1}(row(i-1), j-1) + \chi(i', k)\omega_{\mathrm{m}} + \omega_{\mathrm{a}} \\ DP_k(row(i-1), j) + \omega_{\mathrm{r}} \\ DP_k(i, j-1) + \omega_{\mathrm{d}} \\ DP_{k+1}(i, j-1) + \omega_{\mathrm{d}} \\ DP_{k+1}(row(i-1), j-2) + (\chi(i, j+k) \\ \quad + \chi(i', k+1))\omega_{\mathrm{m}} + \\ comp(k+1, j+k)\omega_{\mathrm{b}}, \ \text{if } j > 1 \end{cases} & \text{if } j > 0 \end{cases} \quad (8)$$

   (b) If (a) holds and either $R[i'-1] = .$ or $R[i'-1] =)$, compute in addition to Equation (8)

$$DP_k(row(i), j) = \begin{cases} DP_k(row(i'-1), 0) + DP_k(i, 0) & \text{if } j = 0 \\ \min\{DP_k(row(i'-1), j') + DP_{k+j'}(i, j-j') \mid 0 \leq j' \leq j\} & \text{if } j > 0 \end{cases} \quad (9)$$

A natural way to compute these DP matrices is top down, checking whether case 1, 2(a), or 2(b) applies, in this order. Due to the matrix dependencies in cases 2(a) and (b), the matrices need to be computed simultaneously.

Note that for all j, $1 \leq j \leq m'$, clearly $DP_1(row(m), j) = dist(\mathcal{Q}, T[1..j])$. Therefore all candidate matches shorter than m' beginning at position p are also computed in the computation of $dist(\mathcal{Q}, T[1..m'])$. The following Lemma is another important contribution of this work and also the key for the development of an efficient algorithm.

Lemma 1. *When sliding a window along S to compute* $dist(\mathcal{Q}, S[p..q])$, $1 \leq p \leq q \leq n$, $m' = q - p + 1 = m + d$, *a window shift by one position to the right requires to compute only column* $m' - k + 1$, *i.e. the last column of matrices* DP_k, $1 \leq k \leq m'$.

Proof. Let $T[1..m'] = S[p..q]$. The computation of $dist(\mathcal{Q}, T[1..m'])$ requires to compute $m'+1$ DP matrices, one for each suffix T_k of string $T = T[1..m']$, $1 \leq k \leq m'$, and one for the empty sequence ε. As a result, it holds for every k that $dist(\mathcal{Q}, T_k) = DP_k(row(m), m')$ which is obtained as a by-product of the $dist(\mathcal{Q}, T)$ computation. Because each substring $T_{l+1}[1..m'-l] = S[p+l..q]$,

$0 \le l < m'$, only differs by its last character from $S[p+l+1..q+1]$ which are suffixes of the window substring shifted by one position to the right, the lemma holds. □

Due to Lemma 1, our algorithm computes only the last column of the *DP* matrices for every shifted window substring (see the example in Figure 2) and just for the first window $S[1..m']$ it computes every column. We call this algorithm *ScanAlign*. We note that during the reviewing process of this manuscript, Will *et al.* [27] submitted and published an algorithm for semi-global sequence-structure alignment of RNAs. As our method, this algorithm saves computation time by reusing entries of dynamic programming tables while scanning the target sequence.

Our *ScanAlign* algorithm has the following time complexity: computing $DP_k(i,j)$ in cases 1 and 2(a) takes $O(1)$ time and in case 2(b) it takes $O(m')$ time. Now consider the two situations:

- For the first computed window substring $S[1..m']$, cases 1 and 2(a) require $O(mm'^2)$ time in total and case 2(b) requires $O(mm'^3)$ time in total. This leads to an overall time of $O(mm'^3)$.
- For one window shift, cases 1 and 2(a) require $O(mm')$ time in total and case 2(b) requires $O(mm'^2)$ time in total, leading to an overall time of $O(mm'^2)$.

Since there are $n - m' - 1$ window shifts, the computation for all shifted windows takes $O(mm'^2(n - m')) = O(mm'^2 n)$ time. We observe that the time needed by *ScanAlign* to compute all window shifts reduces to $O(mm'n)$ if recurrence case 2(b) is not required. This is the case if the structure of $\mathcal{Q}$ does not contain unpaired bases before a base pair constituting e.g. a left dangling end or left bulge.

Faster online alignment with early-stop computation: LScanAlign

Often, before completing the computation of the alignment between an RSSP $\mathcal{Q}$ and a window substring $S[p..q]$ of the searched RNA sequence, we can determine whether the cost of this alignment will exceed the cost threshold $\mathcal{K}$. By identifying this situation as early as possible, we can improve algorithm *ScanAlign* to skip the window, thus saving computation time and proceed with aligning the next window. The idea consists in checking, during the alignment computation, whether the cost of an already aligned region of $\mathcal{Q}$ and a substring of $S[p..q]$ exceeds $\mathcal{K}$. In such a case, the alignment cost of the complete $\mathcal{Q}$ and $S[p..q]$ will also exceed $\mathcal{K}$. In more detail, this works as follows.

- We decompose the RSSP $\mathcal{Q}$ into regions that can themselves represent a pattern, e.g. a stem-loop or unpaired region. A basic constraint is to not split base pairs to different regions.
- We compute the alignment of a given initial RSSP region and a substring of the current window $S[p..q]$, progressively extending the alignment to other regions.
- If the cost of aligning an RSSP region to a substring of the window exceeds cost threshold $\mathcal{K}$, then the entire pattern cannot match the window. This means that the window can immediately be skipped.

Formally, a valid RSSP region $\mathcal{Q}[x..y]$, $1 \le x \le y \le m$, satisfies exactly one of the following conditions.

k=1

R	P	$DP_k(i,j)$	ε	A	C	C	C	U	C	U	U
			0	1	2	3	4	5	6	7	8
	ε	0	0	1	2	3	4	5	6	7	8
.	A	1	1	0	1	2	3	4	5	6	7
.	A	2	2	1	1	2	3	4	5	6	7
(	G	3	8	7	6	6	6	6	5	6	6
.	U	4	1	1	2	3	4	4	5	6	7
.	U	5	2	2	2	3	4	4	5	5	6
.	U	6	3	3	3	3	4	4	5	5	5
)	C	7	6	6	5	5	5	5	6	6	6

k=2

ε	C	C	C	U	C	U	U
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	2	3	4	5	6	7
8	7	7	7	7	6	6	6
1	1	2	3	3	4	5	6
2	2	2	3	3	4	4	5
3	3	3	3	3	4	4	4
6	5	5	5	5	5	6	6

k=3

ε	C	C	U	C	U	U
0	1	2	3	4	5	6
0	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	2	3	4	5	6
8	7	7	7	6	6	6
1	1	2	2	3	4	5
2	2	2	2	3	3	4
3	3	3	2	3	3	3
6	5	5	5	4	5	5

k=4

ε	C	U	C	U	U
0	1	2	3	4	5
0	1	2	3	4	5
1	1	2	3	4	5
2	2	2	3	4	5
8	7	7	6	6	6
1	1	1	2	3	4
2	2	1	2	2	3
3	3	2	2	2	2
6	5	5	4	4	4

k=5

ε	U	C	U	U
0	1	2	3	4
0	1	2	3	4
1	1	2	3	4
2	2	2	3	4
8	7	6	6	6
1	0	1	2	3
2	1	1	1	2
3	2	2	1	1
6	5	4	4	4

k=6

ε	C	U	U
0	1	2	3
0	1	2	3
1	1	2	3
2	2	2	3
8	7	7	6
1	1	1	2
2	2	1	1
3	3	2	1
6	5	5	4

k=7

ε	U	U
0	1	2
0	1	2
1	1	2
2	2	2
8	7	6
1	0	1
2	1	0
3	2	1
6	5	4

k=8

ε	U
0	1
0	1
1	1
2	2
8	7
1	0
2	1
3	2
6	5

k=9

ε
0
0
1
2
8
1
2
3
6

Figure 2 ***DP*** **tables for the sequence-structure alignment computation of RSSP $\mathcal{Q} = (\text{AAGUUUC}, ..(...))$ and window substring $T = \text{ACCCUCUU}$ when scanning a sequence S with algorithm** ***ScanAlign*****.** Only the entries in red have to be computed for each window shift, whereas the entries in green are reused. Entries in yellow boxes are on a possible minimizing path for alignments with up to $d = 1$ indels. The following operation costs were used: $\omega_d = \omega_m = 1$, $\omega_b = \omega_a = 2$, and $\omega_r = 3$.

1. $\mathcal{Q}[x..y]$ is a left dangling (unpaired) end of the pattern in $5'$ to $3'$ direction, i.e. $x = 1$. Alternatively, it is an unpaired region of maximal length such that position $x-1$ forms a base pair $(x-1, y') \in \widehat{R}$ for some position y' of $\mathcal{Q}$. Observe that no extension of $\mathcal{Q}[x..y]$ by another unpaired position is possible. As an example, consider the green marked regions $\mathcal{Q}[1..2]$, $\mathcal{Q}[4..4]$, $\mathcal{Q}[6..8]$, and $\mathcal{Q}[12..15]$ in Figure 3.
2. Position y is unpaired and there is at least one base pair $(x', y') \in \widehat{R}$, $x \leq x' < y' < y$. No extension of $\mathcal{Q}[x..y]$ by another unpaired position is possible. As examples of regions under these requirements, see the regions in orange of the RSSP $\mathcal{Q}$ in Figure 3, namely $\mathcal{Q}[4..10]$, $\mathcal{Q}[4..18]$, and $\mathcal{Q}[1..20]$.
3. $(x, y) \in \widehat{R}$ is a base pair. For examples of such RSSP regions, see the regions in blue of the RSSP in Figure 3, namely $\mathcal{Q}[5..9]$, $\mathcal{Q}[11..16]$, and $\mathcal{Q}[3..19]$.
4. y forms a base pair $(x', y) \in \widehat{R}$ where either $R[x'-1] = .$ or $R[x'-1] =)$, $1 \leq x \leq x'-1$. In addition, $x = 1$ or $(x-1, y') \in \widehat{R}$ for some $y' > y$. Examples of such RSSP regions are shown in red in Figure 3, i.e. regions $\mathcal{Q}[4..9]$, $\mathcal{Q}[4..16]$, and $\mathcal{Q}[1..19]$.

Note that regions can be embedded in other regions but cannot partially overlap another.

Our progressive alignment computation of an RSSP $\mathcal{Q}$ and a window substring of the searched RNA sequence S begins by considering only an in general small region of $\mathcal{Q}$ embedded in another region. The computation is then extended to a surrounding region, e.g. from region $\mathcal{Q}[6..8]$ to $\mathcal{Q}[5..9]$ of the RSSP shown in Figure 3, until it entails the largest region surrounding all other regions, e.g. $\mathcal{Q}[1..20]$ of the same example. Formally, we elaborate the alignment computation as follows. Let $T = T[1..m']$ be a window substring of length $m' = m + d$ of S and d be the number of allowed indels. Pattern regions have the property that, for any region $\mathcal{Q}[x..y]$, computing $dist(\mathcal{Q}[x..y], T)$ does not depend on any other region $\mathcal{Q}[x'..y']$ for some $y' < x$ and $x' < y$. Therefore, they can easily be sorted to indicate the order by which the rows of the DP matrices are computed. We observe that the top-down computation of the DP matrices, as described above, automatically sorts the regions and respects the dependency between rows. To obtain from the sorted regions the indices of the rows to be computed, we consider the condition satisfied by each region. The rows obtained according to each condition are computed according to one case of the recurrence. Given region $\mathcal{Q}[x..y]$ identified by one of the four conditions this region satisfies, the following rows of the matrices have to be computed.

1. All rows in the interval $[x..y]$ are computed by Equation (7).
2. One scans the structure of region $\mathcal{Q}[x..y]$ from position y to position x until one finds a paired position y'. Then, all rows in the interval $[y'+1..y]$ are computed by Equation (7).
3. Row y is computed by recurrence (a) of Equation (8).
4. Row $row(y)$ is computed by recurrence (b) of Equation (8).

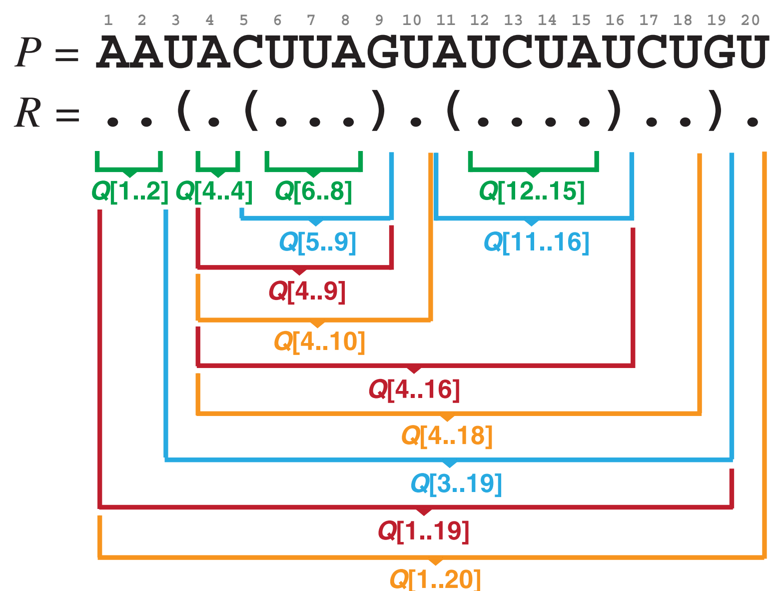

Figure 3 Regions of RSSP $\mathcal{Q} =$ (AAUACUUAGUAUCUAUCUGU, . . (. (. . .) . (. . . .) . .) .) according to conditions 1 (green), 2 (orange), 3 (blue), and 4 (red) described in the text.

The sequential computation of the rows belonging to each region naturally leads to the computation of the entire alignment of $\mathcal{Q}$ and sequence-structure edit distance $dist(\mathcal{Q}, T)$.

Our improvement of the *ScanAlign* algorithm is based on the following two observations.

- The standard dynamic programming algorithm for aligning two plain text sequences of lengths m and n requires an $(m+1) \times (n+1)$ matrix. Let i and j be indices of each of the matrix dimensions and a diagonal v be those entries defined by i and j such that $j - i = v$. Given that the cost of each edit operation is a positive value, the cost of the entries along a diagonal of the matrix are always non-decreasing [28].
- Moreover, one indel operation implies that an optimal alignment path including an entry on diagonal v also includes at least one entry on diagonal $v+1$ or $v-1$. Now let v be the diagonal ending at the entry on the lower-right corner of the matrix and d be the number of allowed indels. One can stop the alignment computation as soon as all the entries of one row in the matrix and along diagonals $v + d'$, $-d \leq d' \leq d$, exceed $\mathcal{K}$.

For our improvement of algorithm *ScanAlign*, based on the following Lemma, we define a diagonal for each RSSP region instead of only one for the entire matrices.

Lemma 2. *Assume an RSSP* $\mathcal{Q} = (P, R)$*, a region* $\mathcal{Q}[x..y]$ *of length* $l = y - x + 1$*, a window substring* $T[1..m']$ *of the searched RNA sequence, a cost threshold* $\mathcal{K}$*, and number* d *of allowed indels. If for every* d', $-d \leq d' \leq min\{d, x\}$, $z \in \{|d'| - d, -|d'| + d\}$, $y + d' \leq m'$*, it holds that* $dist(\mathcal{Q}[x..y], T_{x+d'}[1..l+z]) > k$*, then, for every* d'', $0 \leq d'' \leq d$, $dist(\mathcal{Q}, T[1..m' - d'']) > k$.

Proof. If the RSSP region $\mathcal{Q}[x..y]$ originates from condition 1 or 2 (3 or 4) above, we define the entries on a diagonal e as those entries $DP_k(i,j)$ $(DP_k(row(y), j))$, $1 \leq k \pm d \leq m'$, such that $j - i + offset = e$, where $offset = x - 1$. Without loss of generality let $d = 1$. Assuming $x - 1 > 0$ and $y + 1 \leq m'$, this means that an optimal alignment of pattern $\mathcal{Q}$ and substring T requires $\mathcal{Q}[x..y]$ to align with:

- $T[x..y]$, $T[x..y-1]$, or $T[x..y+1]$, requiring for all three alignments the computation of $dist(\mathcal{Q}[x..y], T_x[1..l+z])$ for $z \in \{0-1, 0+1\} = \{-1, 1\}$;
- $T[x-1..y-1]$, requiring the computation of $dist(\mathcal{Q}[x..y], T_{x-1}[1..l+z])$ for $z \in \{|-1|-1, -|-1|+1\} = \{0\}$; or
- $T[x+1..y+1]$, requiring the computation of $dist(\mathcal{Q}[x..y], T_{x+1}[1..l+z])$ for $z \in \{|1|-1, -|1|+1\} = \{0\}$.

The alignments with $T[x..y]$, $T[x..y+1]$, and $T[x..y-1]$ end in matrix DP_x. The alignments with $T[x-1..y-1]$ end in matrix DP_{x-1}, and the alignments with $T[x+1..y+1]$ end in matrix DP_{x+1}. Every minimizing path obtained for the entire alignment of $\mathcal{Q}$ and T can only include the entries on the diagonals e, $e+1$, and/or $e-1$ for the alignments with $T[x..y]$, $T[x..y+1]$, and $T[x..y-1]$, and can only include the entries on diagonal e for the alignments with $T[x-1..y-1]$ and $T[x+1..y+1]$ because these substrings already imply alignments with one indel. As the sum of the cost of the edit operations on the minimizing path increases monotonically and there cannot be other minimizing paths due to the limited number of indels d, the lemma holds. □

Let $\mathcal{Q}$ be an RSSP whose regions are sorted by the order of computation of their respective rows in the *DP* tables above, let d be the number of allowed indels, and $T = T[1..m']$ be a window substring of the searched RNA sequence. Applying Lemma 2, we modify algorithm *ScanAlign* to compute the alignment of each region $\mathcal{Q}[x..y]$ to substrings $T_{x+d'}$, $-d \leq d' \leq \min\{d, x\}$, $y + d' \leq m'$, and progressively extend the alignment to other RSSP regions and substrings of T as long as $dist(\mathcal{Q}[x..y], T_{x+d'}[1..l+z]) \leq k, z \in \{|d'| - d, -|d'| + d\}$, holds. That is, for each RSSP region, it determines the rows and recurrence case required for their computation according to conditions 1, 2, 3, or 4 above. Then, within each processed row i, it checks whether for at least one entry $DP_k(i,j)$ on a possible minimizing path, i.e. on diagonals e', $e - d \leq e' \leq e + d$, $DP_k(i,j) \leq k$. If no entry is below $\mathcal{K}$, it skips the alignment computation for all remaining RSSP regions and proceeds with aligning the next window. See Figure 2 for an example of the *DP* matrices of an alignment computation whose entries on a possible minimizing path are highlighted in yellow.

When scanning the searched RNA sequence, a window can be shifted before all *DP* matrices entries are computed. Hence, a direct application of Lemma 1 is no longer possible. To overcome this, we define an array Z in the range 1 to z, where z is the number of RSSP regions, and associate each region with an index r, $1 \leq r \leq z$. Let p be the starting position of the window substring $S[p..q]$ in the RNA sequence. We set $Z[r] = p$ whenever all *DP* matrices rows and columns belonging to region r are computed. This occurs when the cost of aligning this region does not exceed cost threshold $\mathcal{K}$. Now, when aligning the same RSSP region r to a different window substring $S[p'..q']$, $p' > p$, computing all *DP* matrices columns requires to compute the last $p' - p$ columns. If $p' - p < m'$ (recall that $m' = q - p = q' - p'$), this means that the two window substrings do not overlap and therefore no *DP* matrix column can be reused.

Our improved algorithm, hereinafter called *LScanAlign*, in the worst case needs to process every RSSP region for every window shift. Hence, it has the same time complexity as algorithm *ScanAlign*. However, as in many cases only a few RSSP regions are evaluated, it is much faster in practice as will be shown later. *ScanAlign* and *LScanAlign* are the basis for further improvements presented in the subsequent sections.

Index-based search: LESAAlign

Suffix trees and enhanced suffix arrays are powerful data structures for exact string matching and for solving other string processing problems [29,30]. In the following we show how the use of enhanced suffix arrays leads to even faster algorithms for searching for matches of an RSSP $\mathcal{Q}$ in an RNA sequence S.

The enhanced suffix array of a sequence S is composed of the suffix array suf and the longest common prefix array lcp. Let \$, called terminator symbol, be a symbol not in $\mathcal{A}$ for marking the end of a sequence. \$ is larger than all the elements in $\mathcal{A}$. suf is an array of integers in the range 1 to $n+1$ specifying the lexicographic order of the $n+1$ suffixes of the string $S\$$. That is, $S_{\mathsf{suf}[1]}, S_{\mathsf{suf}[2]}, ..., S_{\mathsf{suf}[n+1]}$ is the sequence of suffixes of S in ascending lexicographic order. Table suf requires $4n$ bytes and can be constructed in $O(n)$ time and space [31]. In practice non-linear time construction algorithms [32,33] are often used as they are faster. lcp is a table in the range 1 to $n+1$ such that $\mathsf{lcp}[1] = 0$, and $\mathsf{lcp}[i]$ is the length of the longest common prefix between $S_{\mathsf{suf}[i-1]}$ and $S_{\mathsf{suf}[i]}$ for $1 < i \leq n+1$. Table lcp requires n bytes and stores entries with value up to 255, whereas occasional larger entries are stored in an exception table using 8 bytes per entry [30]. More space efficient representations of the lcp table are possible (see [34]). The construction of table lcp can be accomplished in $O(n)$ time and space given suf [35]. For an example of an enhanced suffix array, see Figure 4. In the following we assume that the enhanced suffix array of S has already been computed.

Consider an RSSP $\mathcal{Q}$ to be matched against an RNA sequence S with up to d indels. For each i, $1 \leq i \leq n$, let $p_i = \min\{m+d, |S_{\mathsf{suf}[i]}|\}$ be the *reading depth* of suffix $S_{\mathsf{suf}[i]}$. When searching for matches of $\mathcal{Q}$ in S, we observe that algorithms *ScanAlign* and *LScanAlign* scan S computing $dist(\mathcal{Q}, S[p..q])$ for every window substring of length $q-p+1 = m+d$. In the suffix array, each substring $S[p..q]$ is represented by a suffix $S_{\mathsf{suf}[i]}$ up to reading depth p_i, i.e. there is a substring $S_{\mathsf{suf}[i]}[1..p_i]$ such that $S_{\mathsf{suf}[i]}[1..p_i] = S[p..q]$. To match $\mathcal{Q}$ in S using a suffix array, we simulate a depth first traversal of the lcp interval tree [30] of S on the enhanced suffix array of S such that the reading depth of each suffix is limited by p_i. That is, we traverse the suffix array of S top down, computing the sequence-structure edit distance $dist(\mathcal{Q}, S_{\mathsf{suf}[i]}[1..p_i])$ for each suffix $S_{\mathsf{suf}[i]}$. We recall that candidate matches of $\mathcal{Q}$ have length between $m-d$ and $m+d$ and that $p_i \leq m+d$. In case $p_i < m-d$, we can skip $S_{\mathsf{suf}[i]}$. Also, remember that all candidate matches shorter than p_i are obtained as a by-product of the computation of $dist(\mathcal{Q}, S_{\mathsf{suf}[i]}[1..p_i])$. Hence, for every p', $m-d \leq p' \leq p_i$, if $dist(\mathcal{Q}, S_{\mathsf{suf}[i]}[1..p']) \leq \mathcal{K}$ we report $[\mathsf{suf}[i]..\mathsf{suf}[i]+p']$ as a matching interval of $\mathcal{Q}$ in S. That is, $\mathcal{Q}$ matches substring $S[\mathsf{suf}[i]..\mathsf{suf}[i]+p']$ beginning at position $\mathsf{suf}[i]$ of S.

Our algorithm for the suffix array traversal and $dist(\mathcal{Q}, S_{\mathsf{suf}[i]}[1..p_i])$ computation, hereinafter called *LESAAlign*, builds on algorithms *ScanAlign* and *LScanAlign*. *ScanAlign* and *LScanAlign* exploit overlapping substrings of consecutive window substrings to avoid recomputation of *DP* matrices entries. *LESAAlign* exploits the enhanced suffix array in two different ways. First, for a single suffix $S_{\mathsf{suf}[i]}$, $i > 0$, it benefits from the common prefix of length $\mathsf{lcp}[i]$ between two consecutive suffixes $S_{\mathsf{suf}[i]}$ and $S_{\mathsf{suf}[i-1]}$ by avoiding the recomputation of columns j, $1 \leq j \leq \mathsf{lcp}[i]-k+1$, of each matrix DP_k. This means that, for $lcp = \min\{p_i, \mathsf{lcp}[i]\}$, it avoids the

i	suf[i]	lcp[i]	suf^{-1}[i]	$S_{\mathsf{suf}[i]}$
1	14	0	11	ACCACCCUCUU\$
2	10	3	7	ACCCACCACCCUCUU\$
3	3	4	3	ACCCCCCACCCACCACCCUCUU\$
4	17	4	17	ACCCUCUU\$
5	13	0	16	CACCACCCUCUU\$
6	9	4	15	CACCCACCACCCUCUU\$
7	2	5	14	CACCCCCCACCCACCACCCUCUU\$
8	16	5	10	CACCCUCUU\$
9	12	1	6	CCACCACCCUCUU\$
10	8	5	2	CCACCCACCACCCUCUU\$
11	1	6	13	CCACCCCCCACCCACCACCCUCUU\$
12	15	6	9	CCACCCUCUU\$
13	11	2	5	CCCACCACCCUCUU\$
14	7	6	1	CCCACCCACCACCCUCUU\$
15	6	3	12	CCCCACCCACCACCCUCUU\$
16	5	4	8	CCCCCACCCACCACCCUCUU\$
17	4	5	4	CCCCCCACCCACCACCCUCUU\$
18	18	3	18	CCCUCUU\$
19	19	2	19	CCUCUU\$
20	20	1	20	CUCUU\$
21	22	2	22	CUU\$
22	21	0	21	UCUU\$
23	23	1	23	UU\$
24	24	1	24	U\$
25	25	0	25	\$

Figure 4 Enhanced suffix array of sequence $S\$$ = CCACCCCCCACCCACCACCCUCUU\$ consisting of the suffix array suf, longest common prefix array lcp, and inverse suffix array suf^{-1}. For the definition of suf^{-1}, see the section describing algorithm *LGSlinkAlign*.

recomputation of $\sum_{k=1}^{lcp} lcp - k + 1$ columns for $S_{\mathsf{suf}[i]}$. See an example in Figure 5. We observe that if $p_i \leq lcp$, no *DP* entry needs to be recomputed. In this case, two situations arise:

1. If $p_i \leq lcp$ and $dist(\mathcal{Q}, S_{\mathsf{suf}[i-1]}[1..p_{i-1}]) \leq \mathcal{K}$, then clearly $dist(\mathcal{Q}, S_{\mathsf{suf}[i]}[1..p_i]) \leq \mathcal{K}$ and at least one match of $\mathcal{Q}$ starts at position $\mathsf{suf}[i]$ of S; and
2. If $p_i \leq lcp$ and $dist(\mathcal{Q}, S_{\mathsf{suf}[i-1]}[1..p_{i-1}]) > \mathcal{K}$, then $dist(\mathcal{Q}, S_{\mathsf{suf}[i]}[1..p_i]) > \mathcal{K}$.

These situations allow *LESAAlign* to benefit from the enhanced suffix array in a second important way. That is, it skips all suffixes $S_{\mathsf{suf}[i]}$, $S_{\mathsf{suf}[i+1]}$, ..., $S_{\mathsf{suf}[j]}$ sharing a common prefix of at least length *lcp* with $S_{\mathsf{suf}[i-1]}$. To find the index j of the last suffix $S_{\mathsf{suf}[j]}$ to be skipped, it suffices to look for the largest j such that $\min\{\mathsf{lcp}[i], \mathsf{lcp}[i+1], ..., \mathsf{lcp}[j]\} \geq lcp$. If the first situation above holds, there are matches of $\mathcal{Q}$ in S at positions $\mathsf{suf}[i]$, $\mathsf{suf}[i+1]$, ..., $\mathsf{suf}[j]$. We note that suffixes can also be efficiently skipped using so-called skip-tables as described in [36]. However, to save the $4n$ additional bytes required to store such tables we do not use them here. Our algorithm continues the top-down traversal of the suffix array with suffix $S_{\mathsf{suf}[j+1]}$, taking into account that the *DP* tables were last computed for $S_{\mathsf{suf}[i-1]}$. Consequently, the length of the longest common prefix between $S_{\mathsf{suf}[i-1]}$ and $S_{\mathsf{suf}[j+1]}$ to be considered in the processing of $S_{\mathsf{suf}[j+1]}$ is $\min\{\mathsf{lcp}[i], \mathsf{lcp}[i+1], ..., \mathsf{lcp}[j], \mathsf{lcp}[j+1]\}$.

We also incorporate in our index-based algorithm the early-stop alignment computation scheme of algorithm *LScanAlign*. This allows to skip suffixes $S_{\mathsf{suf}[i]}$ as soon as it becomes clear that the sequence-structure edit distance of RSSP $\mathcal{Q}$ and $S_{\mathsf{suf}[i]}$ up to reading depth p_i will exceed the cost threshold $\mathcal{K}$. For this, *LESAAlign* progressively aligns regions of $\mathcal{Q}$ to a substring of the current suffix as in algorithm *LScanAlign*, checking whether the cost of each subalignment remains below the cost threshold $\mathcal{K}$, thus applying Lemma 2. If the cost exceeds $\mathcal{K}$, the alignment computation of the remaining pattern regions is skipped and the algorithm proceeds with processing the next suffix. To avoid recomputing as many entries of the *DP* matrices as possible while traversing the suffix array, *LESAAlign* differs from *LScanAlign* in the way it manages (non-) aligned regions for each suffix. Lemma 1, which algorithm *LScanAlign* applies to support early-stop computation, relies on scanning the searched RNA sequence S and overlapping window substrings. This makes it unsuitable for use with the suffix array. Instead, *LESAAlign* only uses information from the lcp table as follows. Let z be the number of regions of $\mathcal{Q}$ indexed from 1 to z and $T = S_{\mathsf{suf}[i]}[1..p_i]$ be the current substring. When progressively aligning the regions of $\mathcal{Q}$ to a substring of T, we store the index r of the first region whose alignment cost exceeds $\mathcal{K}$, if there is any. That is, for the first region $\mathcal{Q}[x..y]$ whose index r we store, it holds that for every d', $-d \leq d' \leq \min\{d, x\}$, $dist(\mathcal{Q}[x..y], T_{x+d'}[1..l+z]) > k$ with $l = y - x + 1$, $z \in \{|d'| - d, -|d'| + d\}$, and $y + d' \leq m + d$ (see Lemma 2). Then, when aligning $\mathcal{Q}$ to a subsequent substring $S_{\mathsf{suf}[j]}[1..p_j]$, we must distinguish the regions of $\mathcal{Q}$ *previously computed* from regions *not computed*.

- *Previously computed pattern regions* are all regions whose index is strictly smaller than r. The alignment computation of these regions profits from the common prefix between $S_{\mathsf{suf}[i]}[1..p_i]$ and $S_{\mathsf{suf}[j]}[1..p_j]$ by avoiding the recomputation of *DP* matrices columns as described above.
- *Non-computed pattern regions* are all regions whose index is larger than or equal to r. In this case, all *DP* matrices columns of the respective pattern region need to be computed, even if $S_{\mathsf{suf}[i]}[1..p_i]$ and $S_{\mathsf{suf}[j]}[1..p_j]$ share a common prefix.

k=1

			ε	A	C	C	C	U	C	U	U
R	*P*	$DP_k(i,j)$	0	1	2	3	4	5	6	7	8
	ε	0	0	1	2	3	4	5	6	7	8
.	A	1	1	0	1	2	3	4	5	6	7
.	A	2	2	1	1	2	3	4	5	6	7
(	G	3	8	7	6	6	6	6	5	6	6
.	U	4	1	1	2	3	4	4	5	6	7
.	U	5	2	2	2	3	4	4	5	5	6
.	U	6	3	3	3	3	4	4	5	5	5
)	C	7	6	6	5	5	5	5	6	6	6

k=2

ε	C	C	C	U	C	U	U
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	2	3	4	5	6	7
8	7	7	7	7	6	6	6
1	1	2	3	3	4	5	6
2	2	2	3	3	4	4	5
3	3	3	3	3	4	4	4
6	5	5	5	5	5	6	6

k=3

ε	C	C	U	C	U	U
0	1	2	3	4	5	6
0	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	2	3	4	5	6
8	7	7	7	6	6	6
1	1	2	2	3	4	5
2	2	2	2	3	3	4
3	3	3	2	3	3	3
6	5	5	5	4	5	5

k=4

ε	C	U	C	U	U
0	1	2	3	4	5
0	1	2	3	4	5
1	1	2	3	4	5
2	2	2	3	4	5
8	7	7	6	6	6
1	1	1	2	3	4
2	2	1	2	2	3
3	3	2	2	2	2
6	5	5	4	4	4

k=5

ε	U	C	U	U
0	1	2	3	4
0	1	2	3	4
1	1	2	3	4
2	2	2	3	4
8	7	6	6	6
1	0	1	2	3
2	1	1	1	2
3	2	2	1	1
6	5	4	4	4

k=6

ε	C	U	U
0	1	2	3
0	1	2	3
1	1	2	3
2	2	2	3
8	7	7	6
1	1	1	2
2	2	1	1
3	3	2	1
6	5	5	4

k=7

ε	U	U
0	1	2
0	1	2
1	1	2
2	2	2
8	7	6
1	0	1
2	1	0
3	2	1
6	5	4

k=8

ε	U
0	1
0	1
1	1
2	2
8	7
1	0
2	1
3	2
6	5

k=9

ε
0
0
1
2
8
1
2
3
6

Figure 5 ***DP*** **tables for the sequence-structure alignment computation of RSSP $\mathcal{Q}$ = (AAGUUUC, . . (. . .)) and substring $S_{\mathsf{suf}[i]}[1..8]$ = ACCCUCUU.** Given that suffix $S_{\mathsf{suf}[i]}$ shares a common prefix of length $\mathsf{lcp}[i] = 4$ with $S_{\mathsf{suf}[i-1]}$, algorithm *LESAAlign* reuses the entries in green and computes the entries in red. Used operation costs: $\omega_d = \omega_m = 1$, $\omega_b = \omega_a = 2$, and $\omega_r = 3$.

We observe that longer ranges of suffixes not containing matches to $\mathcal{Q}$ can be skipped thanks to the early-stop alignment computation scheme. Note that the left-most character of T needed to assert $dist(\mathcal{Q}[x..y], T_{x+d'}[1..l+z]) > \mathcal{K}$ is $T[x+l+d-1] = T[x+y-x+1+d-1] = T[y+d]$ as $l = y-x+1$. Therefore, no suffix sharing prefix $T[1..y+d]$ can match $\mathcal{Q}$ and thus can be skipped in the top-down traversal of the suffix array of S. Because in most cases $y+d < p_i$, more suffixes are likely to share a prefix of length $y+d$ than of length p_i with $S_{\mathsf{suf}[i]}$. For the pseudocode of algorithm *LESAAlign*, see Section 1 of Additional file 1.

Enhanced index-based search: LGSlinkAlign

Given an RSSP $\mathcal{Q}$ to be searched in an RNA sequence S, algorithm *LESAAlign* is very fast when it can

- avoid recomputation of *DP* matrices columns due to a common prefix between suffixes of S; and
- skip long ranges of suffixes of the suffix array suf whose common prefix up to a required reading depth are known to match or not match $\mathcal{Q}$.

Therefore, *LESAAlign* exploits repetitions of substrings of S, i.e. substrings shared by different suffixes, and information of the lcp table to save computation time. However, the use of information of the lcp table alone does not necessarily lead to large speedups. Consider e.g. the *DP* matrices for the computation of the alignment of $\mathcal{Q}$ = (AAGUUUC, . . (. . .)) and substring $S_{\mathsf{suf}[4]}[1..p_4]$ = ACCCUCUU in Figure 5. The enhanced suffix array of S is shown in Figure 4. The substring $S_{\mathsf{suf}[4]}[1..p_4]$ of length 8 shares a common prefix of length $\mathsf{lcp}[4] = 4$ with the previously processed substring $S_{\mathsf{suf}[3]}[1..p_3]$. Despite this common prefix, still $182/252 \approx 72\%$ of the *DP* matrices entries need to be computed (disregarding initialization rows and columns 0) in case no early-stop is possible, i.e. in case $\mathcal{K} > 4$. This is more than the at most $56/252 \approx 22\%$ of the *DP* matrices entries computed by the online algorithm *LScanAlign* for a window shift.

Our next goal is to develop an algorithm traversing the enhanced suffix array of S that:

1. can skip more suffixes; and
2. improves the use of already computed *DP* matrices entries, reusing computed entries for as many suffixes as possible.

To address the first goal, we motivate our method by recalling the alignment computation example in Figure 2. In this example, one of the regions of $\mathcal{Q}$ = (AAGUUUC, . . (. . .)) is $\mathcal{Q}[3..7]$ = (GUUUC, (. . .)). Assume $\mathcal{K} = d = 1$ and observe that $dist(\mathcal{Q}[3..7], T_{3+d'}[1..5+z]) > 1$ for every d', $-1 \leq d' \leq 1$, $z \in \{|d'|-1, -|d'|+1\}$, i.e. the alignment cost for this pattern region already exceeds the cost threshold of 1 (in accordance with Lemma 2). In other words, $\mathcal{Q}[3..7]$ cannot align to any of the substrings $T[2..6]$ = CCCUC, $T[3..6]$ = CCUC, $T[3..7]$ = CCUCU, $T[3..8]$ = CCUCUU, or $T[4..8]$ = CUCUU with a cost lower than 1. Observe further that the alignment computation of region $\mathcal{Q}[3..7]$ does not depend on any previous computation of any other region. We can therefore conclude that no suffix containing substring $T[2..8]$ = CCCUCUU from position 2 to 8 can match $\mathcal{Q}$, independently of any prefix of length 1. Our goal is to find and eliminate from the search space all such suffixes, in addition to skipping all suffixes sharing prefix $T[1..8]$ as performed by *LESAAlign*. That is, we want to skip suffixes sharing a substring, not limited to a prefix, whose alignment cost to a pattern region exceeds cost threshold $\mathcal{K}$.

Let S be an arbitrary RNA sequence and $T[x..y] = S_{\mathsf{suf}[i]}[x..y]$ contain all substrings whose alignment cost to a region of an RSSP $\mathcal{Q}$ exceeds threshold $\mathcal{K}$. Consider the following two cases for skipping suffixes that cannot match $\mathcal{Q}$ as a consequence of containing substring $T[x..y]$ from position x to y. (1) For any value of x, all suffixes sharing prefix $T[1..y]$ can be skipped as performed by algorithm *LESAAlign*. (2) Now let $x > 1$. To find all suffixes of S sharing substring $T[x..y]$ from position x to y, we first locate all suffixes sharing $T[x..y]$ as a prefix. We begin by locating one such suffix, in particular the suffix of index $\mathsf{suf}[j]$ that contains all but the first $x' = x-1$ characters of $S_{\mathsf{suf}[i]}$, i.e. suffix $S_{\mathsf{suf}[j]} = S_{\mathsf{suf}[i]+x'}$. We determine j using a generalization of a concept originated from suffix trees. It is a property of suffix trees that for any internal node spelling out string T there is also an internal node spelling out T_2 whenever $|T| > 1$ [37]. A pointer from the former to the latter node is called a *suffix link*. In the case of suffix arrays, a suffix link can be computed using the inverse suffix array suf^{-1} of $S\$$. suf^{-1} is a table in the range 1 to $n+1$ such that $\mathsf{suf}^{-1}[\mathsf{suf}[i]] = i$. It requires $4n$ bytes and can be computed via a single scan of suf in $O(n)$ time. Given table suf^{-1}, we can define the suffix link from $T = S_{\mathsf{suf}[i]}$ to $T_2 = S_{\mathsf{suf}[i]+1}$ as $link = \mathsf{suf}^{-1}[\mathsf{suf}[i]+1]$, i.e. it holds that $\mathsf{suf}[link] = \mathsf{suf}[i]+1$. Now, if $x' = 1$, we already find that the index $\mathsf{suf}[j]$ of the suffix containing all but the first character of $S_{\mathsf{suf}[i]}$ is $\mathsf{suf}[j] = \mathsf{suf}[link]$ because $S_{\mathsf{suf}[link]} = S_{\mathsf{suf}[i]+x'}$ holds. However, we also want to be able to determine j for any $x' \geq 1$. The obvious solution is to compute suffix links x' successive times. Each suffix link skips the first character of the previously located suffix. For a more efficient solution, we generalize suffix links to point directly to the suffix without a prefix of any length x' of the initial suffix. For this purpose we define a function $link : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as:

$$link(i, x') = \mathsf{suf}^{-1}[\mathsf{suf}[i]+x'] . \tag{10}$$

Then, by letting $j = link(i, x')$, $S_{\mathsf{suf}[link(i,x')]} = S_{\mathsf{suf}[i]+x'}$ holds for any $x' \geq 1$. All suffixes sharing $T[x..y]$ as a prefix are all suffixes in the range j_{start} to j_{end} where j_{start} is the smallest and j_{end} is the largest index satisfying $\min\{\mathsf{lcp}[j_{\text{start}}+1], ..., \mathsf{lcp}[j], ..., \mathsf{lcp}[j_{\text{end}}]\} \geq y - x + 1$. Finally, we find that all suffixes of S sharing substring $T[x..y]$ from position x to y are all $S_{\mathsf{suf}[j']-x'}$, $j_{\text{start}} \leq j' \leq j_{\text{end}}$, satisfying $\mathsf{suf}[j'] > x'$. To skip these suffixes not containing matches to $\mathcal{Q}$ in the top-down traversal of the suffix array suf, we mark their positions as true (for already"processed") in a bit array vtab of n bits. The suffix array traversal proceeds from position $\mathsf{suf}[i]$, but skips the marked suffixes when their positions are reached.

We remark that the described method for skipping suffixes can profit from a resorting according to the order by which RSSP regions are aligned. In the alignment computation example in Figure 2, determining $dist(\mathcal{Q}[3..4], T_{3+d'}[1..2+z]) > 1$, $-1 \leq d' \leq 1$, $z \in \{|d'| - 1, -|d'| + 1\}$, does not depend on character $T[1]$ and region $\mathcal{Q}[1..1]$. Hence, region $\mathcal{Q}[1..1]$ is unnecessarily aligned first when the regions are sorted by a top-down analysis of the *DP* tables. To decrease the chance that unnecessary computations occur, we sort the *RSSP* regions to begin aligning with the left-most *RSSP* region $\mathcal{Q}[x..y]$ not depending on the alignment of any other region and satisfying $x - d > 1$.

We now address the second goal, namely reusing computed *DP* matrices entries for as many suffixes as possible. Recall that computing the sequence-structure edit distance $dist(\mathcal{Q}, S_{\mathsf{suf}[i]}[1..p_i])$ for each suffix $S_{\mathsf{suf}[i]}$ up to reading depth p_i means computing p_i+1 *DP* matrices, one for each suffix T_k of string $T = S_{\mathsf{suf}[i]}[1..p_i]$, $1 \leq k \leq m'$, and one for the empty sequence ε. Observe that each suffix T_k, $T_k \neq T$, also occurs itself as a prefix of a suffix in table suf, i.e. there exists a suffix $S_{\mathsf{suf}[j]}$ shorter than $S_{\mathsf{suf}[i]}$ by exactly $k - 1$ characters which has prefix T_k. Consequently, T_k is processed again in an alignment to RSSP $\mathcal{Q}$ at a different point in time during the traversal of suf. Let $T' = S_{\mathsf{suf}[j]}[1..p_j]$. Now note that if T' is at a (nearly) contiguous position in suf to T, T' and T are likely to share a common prefix due to their similar lexicographic ranking. This allows algorithm *LESAAlign* to avoid recomputation of *DP* matrices columns by using information from the lcp table. Unfortunately, T' and T can be lexicographically ranked far away from each other in table suf, meaning that the *DP* matrices computed for T' either:

- were already computed once because T' is lexicographically smaller than T, but were discarded to allow the processing of other suffixes until T was traversed; or
- are computed for the first time otherwise, but will not be reused to also allow the processing of other suffixes until T' occurs in table suf as a prefix of a suffix itself.

In both cases, redundant computations occur. To avoid this, we optimize the use of computed *DP* matrices by processing T' directly after processing T for fixed $k = 2$, recalling that $T = S_{\mathsf{suf}[i]}[1..p_i]$ and $T' = S_{\mathsf{suf}[j]}[1..p_j]$. This value of k implies that $S_{\mathsf{suf}[j]}$ does not contain the first character of $S_{\mathsf{suf}[i]}$ and that we can locate $S_{\mathsf{suf}[j]}$ in table suf by computing the suffix link $j = link(i, 1)$. Also, $k = 2$ implies that T' only differs by its last character from T, aside from not beginning with character $T[1]$. Therefore, to determine $dist(\mathcal{Q}, T')$, we only have to compute the last column of the *DP* matrices required to compute $dist(\mathcal{Q}, T)$ as shown by Lemma 1. We note that, because i and j are not necessarily contiguous positions in suf, we mark the processed suffix $S_{\mathsf{suf}[j]}$ in the bit array vtab so that it is only processed once. If no match to RSSP $\mathcal{Q}$ begins at position $\mathsf{suf}[j]$, we also mark and skip every suffix sharing the substring with T' whose alignment to a region of $\mathcal{Q}$ is known to exceed threshold $\mathcal{K}$. Once T' is processed and all possible suffixes are skipped, we recursively repeat this optimization scheme by setting $T = T'$ and processing the next $T' = S_{\mathsf{suf}[j']}[1..p_{j'}]$ where $j' = link(j, 1)$. The recursion stops when $p_{j'} < m - d$, meaning that T' is too short to match $\mathcal{Q}$, or when $\mathsf{suf}[j']$ is already marked as processed in vtab. The suffix array traversal proceeds at position $i + 1$ repeating the entire scheme.

We call our algorithm incorporating the presented improvements *LGSlinkAlign*. For its pseudocode, see Section 1 of Additional file 1. *LGSlinkAlign* inherits all the improvements of the above presented algorithms. In summary, its improvements are as follows.

- *LGSlinkAlign* traverses the enhanced suffix array of the searched sequence S, i.e. the suffix array suf enhanced with tables lcp and suf^{-1}. During this traversal, it benefits from common prefixes shared among suffixes to (1) avoid the computation of *DP* matrix columns and to (2) skip ranges of suffixes known to match or not match RSSP $\mathcal{Q}$ as in algorithm *LESAAlign.*
- The suffix array traversal is predominantly top down, but non-contiguous suffixes are processed to optimize the use of computed *DP* matrices.
- *LGSlinkAlign* stops the alignment computation as early as the alignment cost of a region of RSSP $\mathcal{Q}$ and a substring of the prefix of the current suffix exceeds threshold $\mathcal{K}$, an improvement first introduced in algorithm *LScanAlign.*
- Due to the early-stop computation scheme, suffixes sharing common prefixes shorter than $m + d$ can be

skipped, leading to larger ranges of skipped suffixes. The early-stop computation scheme also helps to identify and skip non-contiguous suffixes sharing a common substring which is not their prefix.

Example: searching for an RSSP with algorithm LGSlinkAlign

We elucidate the ideas of algorithm *LGSlinkAlign* with the following example. Consider the RSSP $\mathcal{Q} = (\text{AAGUUUC}, \texttt{..(...)})$ to be matched in the sequence S whose enhanced suffix array is shown in Figure 4. To keep the example simple, we only allow a small cost threshold and number of indels, i.e. we set $\mathcal{K} = d = 1$. The costs of the edit operations are $\omega_d = \omega_m = \omega_b = \omega_a = 1$ and $\omega_r = 2$. When traversing the enhanced suffix array of S, *LGSlinkAlign* always begins to align $\mathcal{Q}$ to a substring of S with region $\mathcal{Q}[4..6]$, because the alignment computation of this region does not depend on any other region. In addition, the left index of this region satisfies $4 - d > 1$. This means that the alignment computation of region $\mathcal{Q}[1..2]$ is avoided if the cost of aligning region $\mathcal{Q}[4..6]$ exceeds the threshold $\mathcal{K}$. The algorithm starts the traversal of the enhanced suffix array of S aligning $\mathcal{Q}[4..6]$ to substrings of $T = S_{\mathsf{suf}[1]}[1..p_1] = S_{14}[1..8]$ from positions $4 - d = 3$ and $6 + d = 7$. For this, it computes $dist(\mathcal{Q}[4..6], T_{4+d'}[1..3 + z])$ for $-1 \leq d' \leq 1$ and $z \in \{|d'| - 1, -|d'| + 1\}$. Observe that $dist(\mathcal{Q}[4..5], T_{4+d'}[1..2 + z]) > 1$ holds. Hence (1) no suffix with prefix $T[1..6] =$ AACACC can match $\mathcal{Q}$ and thus can be skipped and (2) no suffix containing substring $T[3..6] =$ CACC from position $4 - d = 3$ to $5 + d = 6$ can match $\mathcal{Q}$ and thus can be skipped as well. We notice that there is no other suffix with prefix AACACC because $\mathsf{lcp}[2] < 6$, so we analyze case (2). The algorithm looks for suffixes sharing substring CACC from position 3 to 6. It begins by locating suffixes without the first two characters of T and containing CACC as a prefix. It follows the suffix link $link(1, 2) = \mathsf{suf}^{-1}[\mathsf{suf}[1]+2] = \mathsf{suf}^{-1}[16] = 7$ and looks for the smallest j_{start} and largest j_{end} satisfying $\min\{\mathsf{lcp}[j_{\text{start}}+1], ..., \mathsf{lcp}[8], ..., \mathsf{lcp}[j_{\text{end}}]\} \geq 4 = |\text{CACC}|$. It finds that $j_{\text{start}} = 5$ and $j_{\text{end}} = 8$, since $\min\{\mathsf{lcp}[5 + 1], \mathsf{lcp}[7], \mathsf{lcp}[8]\} = \min\{4, 5, 5\} \geq 4$ holds. The suffixes containing CACC from position 3 to 6 are $S_{\mathsf{suf}[5]-2} = S_{11}$, $S_{\mathsf{suf}[6]-2} = S_7$, and $S_{\mathsf{suf}[8]-2} = S_{14}$. S_{11} and S_7 are marked in the bit array vtab, whereas $S_{14} = S_{\mathsf{suf}[1]}$ was already processed and does not need to be marked. We observe that $S_{\mathsf{suf}[7]-2} = S_{-1}$ is not a valid suffix. To reuse as many computed *DP* matrices entries as possible, the algorithm next processes the suffix $S_{\mathsf{suf}[j]}$ which does not contain the first character of $S_{\mathsf{suf}[1]}$. It determines $j = link(1, 1) = \mathsf{suf}^{-1}[\mathsf{suf}[1]+1] = 11$ and sets $T = S_{\mathsf{suf}[12]}[1..p_{12}] = S_{15}[1..8]$. The alignment to this substring T begins with its substrings from positions 3 to 7 and $\mathcal{Q}[4..6]$. We observe that $dist(\mathcal{Q}[4..5], T_{4+d'}[1..2 + z]) > 1$ holds and consequently T cannot match $\mathcal{Q}$. Because suffix $S_{\mathsf{suf}[12]} = S_{15}$ was traversed via a suffix link, it is marked as processed in vtab. We now again analyze two cases of suffixes that cannot match $\mathcal{Q}$ and therefore can be skipped: (1) suffixes sharing prefix $T[1..6] =$ CCACCC and (2) suffixes containing substring $T[3..6] =$ ACCC from position 3 to 6. Satisfying case (1) are suffixes $S_{\mathsf{suf}[11]} = S_1$ and $S_{\mathsf{suf}[10]} = S_8$ since $\mathsf{lcp}[12] \geq 6$ and $\mathsf{lcp}[11] \geq 6$. These suffixes are marked in vtab. We now check if there are suffixes satisfying case (2). The algorithm begins by locating suffixes containing substring $T[3..6] =$ ACCC as a prefix. For this, it follows the suffix link $link(12, 2) = \mathsf{suf}^{-1}[\mathsf{suf}[12]+2] = 4$ and determines $j_{\text{start}} = 2$ and $j_{\text{end}} = 4$. The property $\min\{\mathsf{lcp}[2 + 1], \mathsf{lcp}[4]\} \geq 4$ is satisfied. The suffixes containing ACCC from position 3 to 6 are $S_{\mathsf{suf}[2]-2} = S_8$, $S_{\mathsf{suf}[3]-2} = S_1$, and $S_{\mathsf{suf}[4]-2} = S_{15}$. Since these were already marked in vtab, none of them needs to be marked. The algorithmic scheme of *LGSlinkAlign* to reuse as many computed *DP* matrices entries as possible continues processing other suffixes which are located by iteratively following the suffix links. It locates suffixes $S_{\mathsf{suf}[8]}$, $S_{\mathsf{suf}[4]}$, $S_{\mathsf{suf}[18]}$, and $S_{\mathsf{suf}[19]}$ because $link(12, 1) = 8$, $link(8, 1) = 4$, $link(4, 1) = 18$, and $link(18, 1) = 19$, respectively. These suffixes are processed analogously as above, one after the other, not resulting in matches to $\mathcal{Q}$. The iteration then leads to suffix $S_{\mathsf{suf}[20]}$, since $link(19, 1) = 20$. However, $|S_{\mathsf{suf}[20]}| < m - d$, meaning that this suffix is too short to contain a match to $\mathcal{Q}$. This causes the iteration to stop. The suffix array traversal proceeds and repeats the entire matching scheme from the suffix that follows the last processed suffix not located via a suffix link, i.e. suffix $S_{\mathsf{suf}[2]}$. After processing and skipping all possible suffixes, we note that *LGSlinkAlign* does not report any matches for the defined cost threshold and allowed number of indels $\mathcal{K} = d = 1$. By setting $\mathcal{K} = 5$, it reports a match at position 16.

RNA secondary structure descriptors based on multiple ordered RSSPs

RNAs with complex branching structures often cannot be adequately described by a single RSSP due to difficulties in balancing sensitivity, specificity, and reasonable running time of the used search algorithm. Although their description by a single short RSSP specifying an unbranched fragment of the molecule might be very sensitive, it is often too unspecific and likely to generate many spurious matches when searching for structural homologs in large sequence databases or complete genomes. In contrast, using a single long RSSP often requires a higher cost threshold $\mathcal{K}$ for being sensitive enough which in turn, together with the increased RSSP length, has a negative influence on the search time. This might lead to disadvantageous running times in larger search scenarios in practice.

We solve this problem by applying the powerful concept of RNA secondary structure descriptors (SSDs for

short) recently introduced in [23]. The underlying concept of SSDs is similar to the idea of PSSM family models [38], which are successfully used for fast and sensitive protein homology search. SSDs use the information of multiple ordered RSSPs derived from the decomposition of an RNA's secondary structure into stem-loop like structural elements. In a first step, approximate matches to the single RSSPs the SSD consists of are obtained using one of the algorithms presented above. From these matches, either local or global high-scoring chains are computed with the efficient chaining algorithms described in [23]. These algorithms take the chain's score, i.e. the weights of the fragments in the chain, into account (see [23] for details). For chaining of approximate RSSP matches, we use the fragment weight $\omega^*_\mathcal{Q} - dist(\mathcal{Q}, T)$ for an RSSP $\mathcal{Q}$ of length m matching substring T, where $\omega^*_\mathcal{Q} = m * \omega_\mathrm{m} + bps * \omega_\mathrm{r}$ and bps denotes the number of base pairs in $\mathcal{Q}$. Here $\omega^*_\mathcal{Q}$ is the maximal possible weighting $\mathcal{Q}$ can gain when being aligned and therefore it reflects the situation of a perfect match between $\mathcal{Q}$ and T. With this definition of a fragment's weight, a positive weight is always guaranteed, thus satisfying a requirement for the chaining algorithm. Once the chaining of matches to the RSSPs is completed, the high-scoring chains are reported in descending order of their chain score. By restricting to high-scoring chains, spurious RSSP matches are effectively eliminated. Moreover, the relatively short RSSPs used in an SSD can be matched efficiently with the presented algorithms leading to short running times that even allow for the large scale application of approximate RSSP search.

Results and discussion

Implementation and computational results

We implemented (1) the fast index-based algorithms *LESAAlign* and *LGSlinkAlign*, (2) the online algorithms *LScanAlign*, *ScanAlign*, both operating on the plain sequence, and (3) the efficient global and local chaining algorithms described in [23]. In our experiments we use *ScanAlign*, which is the scanning version of the method proposed in [25], for reference benchmarking. All algorithms are included in the program *RaligNAtor*. The algorithms for index construction were implemented in the program *sufconstruct*, which makes use of routines from the *libdivsufsort2* library (see http://code.google.com/p/libdivsufsort/) for computing the suf table in $O(n \log n)$ time. For the construction of table lcp we employ our own implementation of the linear time algorithm of [35]. All programs were written in C and compiled with the GNU C compiler (version 4.5.0, optimization option -O3). All measurements are performed on a Quad Core Xeon E5620 CPU running at 2.4 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 are canonical Watson-Crick and wobble, unless stated otherwise. The used sequence-structure operation costs are $\omega_\mathrm{d} = \omega_\mathrm{m} = \omega_\mathrm{b} = \omega_\mathrm{a} = 1$ and $\omega_\mathrm{r} = 2$.

Comparison of running times

In a first benchmark experiment we measure the running times needed by the four algorithms to search with a single RSSP under different cost thresholds $\mathcal{K}$ and number of allowed indels d. We set (1) $\mathcal{K} = d$ varying the values in the interval $[0, 6]$, (2) $\mathcal{K} = 6$ varying d in the interval $[0, 6]$, and (3) $d = 0$ varying $\mathcal{K}$ in the interval $[0, 6]$. The searched dataset contains 2, 756, 313 sequences with a total length of ≈ 786 MB from the full alignments of all Rfam release 10.1 families. The construction of all necessary index tables needed for *LESAAlign* and *LGSlinkAlign* with *sufconstruct* and their storage on disk required 372 seconds. In the following we refer to this dataset as RFAM10.1 for short. In this experiment we use the RSSP tRNA-pat of length $m = 74$ shown in Figure 6 describing the consensus secondary structure of the tRNA family (Acc.: RF00005). The results of this experiment are presented in Figure 7 and Table S4, S5, and S6 of Additional file 1. *LGSlinkAlign* and *LESAAlign* are the fastest algorithms. *LGSlinkAlign* is faster in particular for increasing values of $\mathcal{K}$ and d, being only slower than *LESAAlign* for small values of $\mathcal{K}$ and d and for fixed $d = 0$. The advantage of *LGSlinkAlign* over *LESAAlign* with higher values of $\mathcal{K}$ and d is explained by the increased reading depth in the suffix array implicated by $\mathcal{K}$ and d and the fewer suffixes sharing a common prefix that can be skipped. This holds for both *LGSlinkAlign* and *LESAAlign*, however *LGSlinkAlign* counterbalances this effect by reusing computed *DP* matrices for non-contiguous suffixes of the suffix array. In a comparison to the two online algorithms considering only approximate matching, i.e. $\mathcal{K} \geq 1$, the speedup factor of *LGSlinkAlign* over *ScanAlign* (*LScanAlign*) is in the range from 560 for $\mathcal{K} = 1$ and $d = 0$ to 17 for $\mathcal{K} = d = 6$ (from 15 for $\mathcal{K} = 2$ and $d = 0$ to 3 for $\mathcal{K} = d = 6$). *LESAAlign* achieves a speedup factor over *ScanAlign* (*LScanAlign*) in the range from 1, 323 for $\mathcal{K} = 1$ and $d = 0$ to 9 for $\mathcal{K} = d = 6$ (29 for $\mathcal{K} = 1$ and $d = 0$ to 1.6 for $\mathcal{K} = d = 6$). In a comparison between the online algorithms, *LScanAlign* is faster than *ScanAlign* by up to factor 45 for $\mathcal{K} \geq 1$. In summary, all algorithms except *ScanAlign* profit from low values of $\mathcal{K}$ and d reducing their search times. This is a consequence of the use of the early-stop alignment computation scheme. As shown in Figure 7(2), also the number of allowed indels d influences the search time. For an additional experiment investigating the influence of $\mathcal{K}$ and d on the search time required by the four algorithms, see Section 2 of Additional file 1. A further experiment, described in Section 3 of Additional file 1, compares *RaligNAtor* and

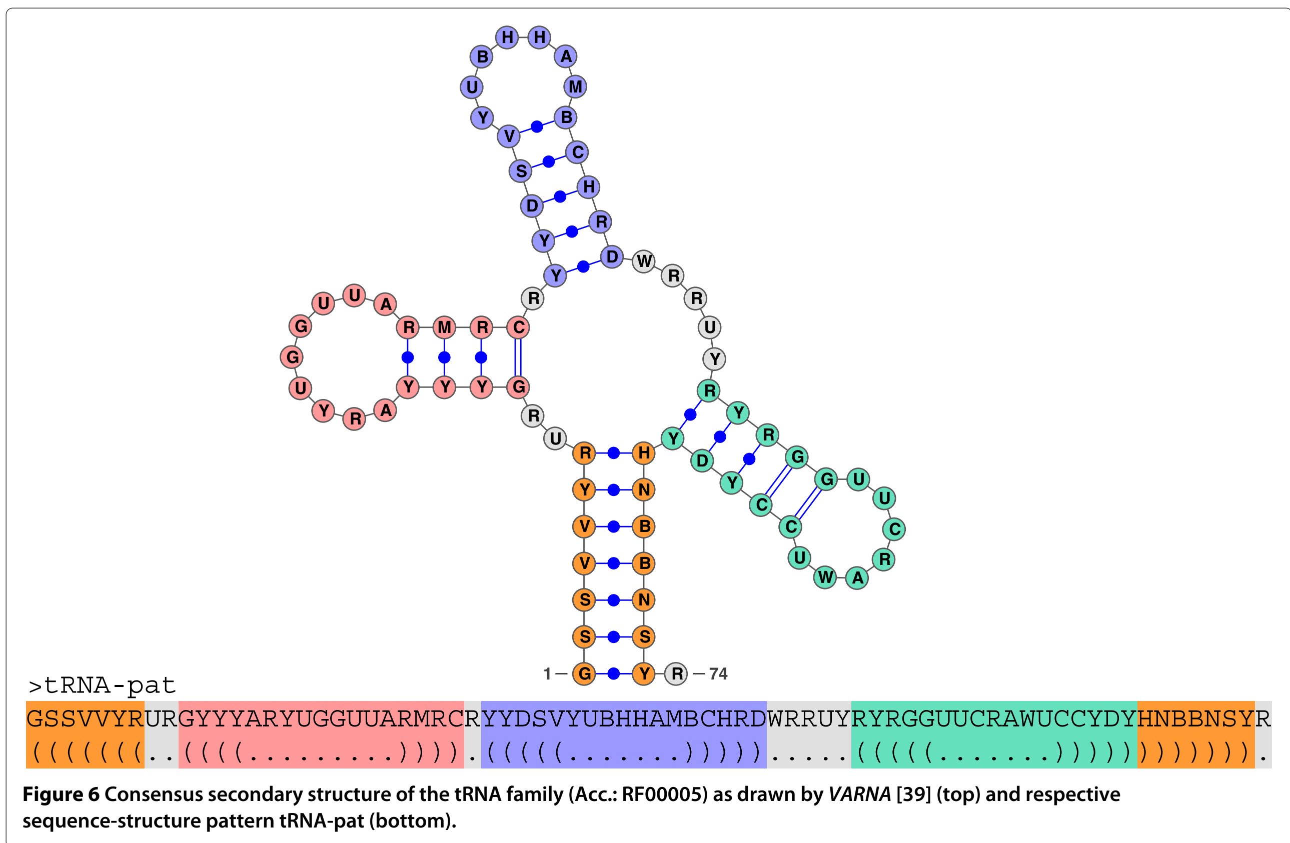

Figure 6 Consensus secondary structure of the tRNA family (Acc.: RF00005) as drawn by *VARNA* [39] (top) and respective sequence-structure pattern tRNA-pat (bottom).

the widely used tool *RNAMotif* [15] in terms of sensitivity and specificity in searches for the tRNA-pat depicted in Figure 6.

Scaling behavior of the online and index-based algorithms

In a second experiment we investigate how the search time of algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* scales on random subsets of RFAM10.1 of increasing size. The searched RSSPs flg1, flg2, and flg3 were derived from the three stem-loop substructures the members of family flg-Rhizobiales RNA motif (Acc.: RF01736) [40] fold into. These patterns differ in length, cost threshold $\mathcal{K}$ and number of allowed indels d; see Figure 8 for their definition, noting that $\mathcal{K}$ and d are simply denoted *cost* and *indels* in the *RaligNAtor* RSSP syntax. The results are shown in Figure 9 and Table S7 of Additional file 1. *LGSlinkAlign* and *LESAAlign* show a sublinear scaling behavior, whereas *LScanAlign* and *ScanAlign* scale linearly. The fastest algorithm is *LGSlinkAlign*, requiring only 11.68 (53.08) minutes to search for all three patterns in the smallest (full) subset. The second fastest algorithm is *LESAAlign*, followed by *LScanAlign* and *ScanAlign*, which require 32.27 (126.97), 40.47 (321.01), and 98.35 (754.66) minutes, respectively, to search for all the patterns in the smallest (full) subset. This corresponds to a speedup of 8.4 to 14.2 of *LGSlinkAlign* over *ScanAlign* on the smallest and the full subsets. Comparing the search time for pattern flg3 individually, the speedup of *LGSlinkAlign* over *ScanAlign* ranges from 22.6 to 38.8. We also observe that *ScanAlign* requires the longest time to match the longest pattern flg2 of length $m = 37$. The other algorithms profit from the early-stop computation approach to reduce the search time for this pattern on every database subset.

Influence of stem and loop lengths on the search time

When searching a database for matches of a given pattern, our algorithms compute the required *DP* matrices using recurrences according to two main cases: either a row corresponds to an unpaired or to a paired base of the pattern. To analyze the influence of the used recurrence on the search time of each algorithm, we search RFAM10.1 for artificial stem-loop patterns. Therefore we vary the number of bases in the loop of pattern $\mathcal{Q}$ = (NNNACANNN, (((...)))) from 3 to 12 by using As and Cs. Additionally, we vary the number of base pairs in the stem of pattern $\mathcal{Q}$ = (NNACANN, ((...))) from 2 to 11 by pairs of Ns. Matching the patterns in these two experiments means to increase the use of the *DP* recurrences in Equations (7) and (8), respectively. The cost threshold and the number of allowed indels

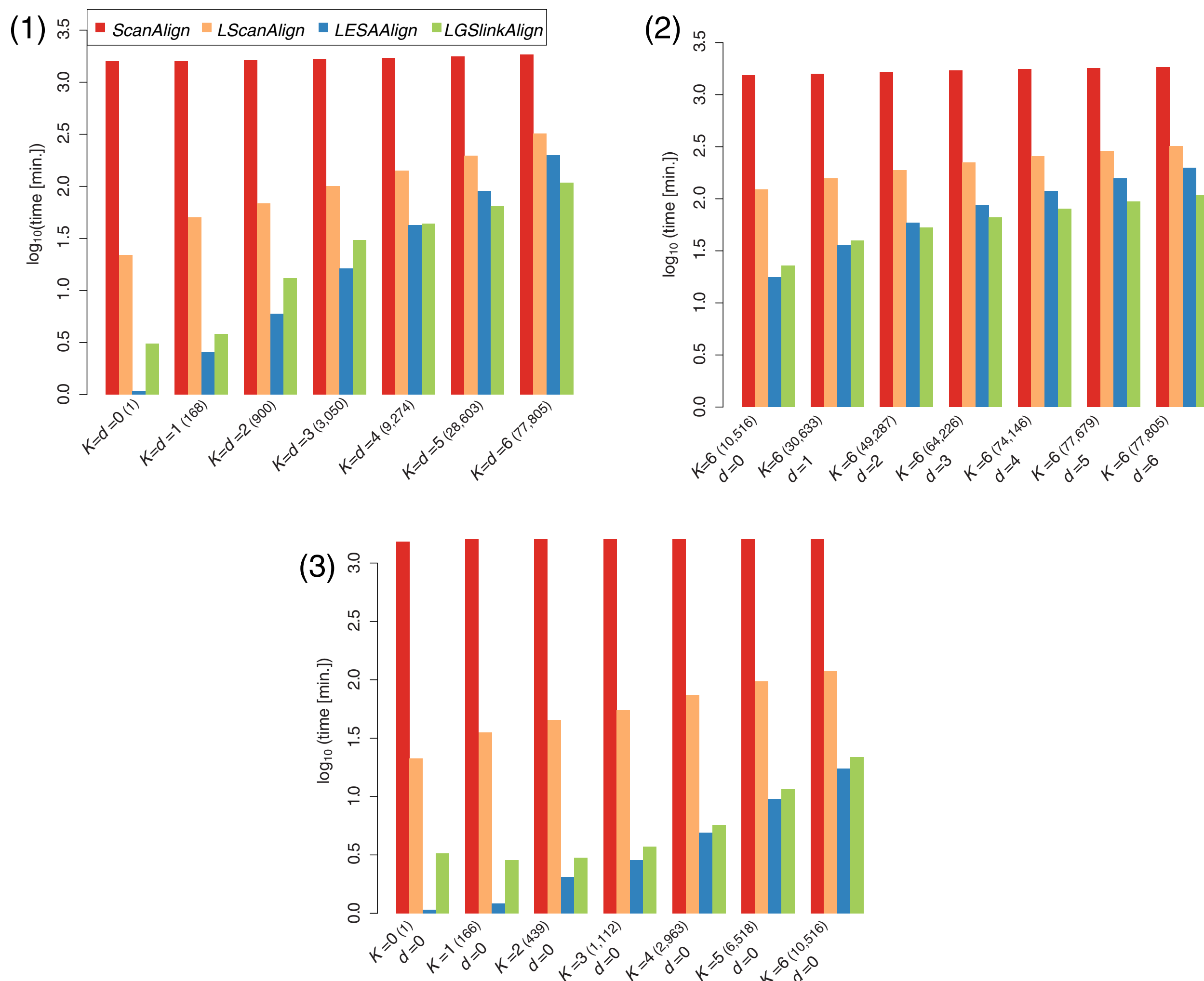

Figure 7 Running times (in minutes and $\log_{10}$ scale) needed by the different algorithms to search with an RSSP describing the tRNA in RFAM10.1. In (1) the cost threshold $\mathcal{K}$ and the number of allowed indels d are identical. In (2) $\mathcal{K} = 6$ is constant and d ranges from 0 to 6. In (3) $d = 0$ is constant and $\mathcal{K}$ ranges from 0 to 6. The numbers of resulting matches are given on the x-axes in brackets.

are fixed at $\mathcal{K} = d = 3$. Allowed base pairs are (A, U), (U, A), (C, G), and (G, C). The results are shown in Figure 10. We observe that increasing the number of bases in the loop has little influence and even reduces the running time of the two fastest algorithms *LGSlinkAlign* and *LESAAlign*. This can be explained by the use of the early-stop alignment computation scheme in these algorithms. The reduction of the running time is explained by the fewer matches that need to be processed as the pattern gets longer and more specific. For an increasing number of base pairs in the stem, *LGSlinkAlign* is the least affected algorithm. We also observe that the linear increase in running time of the basic online algorithm *ScanAlign*, caused by an extension of the pattern by one base pair, is similar to the effect of adding two bases in the loop.

RNA family classification by global chaining of RSSP matches

In the next experiment we show the effectiveness of global chaining when searching with two SSDs built for Rfam families Cripavirus internal ribosome entry site (Acc.: RF00458) and flg-Rhizobiales RNA motif (Acc.: RF01736) [40]. These two families present only 53% and 69% sequence identity, respectively, much below the average of $\sim$ 80% of the Rfam 10.1 families. This illustrates the importance of using both sequence and structure information encoded in the SSDs of this experiment. The SSD of family RF01736 comprises three RSSPs, denoted by flg1, flg2, and flg3 in Figure 8, derived from the three stem-loop substructures the members of this family fold into. The SSD of family RF00458 comprises five RSSPs, denoted by ires1, ires2, ires3, ires4, and ires5 in Figure S5 of Additional file 1, where the last four RSSPs describe the

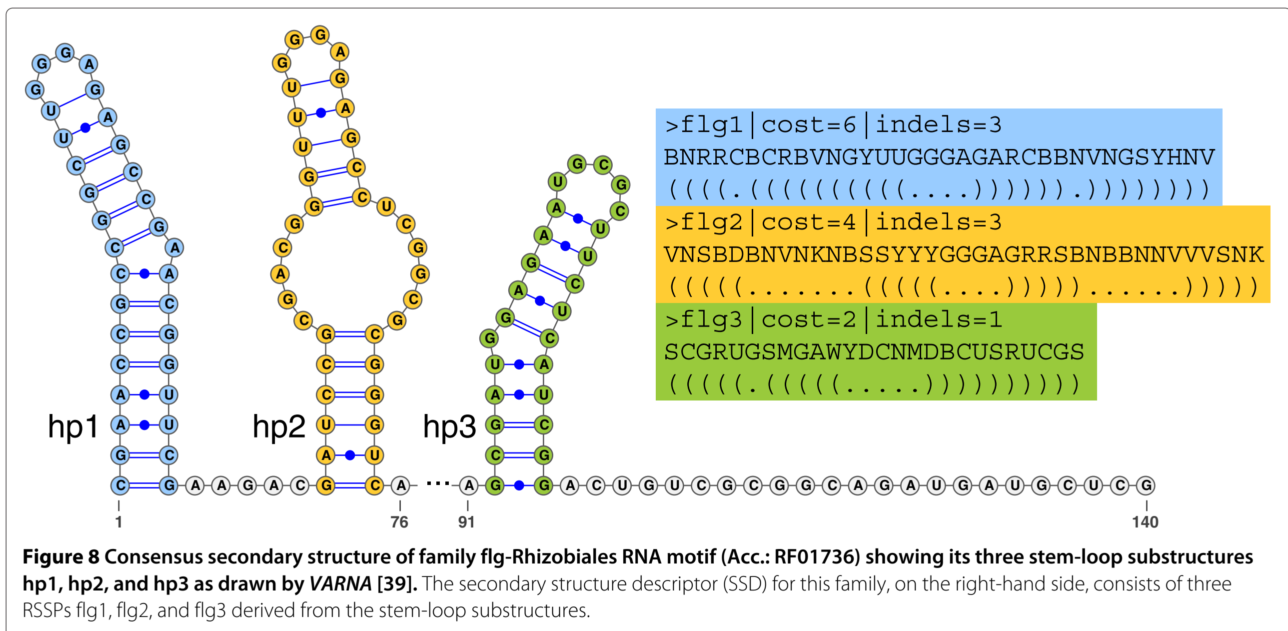

Figure 8 Consensus secondary structure of family flg-Rhizobiales RNA motif (Acc.: RF01736) showing its three stem-loop substructures hp1, hp2, and hp3 as drawn by *VARNA* [39]. The secondary structure descriptor (SSD) for this family, on the right-hand side, consists of three RSSPs flg1, flg2, and flg3 derived from the stem-loop substructures.

stem-loop substructures the members of this family fold into. ires1 describes a moderately conserved strand occurring in these members. Observe also in Figures 8 and S5 the cost threshold $\mathcal{K}$ and allowed number of indels d used per pattern, remembering that these are denoted *cost* and *indels* in the *RaligNAtor* RSSP syntax.

Searching with the SSD of family RF00458 in RFAM10.1 delivers $16,033,351$ matches for ires1, $8,950,417$ for ires2, $1,052$ for ires3, 112 for ires4, and $1,222,639$ for ires5. From these matches, *RaligNAtor* computes high-scoring chains of matches, eliminating spurious matches and resulting in exactly 17 chains. Each chain occurs in one of the 16 sequence members of the family in the full alignment except in sequence AF014388, where two chains with equal score occur. The highest (lowest) chain score is 171 (162). Using *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign*, the search for all five RSSPs requires 688.32, 585.59, 186.88, and 92.25 minutes, respectively, whereas chaining requires 13.66 seconds. See Table S8 of Additional file 1 for the time required to match each pattern using the different algorithms.

The same search is performed using the SSD of family RF01736. It results in $4,145$ matches for flg1, $68,024$ for flg2, and 67 for flg3. Chaining the matches leads to 15 chains occurring each in one of the 15 sequence members of the family in the full alignment. The highest (lowest) chain score is 163 (156). Using *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign*, the search for all three RSSPs requires 755.48, 336.69, 133.58, and 52.86 minutes, respectively, whereas chaining requires 0.03 seconds. The time required to match each pattern using each algorithm is reported in Table S9 of Additional file 1.

We also show that the lack of the sequence-structure edit operations supported by *RaligNAtor* deteriorates sensitivity and specificity in the search for sequence members of families RF00458 and RF01736. For this, we report in Section 4 and Table S10 of Additional file 1 results obtained with the *Structator* tool [23]. *Structator* is much faster but, in contrast to *RaligNAtor*, does not support all sequence-structure edit operations.

Importance of structural constraints for RNA family classification

To assess the potential of using RSSPs for reliable RNA homology search on a broader scale and to investigate the effect of using base pairing information, we evaluated *RaligNAtor* on 35 RNA families taken from Rfam 10.1 with different degrees of sequence identity and of different sizes. See Table 1 for more information about the selected families. In our experiment, we compared (1) *RaligNAtor* results obtained by using RSSPs derived from Rfam seed alignments with (2) results obtained for the same RSSPs ignoring base pairing information and (3) results obtained by *blastn* [41] searches with the families' consensus sequence. For each selected family, we automatically compiled an RSSP $\mathcal{Q} = (P, R)$ from the family's seed alignment using the following procedure: at each position of the RSSP's sequence pattern P, we choose the IUPAC wildcard matching all symbols in the corresponding alignment column. As structure string R, we use the secondary structure consensus available in the Rfam seed alignment. From the resulting RSSPs we remove the maximum prefix and suffix containing neither sequence information (i.e. IUPAC symbol N) nor base pairing information. To obtain

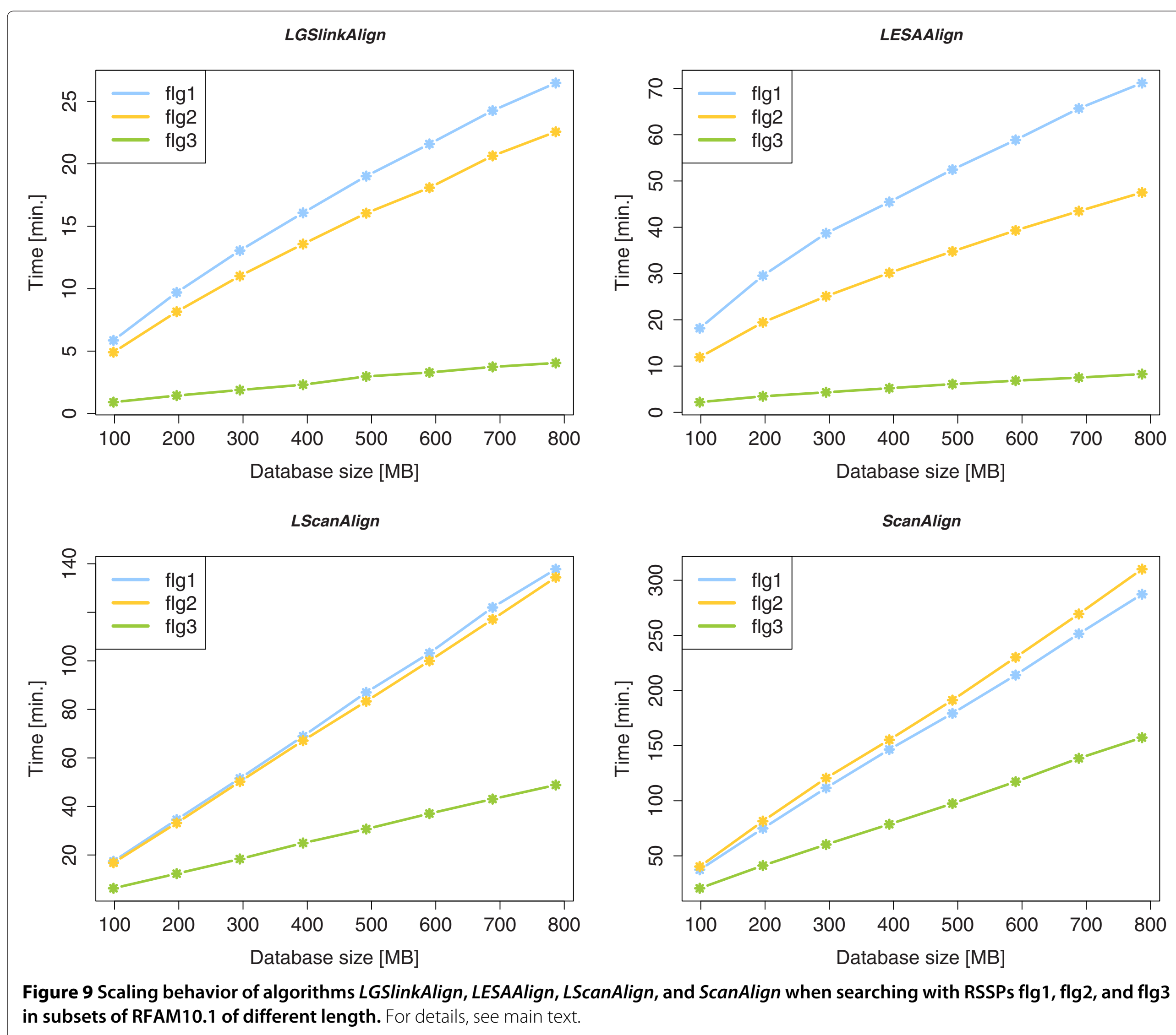

Figure 9 Scaling behavior of algorithms *LGSlinkAlign*, *LESAAlign*, *LScanAlign*, and *ScanAlign* when searching with RSSPs flg1, flg2, and flg3 in subsets of RFAM10.1 of different length. For details, see main text.

a query sequence for *blastn*, we compute the consensus sequence from the family's seed alignment. Because *blastn* does not appropriately handle IUPAC wildcard characters in the query, we choose the most frequent symbol occurring in a column as representative symbol in the consensus sequence. For the *RaligNAtor* searches, we adjust the cost threshold $\mathcal{K}$ and number of allowed indels d such that we match the complete family. That is, we achieve a sensitivity of 100%. The used operation costs are $\omega_d = \omega_m = 1$, $\omega_b = \omega_a = 2$, and $\omega_r = 3$. For the Blast searches, we called *blastn* with parameters -m8 -b 250000 -v 250000 and a very relaxed E-value cutoff of 1000. From the two *RaligNAtor* and one *blastn* outputs we count the number of true positives (#TPs) and false positives (#FPs) and compute ROC curves on the basis of the *RaligNAtor* score $\omega^*_{\mathcal{Q}} - dist(\mathcal{Q}, T)$ and the *blastn* bit score. See Table 1 and Figure 11 for the results of this experiment. A ROC curve with values averaged over all families is shown in Figure 11(1).

In addition, we show in Figures 11(2) and (3) the results of the ROC analysis for the families with the lowest and highest degree of sequence identity. For the ROC curve of each selected family, see Figures S7 and S8 of Additional file 1. Clearly, by using base pairing information, *RaligNAtor* achieves a higher sensitivity with a reduced false positive rate compared to searches ignoring base pairing (compare columns "*RaligNAtor*" and "*RaligNAtor* (sequence only)" in Table 1). This is in particular evident when searching for families with a low degree of sequence identity. This can be explained by the small amount of information left in the RSSP for such a family, once the structural information is removed. Due to the high variability of bases in the columns of the multiple alignment of the family, the pattern contains a large

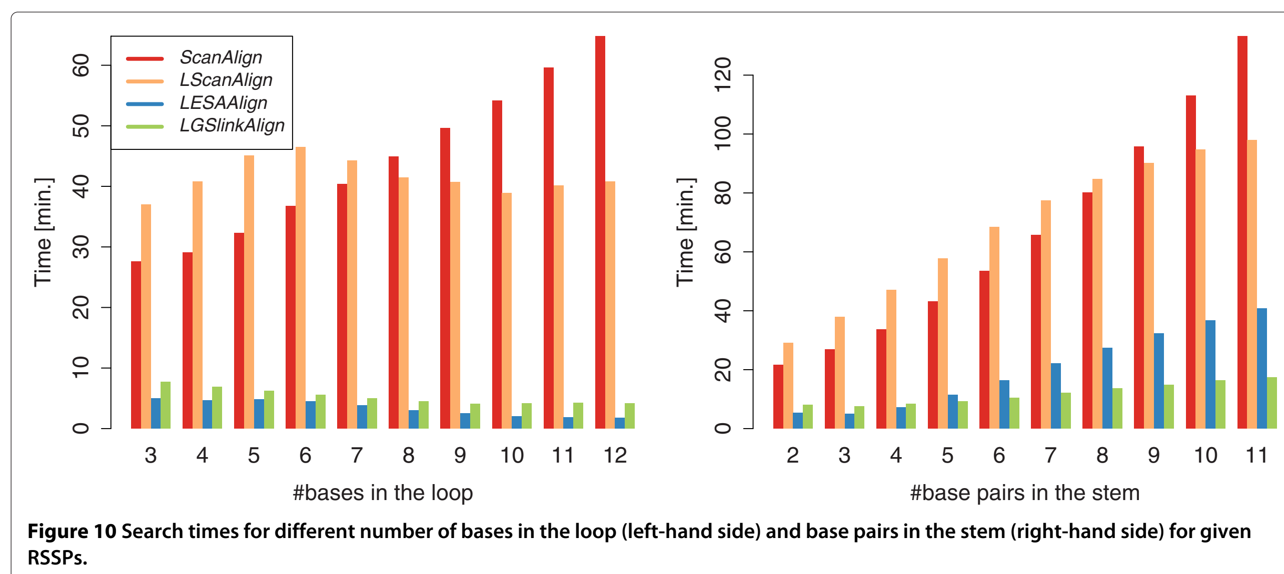

Figure 10 Search times for different number of bases in the loop (left-hand side) and base pairs in the stem (right-hand side) for given RSSPs.

number of wildcards. These symbols alone, without the constraints imposed by the base pairs, lead to unspecific patterns and therefore to a large number of false positives. We observe that, for families with sequence identity of up to 59%, the area under the curve (AUC) is considerably larger when base pairing information is taken into account. This difference decreases with increasing sequence identity (compare Figures 11(2) and (3)). Overall, the average AUC value over all families is, with a value of 0.93, still notably higher when base pairing information is considered compared to 0.89 if base pairing information is ignored (see Table 1). In this experiment, *blastn* only finds all members of those families whose sequence identity is at least 85%. This is due to the fact that *blastn* cannot appropriately handle IUPAC wildcard characters. Hence, by taking the most frequent symbol in an alignment column as consensus symbol, the heterogeneity of less conserved positions in the alignment cannot be adequately modeled. For the *blastn* searches, the average AUC value over all families is only 0.72.

RaligNAtor software package

RaligNAtor is an open-source software package for fast approximate matching of RNA sequence-structure patterns (RSSPs). It allows the user to search target RNA or DNA sequences choosing one of the new online or further accelerated index-based algorithms presented in this work. The index of the sequence to be searched can be easily constructed with program *sufconstruct* distributed with *RaligNAtor*.

Searched RSSPs can describe any (branching, non-crossing) RNA secondary structure; see examples in Figures 1, 6, 8, and S5 of Additional file 1. Bases composing the sequence information of RSSPs can be ambiguous IUPAC characters. As part of the search parameters for RSSPs, the user can specify the cost of each sequence-structure edit operation defined above, the cost threshold of possible matches, and the number of allowed indels. The RSSPs, along with costs and thresholds per RSSP, are specified in a simple text file using a syntax that is expressive but easy to understand as shown in the mentioned figures. Another possibility is to provide the same costs and thresholds for all searched patterns as parameters in the command line call to *RaligNAtor*. To ensure maximal flexibility, the user can also define the base pairing rules from an arbitrary subset of $\mathcal{A} \times \mathcal{A}$ as valid pairings in a separate text file. Searches can be performed on the forward and reverse strands of the target sequence. Searching on the reverse strand is implemented by reversal of the RSSP and transformation according to Watson-Crick base pairing. Wobble pairs {(G,U), (U,G)} automatically become {(C,A), (A,C)}. Due to these transformations, the index is built for one strand only.

For describing a complex RNA with our concept of secondary structure descriptor (SSD), i.e. with multiple RSSPs, the user specifies all RSSPs in one text file. The order of the RSSPs in the file will then specify the order of the RSSP matches used to build high-scoring chains. The chain score directly depends on the score of each match occurring in the chain. This is inversely proportional to the sequence-structure edit distance of the RSSP and its matching substring in the target sequence. Hence, higher scores indicate sequences with a higher conservation which are probably more closely related to the sought RNA family.

Chaining of matches discards spurious matches not occurring in any chain. An additional filtering option

Table 1 Results of *RaligNAtor* and *blastn* database searches for members of RNA families of different degrees of sequence identity in RFAM10.1

			RaligNAtor					*RaligNAtor* (sequence only)					*blastn*			
Family Acc.	**Size**	**Seq. ident.**	$\mathcal{K} = d$	**#TP**	**#FP**	**AUC**	**(pAUC)**	$\mathcal{K} = d$	**#TP**	**#FP**	**AUC**	**(pAUC)**	**#TP**	**#FP**	**AUC**	**(pAUC)**
RF00032	9,900	48%	3	9,900	1,088,131	0.95	(0.17)	3	9,900	2,723,135	0.82	(0.09)	3,000	68	0.29	(0.05)
RF00080	688	52%	33	688	698,942	0.71	(0.08)	19	688	1,279,375	0.60	(0.06)	326	540	0.42	(0.06)
RF02003	176	52%	21	176	1,174,167	0.53	(0.03)	6	176	1,168,093	0.32	(0.00)	28	814	0.11	(0.01)
RF00458	16	53%	20	16	88	0.94	(0.18)	14	16	2,688	0.96	(0.18)	12	1,224	0.73	(0.13)
RF00685	131	55%	18	131	40,952	0.98	(0.19)	7	131	103,276	0.97	(0.19)	88	2,945	0.63	(0.10)
RF00167	1,244	56%	25	1,244	2,514,701	0.58	(0.04)	17	1,244	2,611,256	0.28	(0.00)	660	624	0.52	(0.10)
RF01705	598	56%	26	598	2,704,796	0.49	(0.02)	17	598	2,698,712	0.42	(0.00)	57	60	0.08	(0.01)
RF01852	1,050	56%	22	1,050	1,026,233	0.99	(0.19)	14	1,050	1,488,254	0.94	(0.17)	543	83,268	0.44	(0.06)
RF01734	584	57%	10	584	2,614,228	0.69	(0.05)	5	584	2,668,392	0.46	(0.01)	201	114	0.30	(0.05)
RF00556	201	58%	8	201	69,808	0.97	(0.18)	6	201	1,514,311	0.92	(0.15)	91	1,024	0.44	(0.08)
RF00713	14	58%	27	14	10,349	0.99	(0.19)	18	14	16,477	0.88	(0.16)	13	552	0.92	(0.18)
RF00170	41	59%	13	41	53	0.97	(0.18)	9	41	9,197	0.96	(0.18)	29	176	0.70	(0.14)
RF00706	69	59%	13	69	1	1.00	(0.20)	9	69	12	0.97	(0.19)	66	194	0.95	(0.18)
RF00747	29	59%	20	29	130	0.97	(0.18)	16	29	159,898	0.96	(0.18)	28	236	0.96	(0.19)
RF00778	20	59%	33	20	394,560	0.93	(0.17)	23	20	167,029	0.79	(0.13)	17	390	0.84	(0.16)
RF01065	118	59%	17	118	0	1.00	(0.20)	9	118	0	1.00	(0.20)	70	305	0.59	(0.11)
RF01733	9	63%	9	9	0	1.00	(0.20)	7	9	0	1.00	(0.20)	7	918	0.77	(0.15)
RF00522	415	67%	5	415	1,461	0.99	(0.19)	5	415	32,224	0.99	(0.19)	359	391	0.63	(0.10)
RF01862	15	67%	7	15	0	1.00	(0.20)	5	15	0	1.00	(0.20)	10	82	0.66	(0.13)
RF00104	406	69%	24	406	989,362	0.99	(0.19)	14	406	1,560,674	0.99	(0.19)	237	72	0.45	(0.07)
RF00165	431	69%	9	431	0	1.00	(0.20)	8	431	1	0.99	(0.19)	318	192	0.73	(0.14)
RF01185	108	69%	13	108	24,759	0.99	(0.19)	13	108	24,759	0.99	(0.19)	104	329	0.93	(0.18)
RF01838	77	69%	4	77	0	1.00	(0.20)	4	77	0	1.00	(0.20)	77	172	1.00	(0.20)
RF02031	164	71%	17	164	297,941	0.99	(0.19)	12	164	521,018	0.99	(0.19)	100	218	0.60	(0.11)
RF00052	210	72%	16	210	0	1.00	(0.20)	12	210	0	1.00	(0.20)	207	12,496	0.98	(0.19)
RF00543	103	73%	26	103	0	1.00	(0.20)	19	103	0	1.00	(0.20)	102	110	0.99	(0.19)
RF01744	14	73%	7	14	0	1.00	(0.20)	5	14	0	1.00	(0.20)	11	5,377	0.74	(0.14)
RF01769	149	75%	16	149	0	1.00	(0.20)	10	149	0	1.00	(0.20)	149	150	0.99	(0.19)
RF00110	161	81%	19	161	0	1.00	(0.20)	17	161	0	1.00	(0.20)	160	791	0.99	(0.19)
RF01967	50	84%	37	50	660,130	0.98	(0.19)	26	50	475,242	0.98	(0.19)	48	691	0.95	(0.19)
RF01472	26	85%	6	26	0	1.00	(0.20)	1	26	0	1.00	(0.20)	26	412	1.00	(0.20)

Table 1 Results of *RaligNAtor* and *blastn* database searches for members of RNA families of different degrees of sequence identity in RFAM10.1 *(Continued)*

RF01953	46	85%	32	46	0	1.00	(0.20)	22	46	0	1.00	(0.20)	46	772	1.00	(0.20)
RF00372	45	86%	28	45	0	1.00	(0.20)	24	45	0	1.00	(0.20)	45	197	0.99	(0.19)
RF01980	43	86%	39	43	830,971	0.97	(0.19)	28	43	702,352	0.96	(0.19)	43	341	1.00	(0.20)
RF00469	1,366	89%	12	1,366	46,351	0.99	(0.19)	7	1,366	99,045	0.99	(0.19)	1,341	474	0.97	(0.19)
Average		66%				0.93	(0.17)				0.89	(0.16)			0.72	(0.14)

Searches are performed using *RaligNAtor* with and without base pairing information (column "*RaligNAtor* (sequence only)") and using program *blastn* with the families' seed alignment consensus sequence as query. Column "size" indicates the number of members in a family. Column "seq. ident." gives the families' sequence identity as listed in the Rfam database. #TP and #FP stand for number of found true and false positives, respectively. AUC is the area under the curve of the corresponding ROC curves shown in Figures 11, S7, and S8 of Additional file 1. Column pAUC gives the partial area under the curve up to a false positive rate of 20%. For additional details, see main text.

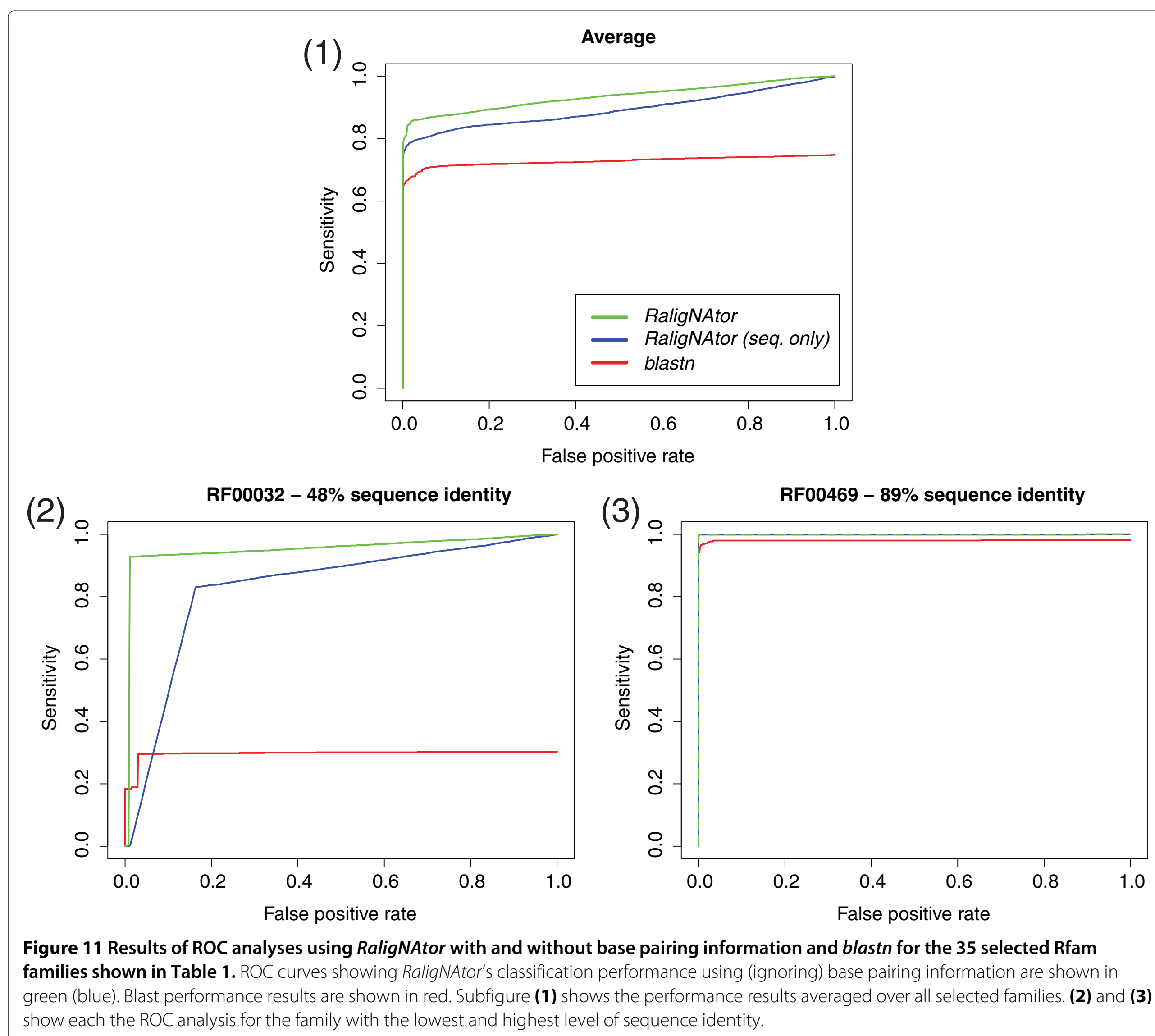

Figure 11 Results of ROC analyses using *RaligNAtor* with and without base pairing information and *blastn* for the 35 selected Rfam families shown in Table 1. ROC curves showing *RaligNAtor*'s classification performance using (ignoring) base pairing information are shown in green (blue). Blast performance results are shown in red. Subfigure **(1)** shows the performance results averaged over all selected families. **(2)** and **(3)** show each the ROC analysis for the family with the lowest and highest level of sequence identity.

eliminates matches overlapping another with a higher score for the same RSSP. This is particularly useful when indels lead to almost identical matches that are only shifted by a few positions in the target sequence.

The output of *RaligNAtor* includes not only matching positions to single RSSPs and chains, but their sequence-structure alignment to the matched substrings as well. In the *RaligNAtor* software package, all programs for searching patterns support multithreading to take advantage of computer systems with multiple CPU cores. There are two modes of parallelism. At first, different patterns are searched using multiple threads. Additionally, the search space (i.e. the sequence for the online algorithms and the index structure for the index-based methods) is partitioned, processing each part using a different thread. Lastly, we remark that our software also provides an implementation of the original algorithm of Jiang *et al.* for global sequence-structure alignment [25], easily applicable by the user.

Conclusions

We have presented new index-based and online algorithms for fast approximate matching of RNA sequence-structure patterns. Our algorithms, all implemented in the *RaligNAtor* software, stand out from previous search tools based on motif descriptors by supporting a full set of edit operations on single bases and base pairs. See Table 2 for an overview of the algorithms. In each algorithm, the application of a new computing scheme to optimally reuse the entries of the required dynamic programming matrices and an early-stop technique to avoid the alignment computation of non-matching substrings

Table 2 Overview of the presented algorithms

Algorithm	Online	Indexed	Early-stop Acceleration	Additional memory Requirements [bytes]	Used index tables			
					suf	lcp	suf^{-1}	vtab
ScanAlign	✓			0				
LScanAlign	✓		✓	0				
LESAAlign		✓	✓	$5n$	✓	✓		
LGSlinkAlign		✓	✓	$9.125n$	✓	✓	✓	✓

The two online algorithms *ScanAlign* and *LScanAlign* need no additional memory except for the searched sequence of length n. Column *additional memory requirements* refers to the additional memory needed by the used index tables. Recall that tables suf and suf^{-1} require $4n$ bytes each. Table lcp can be stored in $1n$ bytes and the bit array vtab requires only n bits ($= 0.125n$ bytes).

led to considerable speedups compared to the basic scanning algorithm *ScanAlign*. Our experiments show superior performance of the index-based algorithms *LGSlinkAlign* and *LESAAlign*, which employ the suffix array data structure and achieve running time sublinear in the length of the target database. When searching for approximate matches of biologically relevant patterns on the Rfam database, *LGSlinkAlign* (*LESAAlign*) was faster than *ScanAlign* and *LScanAlign* by a factor of up to 560 (1,323) and 17 (29), respectively (see Figure 7). Comparing the two index-based algorithms, *LESAAlign* was faster than *LGSlinkAlign* when searching with tight cost threshold (i.e. sequence-structure edit distance) and no allowed indels, but became considerably slower when the number of allowed indels was increased. In this scenario, *LGSlinkAlign* was faster than *LESAAlign* by up to 4 times. In regard to the two online algorithms, *LScanAlign* was faster than *ScanAlign* by up to factor 45. In summary, *LGSlinkAlign* is the best performing algorithm when searching with diverse thresholds, whereas *LScanAlign* is a very fast and space-efficient alternative. *RaligNAtor* also allows to use the powerful concept of RNA secondary descriptors [23], i.e. searching for multiple ordered sequence-structure patterns each describing a substructure of a larger RNA molecule. For this, *RaligNAtor* integrates fast global and local chaining algorithms. We further performed experiments using *RaligNAtor* to search for members of RNA families based on information from the consensus secondary structure. In these experiments, *RaligNAtor* showed a high degree of sensitivity and specificity. Compared to searching with primary sequence only, the use of secondary structure information considerably improved the search sensitivity and specificity, in particular for families with a characteristic secondary structure but low degree of sequence conservation. We remark that, up to now, *RaligNAtor* uses a relatively simple scoring scheme. By incorporating more fine grained scoring schemes like RIBOSUM [13] or energy based scoring [42], we believe that the performance of *RaligNAtor* for RNA homology search can be further improved.

Beyond the algorithmic contributions, we provide with the *RaligNAtor* software distribution, a robust, well-documented, and easy-to-use software package implementing the ideas and algorithms presented in this manuscript.

Availability

The *RaligNAtor* 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/ralignator for details.

Additional file

Additional file 1: Supplemental material. Additional file 1 contains additional experiments, figures, and tables.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
FM and MB developed the algorithms. FM implemented the algorithms. SK implemented the chaining algorithms. MB initiated the project and provided supervision and guidance. All three authors contributed to the manuscript. All authors read and approved the final manuscript.

Acknowledgements
This work was supported by basic funding of the University of Hamburg. We thank Steffen Dettmann for interesting discussions and algorithmic ideas that contributed to this work.

Received: 27 February 2013 Accepted: 11 July 2013
Published: 17 July 2013

References

1. Mattick J: **RNA regulation: a new genetics?** *Nat Rev Genet* 2004, **5**(4):316–323.
2. Burge SW, Daub J, Eberhardt R, Tate J, Barquist L, Nawrocki EP, Eddy SR, Gardner PP, Bateman A: **Rfam 11.0: 10 years of RNA families.** *Nucleic Acids Res* 2012, **41**(D1).
3. Siebert S, Backofen R: **MARNA: multiple alignment and consensus structure prediction of RNAs based on sequence structure comparisons.** *Bioinformatics* 2005, **21**(16):3352–3359.
4. Höchsmann M, Voss B, Giegerich R: **Pure multiple RNA secondary structure alignments: a progressive profile approach.** *IEEE/ACM Trans Comput Bio Bioinformatics* 2004, **1:**53–62.

5. Sankoff D: **Simultaneous solution of the RNA folding, alignment and protosequence problem.** *SIAM J Appl Mathe* 1985, **45:**810–825.
6. 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+.
7. Havgaard JH, Torarinsson E, Gorodkin J: **Fast pairwise structural RNA alignments by pruning of the dynamical programming matrix.** *PLoS Comput Biol* 2007, **3**(10):e193+.
8. Torarinsson E, Havgaard JH, Gorodkin J: **Multiple structural alignment and clustering of RNA sequences.** *Bioinformatics* 2007, **23**(8):926–932.
9. Mathews DH, Turner DH: **Dynalign: an algorithm for finding the secondary structure common to two RNA sequences.** *J Mol Biol* 2002, **317**(2):191–203.
10. Mathews DH: **Predicting a set of minimal free energy RNA secondary structures common to two sequences.** *Bioinformatics* 2005, **21**(10):2246–2253.
11. Dalli D, Wilm A, Mainz I, Steger G: **STRAL: progressive alignment of non-coding RNA using base pairing probability vectors in quadratic time.** *Bioinformatics* 2006, **22**(13):1593–1599.
12. Nawrocki EP, Kolbe DL, Eddy SR: **Infernal 1.0: inference of RNA alignments.** *Bioinformatics* 2009, **25**(10):1335–1337.
13. Klein R, Eddy S: **RSEARCH: finding homologs of single structured RNA sequences.** *BMC Bioinformatics* 2003, **4:**44.
14. 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.
15. Macke T, Ecker D, Gutell R, Gautheret D, Case D, Sampath R: **RNAMotif – A new RNA secondary structure definition and discovery algorithm.** *Nucleic Acids Res* 2001, **29**(22):4724–4735.
16. 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):325–331.
17. **RNABOB: a program to search for RNA secondary structure motifs in sequence databases.** [http://selab.janelia.org/software.html]
18. Chang T, Huang H, Chuang T, Shien D, Horng J: **RNAMST: efficient and flexible approach for identifying RNA structural homologs.** *Nucleic Acids Res* 2006, **34:**W423–W428.
19. Dsouza M, Larsen N, Overbeek R: **Searching for patterns in genomic data.** *Trends Genet* 1997, **13**(12):497–498.
20. Grillo G, Licciulli F, Liuni S, SbisÃă E, Pesole G: **PatSearch: A program for the detection of patterns and structural motifs in nucleotide sequences.** *Nucleic Acids Res* 2003, **31**(13):3608–3612.
21. Billoud B, Kontic M, Viari A: **Palingol: a declarative programming language to describe nucleic acids' secondary structures and to scan sequence database.** *Nucleic Acids Res* 1996, **24**(8):1395–1403.
22. Reeder J, Giegerich R: **A graphical programming system for molecular motif search.** In *Proceedings of the 5th international Conference on Generative Programming and Component Engineering*. New York: ACM Press; 2006:131–140.
23. Meyer F, Kurtz S, Backofen R, Will S, Beckstette M: **Structator: fast index-based search for RNA sequence-structure patterns.** *BMC Bioinformatics* 2011, **12:**214.
24. El-Mabrouk N, Raffinot M, Duchesne JE, Lajoie M, Luc N: **Approximate matching of structured motifs in DNA sequences.** *J Bioinform Comput Biol* 2005, **3**(2):317–342.
25. Jiang T, Lin G, Ma B, Zhang K: **A general edit distance between RNA structures.** *J Comput Biol* 2002, **9**(2):371–388.
26. Abouelhoda M, Ohlebusch E: **Chaining algorithms for multiple genome comparison.** *J Discrete Algo* 2005, **3**(2–4):321–341.
27. Will S, Siebauer M, Heyne S, Engelhardt J, Stadler P, Reiche K, Backofen R: **LocARNAscan: incorporating thermodynamic stability in sequence and structure-based RNA homology search.** *Algo Mol Biol* 2013, **8:**14.
28. Ukkonen E: **Algorithms for approximate string matching.** *Inf Control* 1985, **64**(1–3):100–118.
29. Manber U, Myers E: **Suffix arrays: a new method for on-line string searches.** *SIAM J Comput* 1993, **22**(5):935–948.
30. Abouelhoda M, Kurtz S, Ohlebusch E: **Replacing suffix trees with enhanced suffix arrays.** *J Discrete Algo* 2004, **2:**53–86.
31. Kärkkäinen J, Sanders P: **Simple linear work suffix array construction.** In *Proceedings of the 13th International Conference on Automata, Languages and Programming*. Berlin - Heidelberg: Springer; 2003.
32. Puglisi SJ, Smyth W, Turpin A: **The performance of linear time suffix sorting algorithms.** In *DCC '05: Proceedings of the Data Compression Conference*. Washington: IEEE Computer Society; 2005:358–367.
33. Manzini G, Ferragina P: **Engineering a lightweight suffix array construction algorithm.** *Algorithmica* 2004, **40:**33–50.
34. Fischer J: **Wee LCP.** *Inf Proc Let* 2010, **110**(8–9):317–320.
35. 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*. Berlin - Heidelberg: Springer; 2001:181–192.
36. 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.
37. Ukkonen E: **On-line construction of suffix trees.** *Algorithmica* 1995, **14**(3):249–260.
38. 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.
39. Darty K, Denise A, Ponty Y: **VARNA: Interactive drawing and editing of the RNA secondary structure.** *Bioinformatics* 2009, **25**(15):1974–1975.
40. Weinberg Z, Wang J, Bogue J, Yang J, Corbino K, Moy R, Breaker R: **Comparative genomics reveals 104 candidate structured RNAs from bacteria, archaea, and their metagenomes.** *Genome Biol* 2010, **11**(3):R31.
41. Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ: **Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.** *Nucleic Acids Res* 1997, **25**(17):3389–3402.
42. Mathews DH, Turner DH: **Prediction of RNA secondary structure by free energy minimization.** *Curr Opin Struct Biol* 2006, **16**(3):270–278.

doi:10.1186/1471-2105-14-226
Cite this article as: Meyer *et al.*: Fast online and index-based algorithms for approximate search of RNA sequence-structure patterns. *BMC Bioinformatics* 2013 **14**:226.

Submit your next manuscript to BioMed Central and take full advantage of:

- **Convenient online submission**
- **Thorough peer review**
- **No space constraints or color figure charges**
- **Immediate publication on acceptance**
- **Inclusion in PubMed, CAS, Scopus and Google Scholar**
- **Research which is freely available for redistribution**

Submit your manuscript at
www.biomedcentral.com/submit