Efficient methods for matching RNA sequence-structure patterns

Dissertation
zur Erlangung des akademischen Grades
Dr. rer. nat.
an der Fakultät
für Mathematik, Informatik und Naturwissenschaften der
Universität Hamburg

eingereicht beim Fach-Promotionsausschuss Informatik von
Fernando Meyer
aus Balneário Camboriú, Brasilien

Juni 2014

Gutachter:
Dr. Michael Beckstette
Prof. Dr. Stefan Kurtz
Prof. Dr. Jan Gorodkin

Tag der Disputation:
16. April 2015

Zusammenfassung

Die Sekundärstruktur eines RNA Moleküls ist eng mit seiner Funktion verbunden und häufig stärker konserviert als die Sequenz. Folglich ist für die wichtige Aufgabe der Datenbanksuche nach funktionell ähnlichen RNAs, welche sich evolutionär von einem gemeinsamen Vorgängermolekül entwickelt haben (RNA Homologiesuche), die Suche nach Sequenz-Struktur-Mustern von großer Bedeutung. Allerdings verfügen aktuelle Werkzeuge für diese Aufgabe nur über ein Laufzeitverhalten, welches im besten Fall linear von der Größe der zu durchsuchenden Sequenzdatenbank abhängt. Deshalb sind sie häufig wenig geeignet für die Suche in großen Datenbanken. Der Grund hierfür ist der Verzicht auf Index-Datenstrukturen zur Beschleunigung der Suche. Ein weitere Nachteil aktueller Werkzeuge zur Suche mit Sequenz-Struktur-Mustern, welcher insbesondere ein Hindernis bei sensitiven und spezifischen Suchen darstellt, ist die nur sehr eingeschränkt vorhandene Möglichkeit approximative Treffer struktureller RNA Suchmuster zu finden.

In dieser Arbeit präsentiere ich neue Methoden und direkt einsetzbare Werkzeuge zur schnellen Suche mit RNA Sequenz-Struktur Mustern in großen Sequenzdatenbanken. Die erste vorgestellte Methode basiert auf Affix-Arrays, einer relativ neuen Indexdatenstruktur, welche durch Vorverarbeitung der Zieldatenbank erstellt wird. Im Gegensatz zu etablierten Indexdatenstrukturen wie Suffixbäumen oder arrays, unterstützen Affix-Arrays die bidirektionale Mustersuche. Diese ist notwendig, um die strukturellen Nebenbedingungen eines Suchmusters effizient zu berücksichtigen. Strukturelle Muster, wie zum Beispiel Stem-Loops können von innen nach außen gesucht werden, sodass zuerst die innere Loop Region und dann die paarenden Basen des Stem-Bereiches konsekutiv gesucht werden. Diese Vorgehensweise erlaubt das Ausnutzen von Basenpaarinformationen, um den Suchraum zu reduzieren und führt zu einer erwarteten Laufzeit, welche sich sublinear zur Größe der zu durchsuchenden Sequenzdatenbank verhält. Um die Beschreibung von RNA Molekülen, welche in komplexere Sekundärstrukturen mit multiplen Sequenz-Struktur Mustern falten, zu unterstützen, wurde eine neue Methode zur Verkettung (Chaining) von Mustertreffern in die Mustersuche integriert. Durch die Verkettung werden zufällige Mustertreffer, insbesondere hervorgerufen durch unspezifische Muster, aus der Menge von Zwischenresultaten entfernt. In Benchmark-Experimenten auf der Rfam Datenbank war unsere Methode um bis zu zwei Größenordnungen schneller als bisherige Methoden.

Während die erste in dieser Arbeit vorgestellte Methode zur effizienten Suche mit Sequenz-Struktur-Mustern sehr schnell ist, verfügt sie nur über beschränkte Möglichkeiten approximative Treffer eines RNA Suchmusters, welche zum Beispiel Insertionen/Deletionen an beliebigen Positionen oder die Sekundärstruktur verändernde Mutationen enthalten, zu finden. Diese Einschränkung erlaubt

oftmals nicht die Beschreibung einer RNA Familie mit einem Muster, welches sowohl sensitiv, als auch spezifisch genug ist, um alle Familienmitglieder zu finden. Aus diesem Grund habe ich neue indexbasierte und online Verfahren zur approximativen Suche mit Sequenz-Struktur-Mustern entwickelt, welche Edit-Operationen auf Einzelbasen und Basenpaaren erlauben. Aufgrund des hohen Berechnungsaufwands des hierfür erforderlichen Sequenz-Struktur-Alignments, berechnet das vorgestellte Verfahren effizient nur semi-globale Alignments zwischen strukturellen RNA Mustern und Teilworten der zu durchsuchenden Sequenz, deren Alignmentkosten einen benutzerspezifischen Schwellwert nicht überschreiten. Hierzu wird ein neues, auf dynamischer Programmierung (DP) basierendes Berechnungsschema vorgestellt, welches (1) die Einträge der DP-Matrizen wieder verwendet und (2) die Alignmentberechnung für Teilworte, welche keinen Treffer erzeugen können, vermeidet. Dieses neue Verfahren verwendet ein aus der zu durchsuchenden Sequenzdatenbank generiertes Suffix-Arrays und erzielt eine Laufzeit, welche sublinear mit der Größe der zu durchsuchenden Sequenzen skaliert. Des Weiteren enthalten alle vorgestellten Algorithmen unseren neuen Ansatz zum Verketten von Mustertreffern. Laufzeitexperimente zeigen, dass unser bestes indexbasiertes Verfahren um zwei bis drei Größenordnungen schneller ist als bisherige Methoden.

Abstract

The secondary structure of RNA molecules is intimately related to their function and often more conserved than the sequence. Hence, the important task of searching databases for functionally related RNAs evolving from a common ancestor, i.e. RNA homology search, requires to match sequence-structure patterns. However, current tools for this task have, in the best case, a running time that is only linear in the size of the sequence databases. Consequently, they are not well suited for searching large databases. This can be explained by their failure to use an appropriate index data structure for fast searches. Furthermore, a disadvantage of current tools for matching sequence-structure patterns is their limited capacity to find approximate matches to structural RNA patterns, which poses an obstacle to sensitive and specific searches.

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

While our first method is extremely fast, it has limited capacity to find approximate matches to RNA patterns, such as matches with insertions or deletions at arbitrary positions relative to the pattern or mutations affecting the secondary structure. This limitation often does not allow to define patterns that are specific and sensitive enough to match the sequences belonging to the sought RNA family. Therefore, we have developed novel 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. Due to the high computational cost of the underlying sequence-structure alignment problem, our algorithms 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 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 algorithms exploit suffix arrays preprocessed from the target database and achieve running times that are sublinear in the size of the searched sequences. Moreover, all the new algorithms integrate our approach for chaining matches. Benchmark experiments show that our best index-based algorithm is two to three orders of magnitude faster than previous methods.

Acknowledgments

I thank my main supervisor, Dr. Michael Beckstette, for introducing me to interesting research topics and providing guidance throughout the years. I also thank Prof. Stefan Kurtz for his valuable advisory and contributions to the carried research projects. I thank Dr. Sebastian Will for his contributions to the *Structator* project. I thank Dr. Steffen Dettmann for his contributions that led to the *RaligNAtor* project. I thank my former office colleagues, Sascha Steinbiß and Dirk Willrodt, for their readiness to help at all times and for proofreading parts of my thesis. I thank the Center for Bioinformatics of the University of Hamburg and Prof. Matthias Rarey for the financial and infrastructural support required for carrying my research. I thank Karin Lundt not only for doing part of the office work but also for her friendliness and support. Finally, I thank my girlfriend, Rika Paulisch, for her support and patience and my parents for their support from abroad.

Publications

- **Fernando Meyer**, Stefan Kurtz, and Michael Beckstette. Fast online and index-based algorithms for approximate search of RNA sequence-structure patterns. *BMC Bioinformatics*, 14(1):226, 2013.
- **Fernando Meyer**, Stefan Kurtz, Rolf Backofen, Sebastian Will, and Michael Beckstette. Structator: fast index-based search for RNA sequence-structure patterns. *BMC Bioinformatics*, 12(1):214, 2011. This article was selected for the *Highlight Track - Research Highlights* of the *International German Conference on Bioinformatics*, 2011.

Both publications acquired “highly accessed” designation from the publisher.

Contents

List of Figures

1 Introduction

1.1 RNAs and their manifold functions

Following the pioneering work of Crick, there was for a long time a general belief that the primary function of RNAs was to carry information from DNA to proteins [1, 2]. This was an assumption of the *central dogma of molecular biology*, according to which in most cells genetic information can only flow from DNA to RNA and from this to protein. By the late 1970s, three types of RNAs were known and relatively well understood:

- messenger RNA (mRNA), the carrier of information from DNA to protein;
- ribosomal RNA (rRNA), the RNA component of the ribosome, which is a machinery that synthesizes proteins by converting triplets of bases in the order specified by the mRNA into chains of amino acids; and
- transfer RNA (tRNA), an RNA that carries an amino acid to ribosomes and mediates its recognition to the corresponding base triplet.

RNA molecules were, therefore, classified as *protein-coding* (mRNA) and *non-protein-coding* (tRNA and rRNA) or simply *non-coding*.

The focus on proteins was consistent with the conviction that they had unique importance in living organisms by controlling the majority of regulatory transactions and being the main contributors to organism complexity. In 1972, Ohno even used the term *junk DNA* to denote untranslatable parts of DNA and, a few years later, Orgel and Crick similarly classified pieces of DNA either as encoding proteins by occurring as mRNA or as useless [3, 4]. However, in 1977 the question of the purpose of the "useless parts" became increasingly intriguing with the discovery of Sharp and Roberts stating that genes could be discontinuous in the genome [5, 6]. Their works, for which Sharp and Roberts received a Nobel Prize in 1993, led to the discovery of the process of splicing and the fact that, unlike in prokaryotes, most of the DNA in eukaryotes does not code for proteins. Indeed, only $\sim 1.5\%$ of the human genome is estimated to code for proteins [7], but 83% to 85% is estimated to be transcribed [8, 9]. This suggests that there exists a huge number of non-coding RNAs, whose functions in humans and other organisms we have just started to understand. Nevertheless, we can already recognize RNAs as extremely important molecules.

Diverse findings have radically changed our views about RNAs, now known to participate in many cellular processes. Certain RNAs, for example, can catalyze biochemical reactions similarly to

protein enzymes. The first evidence for these RNAs, called ribozymes, was given by Cech, who showed that a portion of an RNA can have enzyme-like properties that allow self-splicing, removing non-coding parts (introns) of a pre-messenger RNA for the formation of a mature mRNA [10]. Thereafter, Altman showed that RNase P, a kind of ribozyme, acts in the maturation of tRNAs [11]. For their discoveries, Cech and Altman were awarded a Nobel Prize in 1989. Many other ribozymes were later also discovered [12]. Due to the capacity of RNAs to store genetic information similarly to DNA and in particular due to the discoveries of Cech and Altman disregarding the need for enzymatic proteins for RNA replication (and therefore replication of genetic information), Gilbert hypothesized an *RNA world* [13]. According to this hypothesis, RNAs may have pre-existed DNA and proteins, until DNA undertook their role as information carrier due to its increased chemical stability, whereas proteins could become more specialized molecules due to the variety of amino acids they are made of. RNAs, therefore, may have played major roles in the evolution of cellular life.

Further discoveries also indicate RNAs as fundamental agents in life evolution. Contradicting the *central dogma*, it is now known that reverse transcription, i.e. generation of DNA from RNA, occurs in all domains of life [14]. Already in 1970, Temin and Baltimore independently discovered an RNA-dependent DNA polymerase called reverse transcriptase, an enzyme that enables reverse transcription [15, 16]. Their works helped to understand the replication of viruses whose genetic information is stored not in DNA, but in RNA. These so-called retroviruses use reverse transcriptase to replicate themselves in the form of DNA integrated in a host genome. The discovery of reverse transcriptase, for which Temin and Baltimore received a Nobel Prize in 1975, had a huge impact on the research of tumor-causing viruses. The activity of this enzyme also made possible the detection e.g. of the HIV retrovirus in humans causing AIDS. There is also evidence that reverse transcriptase played a major role in the formation of more than one third of the human genome by enabling the replication of retrotransposons, i.e. DNA sequences that use RNA intermediates to amplify themselves in the genome [7]. In plants, the rate of DNA derived from transposable sequences shall be even higher [17].

Non-coding RNAs are also accounted for many functions in (post-)transcriptional regulation of gene expression. In 1993, small non-coding RNAs, called microRNAs, were discovered [18]. With only 22 nucleotides, a microRNA was shown to inhibit the translation of a particular mRNA by being partially complementary to it. In 1998, Fire and Mello managed to manipulate gene expression with RNAs, substantially inhibiting genes in the presence of double stranded RNAs [19]. The inhibition of gene expression by RNAs, which is a post-transcriptional regulation of gene expression, became known as RNA interference (RNAi). These and other discoveries emphasizing the importance in particular of small RNAs in RNAi were announced by the Science Magazine as the *Breakthrough of the Year* in 2002 [20]. In 2006, the work of Fire and Mello rendered them a Nobel Prize. Today, RNAi is at the center of the research of many human diseases including cancer [21], which are commonly related to down or upregulation of genes.

Additional examples of biological processes involving non-coding RNAs are as follows.

- *Alternative splicing*. Non-coding RNAs regulate the removal of introns and connection of exons in the processing of pre-messenger RNAs. This regulation ensures a massive variety of proteins and is considered an important source of complexity in (eukaryotic) organisms [22]. It is also suggested that an organism's complexity correlates with the proportion of non-coding DNA in its genome [23].
- *Chromatin regulation* in eukaryotes. Long non-coding RNAs, normally consisting of more than 200 nucleotides, can mediate protein modifications in the cell nucleus leading to gene silencing [24]. These RNAs can also silence one of the X chromosomes in female cells (of mammals), leaving a single X chromosome to be transcribed in males and females [24].
- *RNA-RNA and RNA-protein interactions*. These interactions, performed e.g. by riboswitches [25], are forms of regulating gene expression. It is also hypothesized that RNA-protein interactions are related to an ancient way by which proteins were directly produced from mRNAs without the need for tRNAs and ribosomes. This idea gives additional support to the *RNA world* hypothesis [26].
- *Immune systems*. RNAs can mediate activation and repression of immune response genes in the antimicrobial defense of a host organism [27] and also regulate gene expression in pathogenic bacteria avoiding detection by the host's immune system [28].

These are only some of the functions of non-coding RNAs. Many more could be listed here and many others continue to be discovered.

1.2 RNA structure and its importance

An RNA molecule consists of a sequence of the nucleotides (or bases) adenine (A), cytosine (C), guanine (G), and uracil (U). Unlike DNA, which usually occurs as a double-stranded molecule and contains thymine instead of uracil, RNA is usually single stranded. On a basic level of organization of an RNA molecule, one observes its *primary structure*, which is a simple specification of the nucleotide sequence composing it. One also observes in RNA that *complementary bases* can form pairs via hydrogen bonds, such as the Watson-Crick pairs A-U and C-G. Other pairings are also possible, such as the wobble pair G-U. Due to these pairings, an RNA molecule can fold into characteristic complex *secondary* and *tertiary structures*. The secondary structure, formed by the set of base pairs occurring in the molecule, can consist of different substructural elements like stem-loops with or without bulges or internal loops as shown in an example in Figure 1.1. The tertiary structure additionally considers specific atomic positions in three-dimensional space [29, 30].

The secondary and tertiary structures are vital for the function of many non-coding RNAs and their interaction with other molecules, with tRNAs and rRNAs being important examples. In all tRNAs, the secondary structure necessary for protein synthesis resembles a characteristic cloverleaf with a

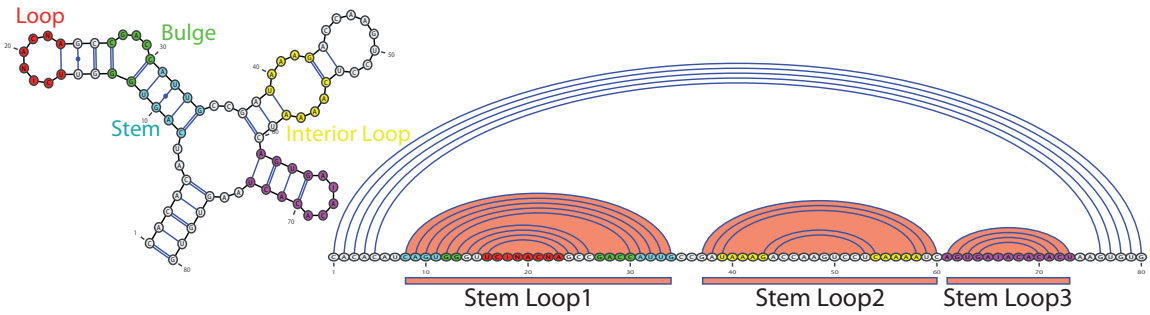


Figure 1.1: Secondary structure elements of an RNA molecule represented by a base-pair graph (left) and as arc-annotated sequence (right). The depicted structure contains three stem-loop substructures.

stem and three stem-loop substructures similarly to the secondary structure shown in Figure 1.1. In this structure, the loop opposite to the stem enables the recognition of triplets of bases from an mRNA to the corresponding amino acid attached to the stem, whereas the other stem-loops further assist in the recognition of the correct amino acid. rRNAs, on the other hand, form along with proteins the structure of the two ribosomal subunits, one binding to an mRNA and the other to tRNA and amino acids.

Most non-coding RNAs with enzymatic activities, either as ribozymes or associated to proteins, also heavily rely on their structure to realize their functions. Ribozymes can use their secondary structure to selectively cleave bases from other RNA molecules. Due to this property, ribozymes have been since recently applied in the treatment of human diseases like AIDS [31]. Associated with proteins, RNAs can also use their primary, secondary, or tertiary structure to act as guides by targeting other RNA molecules or DNA [32]. Examples of such non-coding RNAs are:

- small interfering RNAs (siRNAs), which target mRNAs for degradation;
- small nuclear RNAs (snRNAs), which are involved in the modification of rRNAs; and
- guide RNAs (gRNAs), which catalyse the insertion or deletion of bases U in pre-mRNAs of some protozoan organisms.

The structure is extremely important for the function of a number of other non-coding RNAs (see e.g. [33]).

1.3 The challenge of RNA homology search

Primary and secondary structure conservation among RNAs with similar function is widely acknowledged. Such structure similarities are either inherited from a common ancestor or result from convergent evolution via natural selection. RNAs whose structure similarities classify in the former case are said to be *homologous* and can be grouped into families. For instance, the Rfam database

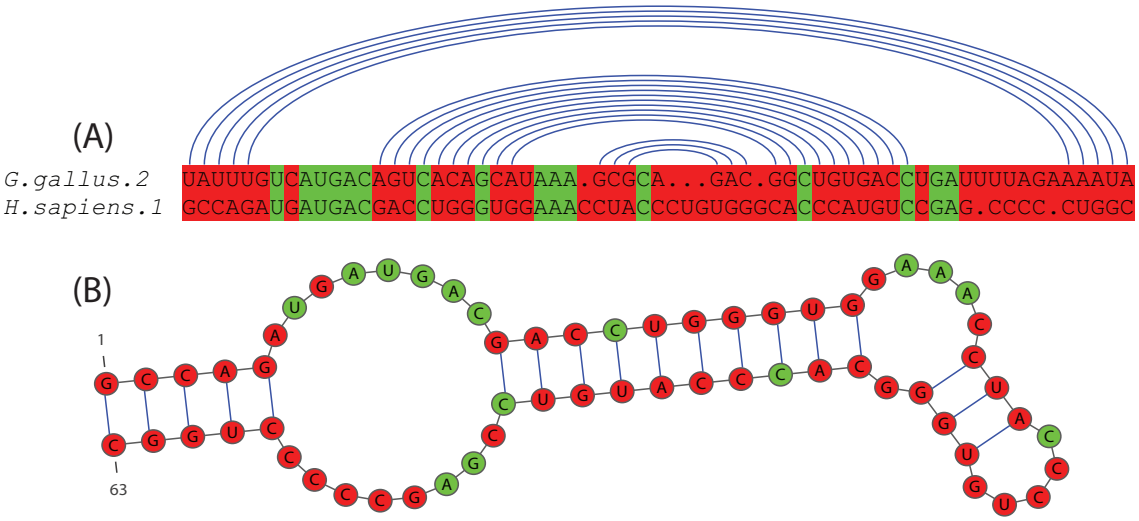


Figure 1.2: (A) Two RNA sequences which, despite differing at the positions marked in red, form the same secondary structure and belong to the same SECIS_1 family (Rfam Acc.: RF00031). (B) Corresponding secondary structure with bases from sequence *H.sapiens.1*.

release 11.0 compiles 2,208 such families [34]. A very important task in bioinformatics is to search sequence databases, e.g. genomes, for occurrences of RNA family members, since this can provide insight about the functions encoded in the searched sequence. This task is called *homology search*.

However, effective RNA homology search is not trivial. Throughout evolution RNAs suffer pressure to retain their function, and consequently also retain primary and secondary structure information, because loss of function usually means an evolutionary disadvantage. Yet, evolutionary pressure on the primary and secondary structure can occur with different intensities. For example, a large number of mutations such as base replacements, deletions, and/or insertions can occur in the sequence, while the RNA may still be able to maintain its secondary structure and function. Even molecules with a relatively low sequence similarity can form similar secondary structures, since the substitution of a paired base can co-vary with the substitution of the other base of the pair, which still allows them to pair according to Watson-Crick and wobble pairing rules (see an example in Figure 1.2). For this reason, primary and secondary structure conservation varies, to different degrees, even among members of well-established RNA families. For example, while in some families of snRNAs like the snRNA Z178 (Rfam Acc.: RF00306) one observes high primary and secondary structure similarity, in others like the U3 family (Rfam Acc.: RF00012) only the secondary structure is highly conserved. Further hampering RNA homology search is the fact that RNAs can vary considerably in length, as observed when comparing micro and long non-coding RNAs. In addition, compared to proteins, the reduced alphabet size reflected by the four nucleotides RNAs can consist of also means reduced sequence information.

Therefore, RNA homology search demands flexible tools making use of both primary and secondary structure information of the sought RNA family. Popular tools based only on sequence comparison like *Blast* [35] and *Fasta* [36], despite providing specific results, are provably not sensitive enough to find members of RNA families folding into characteristic secondary structures but with lower degrees of sequence conservation [37]. Besides ignoring secondary structure information, the heuristic approach of these tools requires exact matching of short fixed-size sequences, which is inappropriate for matching RNAs with frequent insertions and deletions. Combined with the historical focus on the research of proteins, the misuse of these tools unsuitable for the search of RNAs could explain why many non-coding RNAs remained undetected during a long time. Also, other traditional more compute intensive algorithms are based only on sequence, e.g. Smith-Waterman [38] and using HMMs [39]. Hence, newer tools have been developed to enable more sensitive RNA homology searches. Some tools, e.g. *Infernal* [40], *ERPIN* [41], and *RNAMotif* [42], use a model or pattern storing primary and secondary structure information of the sought RNA family. The goal of the model, which can be used to search multiple databases, is to be general enough to represent all members of the family but also be specific to avoid matching false members. Other tools, e.g. *Foldalign* [43] and *LocARNA* [44], directly perform pairwise comparison of RNAs with known or unknown secondary structures trying to identify sequence and structure similarities.

In addition to contributions from new software, much of our today's knowledge about the functions and complexity of the transcriptome (i.e. the set of all coding and non-coding RNAs) in a variety of organisms can be credited to huge advancements in sequencing technologies in the last ten years. For instance, the Human Genome Project initiated in 1990 to completely sequence a human genome for the first time was costly and required thirteen years to complete [45]. Since then, new high-throughput sequencing technologies able to produce millions of sequences in parallel have been transforming genome sequencing into a much cheaper and routine task [46]. These technologies made possible e.g. the complete sequencing of 1,092 human genomes announced in 2012 [47] and put the race to sequence a human genome for a cost of less than 1,000 dollars close to an end [46]. Also, high-throughput sequencing technologies enabled the development of techniques like RNA-Seq [48] and Direct RNA Sequencing [49] for the identification of the whole transcriptome in genomes. These technologies further facilitate the discovery of RNAs and their functions.

While new sequencing technologies can contribute to improving our knowledge about RNAs, the increase in the amount of sequence data they produce by far exceeds the increase in computing capacity for the data analysis. This can be observed in a comparison between the cost of genome sequencing and the Moore's law for computing power [50]. Because the running time of existing tools for RNA homology search considering primary and secondary structure information scales at best linearly in the size of the searched sequences, searching larger and larger sequence databases in plausible time becomes increasingly challenging. Ideally, besides being able to handle primary and secondary structure properties particular to each RNA family, a search tool should have a running time that scales sublinearly in the size of the analyzed sequences.

To accelerate sequence analysis, a well-known approach is to build an index from the target sequences using a full text index data structure like the suffix tree [51], the (enhanced) suffix array [52], or a compressed structure [53]. Once constructed, the index data structure can be used many times to accelerate sequence analysis. This amortizes the time spent in its construction. In the context of biological sequence analysis, enhanced suffix arrays have already successfully been applied in e.g. [54], considerably speeding up database searches using position specific scoring matrices (PSSMs) as query. PSSMs are sequence-based patterns typically used to model short amino acid or nucleotide sequences. For searches with RNA patterns encoding primary and secondary structure information of an RNA family, i.e. *sequence-structure patterns*, however, no practical tool that can exploit an index data structure has yet been developed.

1.4 Thesis structure and contributions

This thesis is concerned with efficient methods for RNA homology search in large sequence databases. Therefore, in the following Chapter, we present existing methods for this task including methods that perform direct comparison of RNAs as well as methods that use a model of an RNA family for the search. We will see that methods following the latter approach, in particular methods for matching RNA sequence-structure patterns, are better suited for searches in a large scale.

In Chapter 3, we present our first novel method for fast matching of RNA sequence-structure patterns. We employ in our method the affix array index data structure, which supports bidirectional pattern search and allows to efficiently handle the structural constraints of the patterns. This leads to an expected running time that is sublinear in the size of the sequence database. To search for complex RNA molecules, we use a new chaining approach which consists in describing the molecule with several patterns and then searching for chains of matches where the order of the patterns is preserved.

To enable even more sensitive and specific searches, we present in Chapter 4 new online and index-based algorithms for approximate matching of RNA sequence-structure patterns. Because this requires to compute semi-global alignments of structural RNA patterns and substrings of the target sequence, we begin with an efficient online algorithm for this purpose that reuses the entries of the required dynamic programming matrices for all substrings. Then, we improve this algorithm by incorporating a technique for avoiding the alignment computation of non-matching substrings and subsequently apply to this algorithm the enhanced suffix array index data structure. We devise two index-based algorithms, both which have a running time that scales sublinearly in the size of the target database. As in our first method, our chaining approach is integrated with all these algorithms. In an extension, we apply general techniques to the algorithms like multithreaded computing for further practical search acceleration.

Because our methods in Chapters 3 and 4 employ very different algorithms and also differ in terms of sensitivity and specificity, each of these chapters present a detailed evaluation of the respective

methods in terms of speed and performance in RNA homology search. Finally, last conclusions and an outlook for future work are given in Chapter 5.

2 Existing RNA homology search methods

2.1 Formal preliminaries

We begin introducing some formal definitions and notations that are used throughout this thesis. Additional definitions will be presented later as needed.

Definition 1 An *RNA alphabet* $\mathcal{A} = \{\text{A, C, G, U}\}$ is a set of characters coding for the bases adenine (A), cytosine (C), guanine (G), and uracil (U).

Definition 2 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 $\varphi(x) \subseteq \mathcal{A}$ [55]. Each character $x \in \mathcal{A}$ can be seen as a character class $\varphi(x) = \{x\}$ of exactly one element.

Definition 3 An *RNA primary structure* or *sequence* S of length $n = |S|$ over $\mathcal{A}$ 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 $S = S[1]S[2]\dots S[n]$ and $1 \leq i \leq j \leq n$, $S[i..j]$ denotes the *substring* $S[i]S[i+1]\dots S[j]$ of S.

Definition 4 Let $S = uv$, 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.

Definition 5 Two bases $(c, d) \in \mathcal{A} \times \mathcal{A}$ are *complementary* if and only if $(c, d) \in \mathcal{C} = \{\text{(A, U), (U, A), (C, G), (G, C), (G, U), (U, G)}\}$. Two complementary bases can form a *base pair*. Less frequently, also non-complementary bases can form pairs.

Definition 6 A *non-crossing RNA secondary structure* $\widehat{R}$ *of length* m is a set of *base pairs* (i, j), $1 \leq i < j \leq m$. Each pair (i, j) stands for the pairing of the base at position i 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 $i < j < i' < j'$

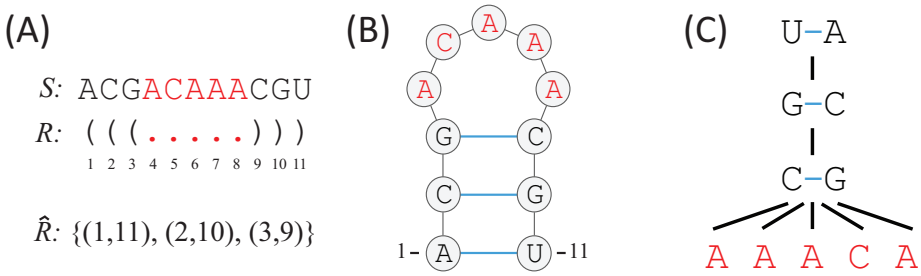


Figure 2.1: (A) Example of an RNA sequence S annotated with a non-crossing RNA secondary structure string R forming a stem-loop. Also shown is the corresponding set of base pairs $\widehat{R}$. (B) The same RNA as a graph and (C) as a tree.

or $i' < j' < i < j$. In the following, we use the single word *structure* to refer to non-crossing RNA secondary structures, unless the structure is explicitly qualified as primary (or tertiary).

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 $(i,j) \in \widehat{R}$.

$\widehat{R}$ is called a *stem-loop* RNA structure if and only if for all $(i,j),(i',j') \in \widehat{R} : i < i' < j' < j$ or $i' < i < j < j'$. See Figure 2.1 for an example of a stem-loop structure in different notations. Stem-loops can also be observed as substructures with bulges and interior loops in Figure 1.1. A stem-loop structure is equivalently called *non-branching*.

2.2 Introduction to existing methods

Given a query RNA sequence of known function or set of homologous RNA sequences belonging to the same family, the goal of homology search methods is to measure similarities or differences between the query and target sequences. High similarity or low difference level can suggest a homologous relationship between the sequences and, therefore, also similar function. Because often sequence information alone is not sufficient to characterize an RNA family, we are interested in methods that make use of both primary and secondary structure information.

2.3 Comparative RNA analysis methods

A recurrent approach for homology search is to directly compare RNAs. Depending on the amount of information available from the primary and secondary structure, which can differ in the query and the target, some methods can simultaneously compare the primary and secondary structures or first focus on comparing either one of them. In case the secondary structure of an RNA to be

compared is not known, a secondary structure can also be computed from its sequence alone or be inferred in combination with other RNAs.

2.3.1 Comparison of RNAs with unknown secondary structure

Commonly, tools for homology search of RNAs with unknown secondary structure make use of methods for the prediction of the secondary structure, i.e. for *RNA folding*. We note that, although folding can also refer to the prediction of the tertiary structure of an RNA molecule, here it is exclusively used to refer to RNA secondary structure prediction. Despite advances of computational methods for tertiary structure prediction, this remains a difficult problem and most mature software for RNA homology search incorporating folding methods focuses on the secondary structure.

The obvious goal of computational methods for RNA folding is to find the structure that best "fits" to the real structure of the RNA in nature. To start with, we are faced with the challenging fact that the number of secondary structures into which an RNA sequence can fold grows considerably (even exponentially, if not only complementary base pairs are allowed) with its length. An algorithm solving this problem is the Nussinov algorithm [56, 39]. Using dynamic programming, it computes the secondary structure with the maximum number of base pairs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space. However, the real structure is influenced by the energetic stability of hydrogen bonds and their effect on stacking (i.e. neighboring) base pairs, loop sizes, and possible different multi loops. Since the Nussinov algorithm does not take these aspects in consideration, the structure for an RNA sequence it computes is in general not biologically relevant. Nevertheless, as one of the first RNA folding algorithms it served as a milestone for the development of new algorithms.

More accurate algorithms compute a secondary structure of a sequence by minimizing its free energy. This is in accordance with the assumption that an energetically stable structure is a structure with minimal free energy (MFE). In this approach, free energies are assigned to substructural elements like stacking base pairs and loops. These free energies have experimentally been determined more precisely over the years, including e.g. the widely used thermodynamic model of Turner [57, 58]. Such a model defines a set of substructural elements with associated free energy parameters. Given free energies, the overall free energy of a structure is calculated as the sum of the free energies of its substructures. A well-known algorithm for MFE computation is the Zuker algorithm, whose dynamic programming recurrences are analogous to those of the Nussinov algorithm [59, 39]. It also runs in $\mathcal{O}\left(n^3\right)$ time if loop sizes are limited by a constant, otherwise it runs in $\mathcal{O}\left(n^4\right)$ time. An implementation of the Zuker algorithm is found in the *mfold* program [60].

A shortcoming of the MFE approach is the fact that the MFE secondary structure is not necessarily the biologically correct one and there can be a huge number of alternative reasonable structures whose free energies differ only modestly. This suggests to introduce a probability distribution over alternative structures. A dynamic programming algorithm for computing the partition function of an RNA sequence over all its alternative secondary structures is given by McCaskill [61]. This

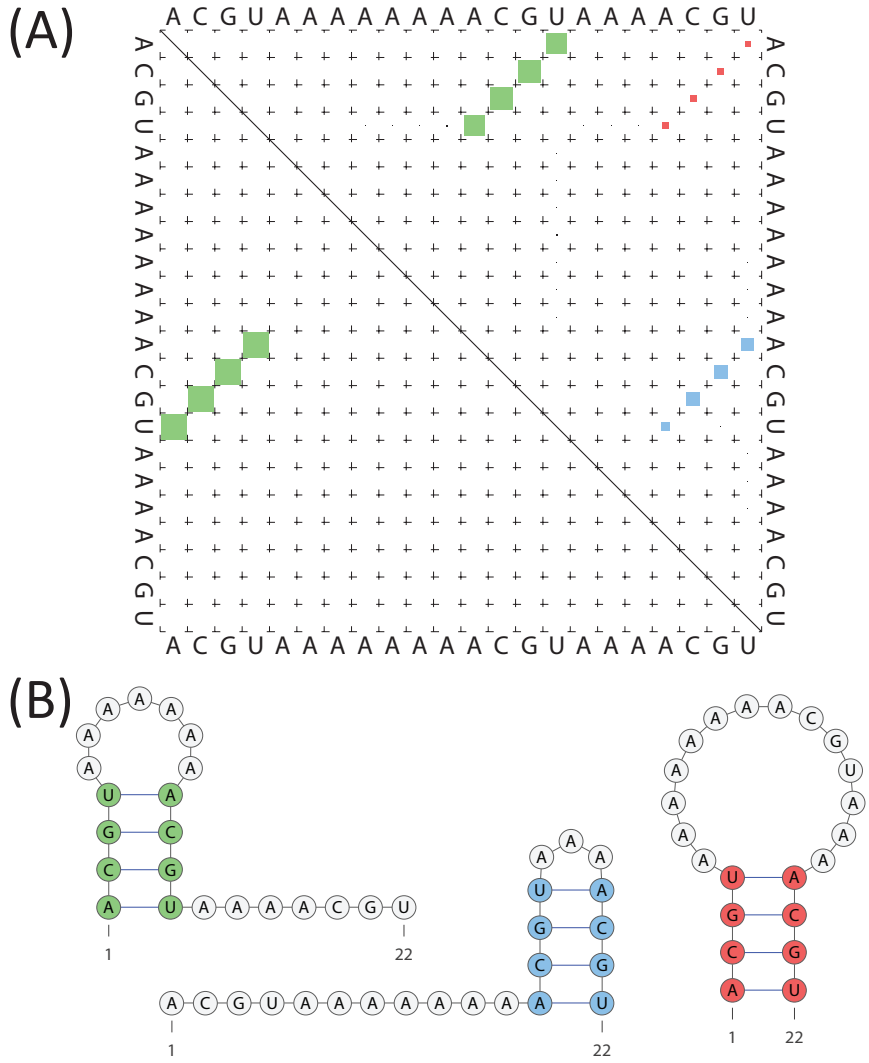


Figure 2.2: (A) Base pairing probability matrix for sequence S = ACGUAAAAAAAACGUAAA ACGU shown repeatedly on the matrix edges. In the upper right triangle, each square denotes the probability of a base pair for the entire ensemble of possible secondary structures. The area of a square is directly proportional to the probability of the corresponding base pair. The lower left triangle shows only squares for base pairs that form the single secondary structure of minimum free energy. (B) Three possible secondary structures derived from the base pair probabilities in the matrix. The color of the base pairs matches the color of the corresponding square in the matrix.

algorithm, which runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space, computes for a given RNA sequence a base pairing probability matrix; see an example in Figure 2.2. This matrix allows to explore the space of possible secondary structures by a derivation of the structures from probable base pairs. Hence, it gives a broader overview of feasible structures instead of a single MFE structure. The algorithm of McCaskill is implemented in the program *RNAfold* of the *ViennaRNA Package* [62].

2.3.2 The three plans of comparative RNA analysis

The three algorithms from Nussinov, Zuker, and McCaskill work in their original form on a single RNA sequence. However, since RNA molecules with similar function tend to form similar structures, the comparison of *several* putative homologous sequences for the prediction of a *consensus secondary structure* common to all these sequences is often a more reliable approach than folding of a single sequence. Even molecules with a relatively low sequence similarity can form similar structures due to co-varying substitutions of bases forming pairs (see an example in Figure 1.2). That is, mutations on the sequence level do not necessarily destroy base pairings. In [63], existing methods for comparative RNA analysis are classified into the following three approaches or *plans*.

1. The sequences are first aligned and then a consensus secondary structure is inferred from the resulting multiple sequence alignment. For an example of a multiple sequence alignment annotated with a consensus secondary structure string R, see Figure 2.3.
2. The sequences are aligned while simultaneously inferring a consensus secondary structure.
3. The sequences are first individually folded and then a structure alignment is computed.

Plan 1 seems appropriate when the sequence conservation is sufficiently high, so that the "correct" bases are aligned in a certain column. However, when the sequence conservation is too low, base shifts in the alignment can misalign base pairs and prevent the formation of consensus base pairs in the folding phase, consequently leading to a suboptimal consensus secondary structure and corrupting the homology analysis. Plan 3, on the other hand, seems appropriate when sequence conservation is too low for the computation of a meaningful alignment based on sequence information. Its disadvantage lies in the fact that individual folding of sequences on a first step by computing e.g. minimum free energies can lead to very diverged secondary structures which can hardly be aligned. Furthermore, any sequence similarity among sequences is completely ignored in the folding phase. The best theoretical solution is to use plan 2, i.e. the Sankoff algorithm [64], which simultaneously computes an optimal multiple alignment and consensus secondary structure by combining recurrences of a standard dynamic programming sequence alignment algorithm and the Nussinov algorithm. However, for m sequences of length n, its high complexity of $\mathcal{O}\left(n^{3m}\right)$ time and $\mathcal{O}\left(n^{2m}\right)$ space makes it of no practical use. Therefore, variants of the Sankoff algorithm with reduced time complexity have been introduced.

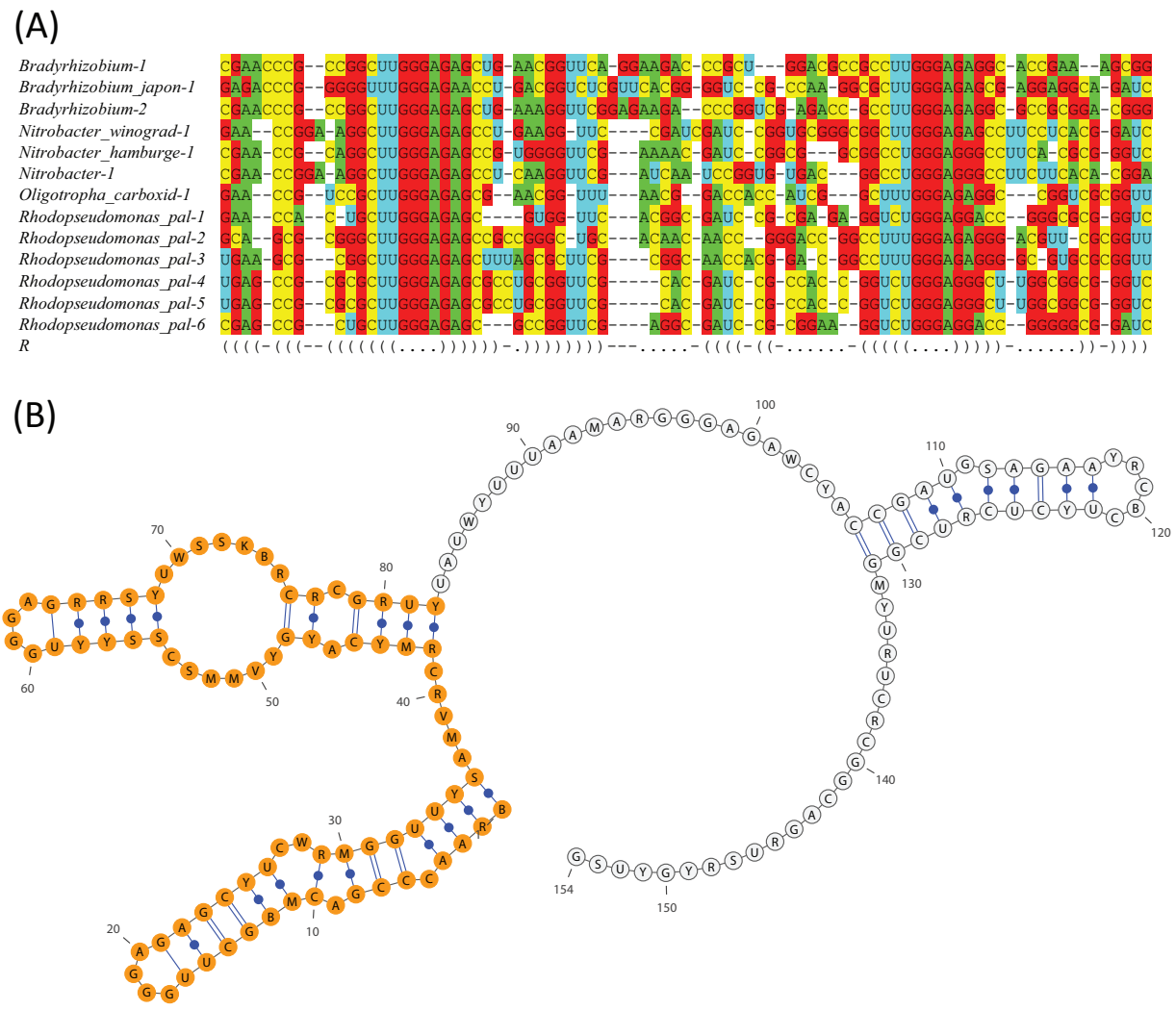


Figure 2.3: (A) Multiple alignment of sequence members of family flg-Rhizobiales RNA (Rfam Acc.: RF01736) annotated with a consensus secondary structure string R. Observe that the alignment is truncated to contain only the columns corresponding to the first two stem-loops of the secondary structure of this family in 5' to 3' direction. (B) Complete secondary structure of this RNA family with the first two stem-loops drawn in orange. In this secondary structure, each ambiguous IUPAC symbol $x \in \Phi$ stands for a character class $\varphi(x) \subseteq \mathcal{A}$.

2.3.3 Faster simultaneous RNA alignment and folding: Sankoff variants

Offering some relief from the high time complexity of the Sankoff algorithm, the program *Foldalign* [65] provides a restricted version of it for the computation of pairwise local or global sequence-structure alignments which does not allow for branching structures. In this way, *Foldalign* achieves a time complexity of $\mathcal{O}\left(n^4\right)$, instead of $\mathcal{O}\left(n^6\right)$ for the Sankoff algorithm. The restriction to non-branching structures has been eliminated in a second version of the program [43], which accelerates the computation by pruning the used dynamic programming matrices. However, the program does not guarantee to find an optimal solution. Another program, *Dynalign* [66], simplifies the computation in the Sankoff algorithm by limiting the distance of aligned bases in two input sequences. That is, for positions k and l in each of the two sequences, $|k-l| \leq M$ must hold, where M is a constant. Such a distance is called the *span* between the two positions. *Dynalign* limits, in addition, the size of loops to achieve a time complexity of $\mathcal{O}\left(n^3M^3\right)$.

Two other variants of the Sankoff algorithm, *PMcomp* [67] and its successor *LocARNA* [44], use a different approach to reduce computational demands. For a pairwise alignment, they use precomputed secondary structure information in the form of base pairing probabilities from each individual sequence, which can be obtained using McCaskill's algorithm. In practice, they take as input postscript files of base pairing probabilities generated for each sequence with program *RNAfold*. In *LocARNA* and *PMcomp*, by transforming these base pairing probabilities into scores and also assigning scores to sequence operations (i.e. (mis)matches, insertions, and deletions), the two input sequences are simultaneously aligned and folded via the computation of an alignment that maximizes the combined scores from the sequence operations and the base pairings forming a consensus secondary structure. More precisely, consider two sequences S and T to be aligned using given base pairing probability matrices P^X, $X \in \{S,T\}$, of dimensions $|X| \times |X|$. Let $\widehat{R}^X$ denote the secondary structure of sequence X. Here, an alignment of S and T consists of

- a set A of alignment edges (i,k) between positions i of S and positions k of T and
- a consensus secondary structure ζ of S and T, which is a set of pairs of base pairs $((i,j),(k,l))$, with $(i,j) \in \widehat{R}^S$ and $(k,l) \in \widehat{R}^T$. Additionally, (i,k) and (j,l) must be alignment edges in A.

To speed up the computation of an alignment, an improvement of *LocARNA* over *PMcomp* consists in eliminating base pairs (i,j) from $\widehat{R}^X$ with very low probability P^X_{ij}. That is, given a probability cutoff p_{min}, if $P^X_{ij} < p_{min}$, then $(i,j) \notin \widehat{R}^X$. This reduces the number of base pairs that are scored and considered in the alignment computation. Both tools compute log-odds scores for base pairs. In *LocARNA*, the score of a base pair $(i,j) \in \widehat{R}^X$ is computed as

$$score^X(i,j) = \log \frac{P^X_{ij}}{p_0} / \log \frac{1}{p_0}. \tag{2.1}$$

Term $\log \frac{1}{p_0}$ normalizes the score so that it does not exceed 1, where p_0 is the expected probability for a pair to occur at random. For $(i,j) \notin \widehat{R}^X$, $score^X(i,j) = -\infty$. To score alignments on the se-

quence level, functions $\sigma(S[i], T[k])$ and $\tau(S[i], S[j]; T[k], T[l])$ give the score for the substitution of unpaired and paired bases, respectively. Let γ be a gap penalty and N be the number of gaps in an alignment. The score of an alignment specified by the pair (A, ζ) is

$$\sum_{((i,j),(k,l))\in\zeta} (score^S(i,j) + score^T(k,l) + \tau(S[i], S[j]; T[k], T[l])) + \gamma N + \sum_{((i,j),(k,l)),((j,i),(k,l))\notin\zeta} \sigma(i,k). \tag{2.2}$$

This score is maximized by *PMcomp* and *LocARNA* using dynamic programming. Similarly to *Dynalign*, *PMcomp* limits the size of loops and the span between aligned unpaired bases and base pairs in the two sequences. For aligned base pairs (i, j) and (k, l), the span is computed as $|(j - i) - (l - k)|$. Using these limitations, *PMcomp* achieves time and space complexities of $\mathcal{O}\left(n^4\right)$ and $\mathcal{O}\left(n^3\right)$, respectively. *LocARNA* also uses this span limitation technique, but profits in addition from a sparse computation of the dynamic programming matrices allowed by the reduced number of base pairs, which are prefiltered according to variable p_{min} as described above. In this way, *LocARNA* requires $\mathcal{O}\left(n^2 + m^2\right)$ time and $\mathcal{O}\left(n^2(n^2 + m^2)\right)$ space, where n and m are the lengths of the aligned sequences. While both tools can compute global alignments, *LocARNA* is also tailored for computing local alignments. For this, it forbids negative entries in the computed matrices and uses a suitable traceback technique similar to the technique used for computing local alignments of plain sequences (see Smith-Waterman algorithm [38]).

PMcomp and *LocARNA*, as well as the previously mentioned tools for comparative RNA analysis, are suitable for comparing sequences of similar lengths by either computing global or local alignments. However, families of homologous RNAs are commonly characterized by short structural motifs, which we often want to search for in large sequences such as genomes. That is, one of the sequences to be aligned is much shorter than the other. In this case we want to compute semi-global alignments by aligning the complete shorter "query" sequence to substrings of the larger sequence. For this purpose, a variant of *LocARNA* called *LocARNAscan* [68] slides a window over a large target sequence applying the *LocARNA* method, with the difference that it aligns the complete query sequence to each window substring. Because the query can represent an RNA family of sequences, *LocARNAscan* allows to incorporate, in the query, information from a multiple sequence alignment. This is done by adjusting the σ and τ functions above to provide log-odds scores computed from a multiple sequence alignment. Let a and b be bases from alphabet $\mathcal{A}$. In addition, let $f_{i,a}$ denote the frequency of a in column i of the multiple alignment and $f_{ij,ab}$ denote the frequency of base pair (a, b) in columns i and j. Functions σ and τ are computed as

$$\sigma(a,b) = \log\left(\frac{f_{i,a}}{b_a}\right) \quad \text{and} \quad \tau(ij,ab) = \log\left(\frac{f_{ij,ab}}{b_{ab}}\right) \tag{2.3}$$

where $b_a = 1/4$ and $b_{ab} = 1/6$ represent a uniform distribution of the background frequencies of the 4 possible bases a and 6 possible complementary base pairs $(a, b) \in \mathcal{C}$, respectively. In terms of time and space complexity, *LocARNA*'s running time and space requirements scale at least quadrati-

cally with the length of both input sequences. This forbids the direct application of its dynamic programming recurrences when one of the sequences is very large. Therefore, *LocARNAscan* adapts the recurrences so that the dynamic programming tables only store entries for the alignment computation for the current window. In addition, when shifting a window it reuses computed entries from overlapping windows. For a window of length m and a target sequence of length n, it achieves time and space complexities of $\mathcal{O}\left(L^2nm\right)$ and $\mathcal{O}\left(Lm\right)$, respectively, where L is the maximal allowed span between aligned base pairs in the two sequences. Furthermore, note that *LocARNAscan* also requires a precomputation of base pairing probabilities. However, it cannot use program *RNAfold* for this purpose since for a genome of length n it creates a huge matrix of size $|n| \times |n|$. Therefore, it uses instead a similar program also available in the *ViennaRNA Package* called *RNAplfold*, which computes base pairing probabilities for windows of length m. This computation takes $\mathcal{O}\left(m^2n\right)$ time.

2.3.4 Comparison of RNAs with known secondary structure

Although programs *LocARNA* and *PMcomp* are in principle designed for the comparison of RNAs with unknown secondary structure, known secondary structure information can be provided to these programs via the precomputation of constrained base pairing probability matrices. A constraint can be the requirement of a pair of bases to pair, realized by assigning probability 1 to the specific pairing. Another constraint can be the requirement of a base to pair with any other base or that a specific base be unpaired. In the latter case, the pairing probability between the specific base and every other base is 0. The computation of constrained base pairing probability matrices is supported by program *RNAfold*.

Other tools strictly require known secondary structure of the RNAs to be compared. For example, *MARNA* [69] computes a multiple alignment of a set of RNAs with known structure. It works in two steps. First, it computes all pairwise sequence-structure alignments based on the dynamic programming algorithm of Jiang *et al.* [70]. Each pairwise alignment computation takes $\mathcal{O}\left(m^2n^2\right)$ time, where m and n are the lengths of the aligned sequences. Secondly, it uses the obtained alignments to weigh edges in the multiple sequence alignment tool *T-Coffee* [71]. Note, however, that this step does not compute a true alignment of primary *and* secondary structures, since *T-Coffee* ignores the dependency between base pairs.

Another tool, *RNAforester* [72], computes a pairwise alignment of secondary structures, which is suitable for aligning RNAs with very little sequence similarity. In *RNAforester*, secondary structures are represented as trees in which internal nodes stand for base pairs and leaves stand for single bases (see an example in Figure 2.1 (C)). An alignment of two trees can be seen as a generalization of a standard sequence alignment. That is, the alignment is represented as another tree, whose nodes are equivalent to alignment edges labeled with either a pair of nodes, one from each tree, or a node from one tree and a gap symbol. *RNAforester* can compute global or local alignments using dynamic programming in $\mathcal{O}\left(mnd^2\right)$ time, where m and n are the number of nodes in each tree and

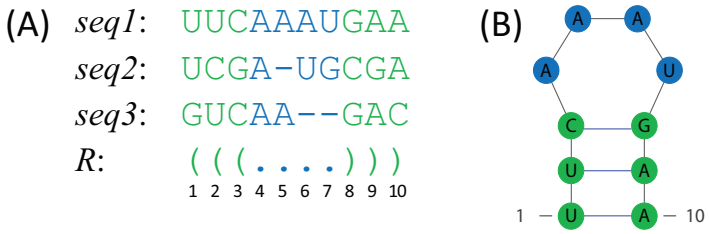


Figure 2.4: (A) Example multiple sequence alignment annotated with a consensus secondary structure string R. Paired (unpaired) positions in green (blue) describe one substructural element for which *ERPIN* constructs a profile. (B) Corresponding consensus secondary structure with bases from sequence *seq1* highlighting its substructures with the same colors as in the alignment.

d is the maximum degree, i.e. number of outgoing edges from a node, observed in the trees. The degree is at least one, since a base pair node must always be connected to another base pair node or leaf node.

MARNA and *RNAforester* are suitable for integration in a pipeline of comparative RNA analysis. More precisely, they can be used in the second step of *plan* 3 described above, taking as input RNA sequences previously folded using e.g. program *RNAfold* or *mfold*. However, they can suffer from the relatively poor quality of folding of single RNAs. In addition, we remark that, by only supporting global or local alignment computations requiring times that scale quadratically in the length of the sequences, these tools are not suitable for homology searches in large sequence databases.

2.4 Secondary structure profiles: *ERPIN*

ERPIN (*Easy RNA Profile IdentificatioN*) [73, 41] is a tool that takes as input an RNA multiple sequence alignment annotated with a consensus secondary structure and builds a statistical model, which it then uses to search sequence databases for matches of the model. The built model, called *secondary structure profile* (SSP), is the combination of profiles for paired and unpaired substructural elements of the input alignment. In this context, a substructural element is a stretch of continuous unpaired positions of the alignment or continuous base paired positions belonging to the same helical element, e.g. the stem of a stem-loop. For an example of a structure-annotated multiple sequence alignment and involved substructural elements, see Figure 2.4. An SSP can be composed of profiles of two types defined as follows.

Definition 7 Let m be the length of an unpaired substructure, e.g. the loop of a stem-loop. A *single-strand profile* of length m is a two-dimensional matrix of size $5 \times m$ modeling an unpaired substructural element. Each column in this matrix corresponds to one unpaired column of the input alignment and each row corresponds to a possible base from $\mathcal{A}$ with the addition of one row

representing a gap, which is treated like a base. An entry of the matrix is a log-odds score of the respective base or gap in the corresponding alignment column.

The log-odds scores making up a single-strand profile are computed in two steps. First, a frequency profile, which is a matrix with the same dimensions as the single-strand profile, is computed from the input alignment. Let N_i be the number of bases in unpaired column i of the alignment and $n_{i,a}$ be the number of occurrences of a specific base $a \in \mathcal{A}$ in this column. An entry of the frequency profile is computed as

$$P_{i,a} = \frac{n_{i,a}}{N_i}. \tag{2.4}$$

In the second step, the frequency profile is used to compute the log-odds scores of the single-strand profile. Let β_a be the background frequencies of unpaired base a in the sequence to be searched. An entry of the single-strand profile is computed as

$$score_{i,a} = \log\left(\frac{P_{i,a}}{\beta_a}\right). \tag{2.5}$$

Scores for gaps are calculated via simulations with profiles built from random sequences with the same composition as the target sequence. For details, see [41].

The profile for paired substructural elements of the input alignment is defined as follows.

Definition 8 Let p be the number of base pairs in a substructural element. A *helix profile* of length p is a two-dimensional matrix of size $16 \times p$ modeling a base-paired substructural element. Each column in this matrix corresponds to two base-paired columns of the input alignment and each row corresponds to a possible base pair from $\mathcal{A} \times \mathcal{A}$. An entry of the matrix is a log-odds score of the respective base pair in the corresponding alignment column.

Notably, a helix profile cannot model gaps. Consequently, columns of the input alignment corresponding to base-paired positions must be ungapped. Like in the computation of a single-strand profile, the log-odds scores making up a helix profile are computed from a corresponding frequency profile, i.e. a matrix with the same dimension as the helix profile. Let N_{ij}, $i < j$, be the number of base pairs in paired columns i and j of the alignment and $n_{ij,ab}$ be the number of occurrences of a specific base pair $(a, b) \in \mathcal{A} \times \mathcal{A}$ in these columns. An entry of the frequency profile for a helix profile is computed as

$$P_{ij,ab} = \frac{n_{ij,ab}}{N_{ij}}. \tag{2.6}$$

Let β_{ab} be the background frequencies of base pair (a, b) in the sequence to be searched. Using the frequency profile, an entry of the helix profile is computed as

$$score_{ij,ab} = \log\left(\frac{P_{ij,ab}}{\beta_{ab}}\right). \tag{2.7}$$

The SSP of a structure-annotated RNA alignment can be composed of a number of helix and single-strand profiles. For example, for the alignment in Figure 2.4 annotated with a simple stem-loop, the

(A)
```
>structure
(((----)))
>seq1
UUCA-AUGAA
>seq2
UCGAAUGCGA
>seq3
GUCAA--GAC
```

(B)

	1	2	3
(A,A)			
(A,C)			
(A,G)			
(A,U)			
(C,A)			
(C,C)			
(C,G)			
(C,U)			
(G,A)			
(G,C)			
(G,G)			
(G,U)			
(U,A)			
(U,C)			
(U,G)			
(U,U)			

(C)

	1	2	3	4
A				
C				
G				
U				
-				

Figure 2.5: (A) Multiple alignment of the sequences shown in Figure 2.4 in a FASTA-like format for input in *ERPIN*. In this format, the entry at the top describes the consensus secondary structure using brackets and hyphens at paired and unpaired positions, respectively. The entries following it are the sequences of the alignment with gaps. (B) and (C) show the matrices of the helix and single-strand profiles for the substructural elements in green and blue of the given alignment, respectively. Because the entries in these profiles are log-odds scores computed from both the input multiple sequence alignment and the target sequence, concrete scores are not shown. For more details, see main text.

SSP consists of one helix profile and one single-strand profile. The required syntax of the input alignment resembles the FASTA format, with the first entry being a consensus secondary structure string similar to a structure string R, but with hyphens (-) in the place of dots. The other entries are the globally aligned sequences. See in Figure 2.5 (A) the alignment of Figure 2.4 in this format and in Figures 2.5 (B) and (C) the respective SSP.

Given an SSP and a target sequence S, the search for occurrences of the SSP in S is performed by scoring substrings of S. Scored substrings can contain deletions at unpaired positions just like the sequences in the alignment used to build the SSP. Therefore, associated to each single-strand profile of the SSP is a number of allowed deletions for the respective substructural element. For a single-strand profile modeling unpaired alignment columns in the range from columns i until j, the associated number of allowed deletions is the maximum number of gaps observed in an aligned sequence in the same range of columns from i until j. No insertions are allowed in scored substrings. As an example, the number of allowed deletions for the single-strand profile in Figure 2.5 (C) is 2, because in the respective alignment in Figure 2.5 (A) the loop substructure in sequence *seq3* corresponding to alignment columns 4 until 7 contains 2 gaps (see in Figure 2.4 (A) the same alignment with numbered columns).

Let m be the summed number of columns of each helix and single-strand profile constituting an SSP and d be the total number of allowed deletions in substrings S' of S to be scored. That is,

d is the sum of the deletions allowed per single-strand profile. To score these substrings with the SSP, an algorithm slides along S a window of length m, such that the scored substrings S' have length between m and $m - d$ and begin at the first position covered by the window. The scoring computation is performed by aligning all profiles constituting the SSP to each substring S', so that each single or pair of positions in the substring corresponds to a column of a profile. This means that helix profiles are aligned simultaneously to the 5' and 3' ends of a base paired substructural element. Since helix profiles do not allow for gaps, aligning them to the substring is straightforward, whereas single-strand profiles are aligned using dynamic programming. The used dynamic programming matrix for aligning a particular single-strand profile has size $m \times m$, where m is the length of the profile. In this matrix, one dimension corresponds to the substring of length m of S to be aligned and the other dimension corresponds to the profile. The score of an entire substring S' covered by a window is the sum of the entries of the corresponding base or base pair in each aligned profile. For each window shift, only the substring S' with the highest score among all substrings of length between m and $m - d$ is a candidate match of the SSP.

For the example SSP in Figures 2.5 (B) and (C) modeling a stem-loop, the scoring of substrings covered by a window is performed systematically as follows. Observe that the helix and single-strand profiles contain 3 and 4 columns, respectively, and that the single-strand profile allows for 2 deletions. Therefore, the length of scored substrings can vary between 8 and 10. Given a window of length $m = 10$ beginning at position i of the target sequence S, the algorithm places the helix profile at the first three positions of the current window corresponding to the 5' end of the stem substructure. Then, it aligns the single-strand profile to substrings $S'[4..5]$, $S'[4..6]$, and $S'[4..7]$ obtaining a score for each. Note that, for all these substrings, the alignment requires to compute only one dynamic programming matrix of size 4×4, since extending the alignment to the right only requires to compute an additional column and row of the matrix. For each of the substrings, the algorithm also computes a score from the helix profile by placing it at the possible 3' end positions $S'[6..8]$, $S'[7..9]$, and $S'[8..10]$ of the stem, respectively. The combined score from both profiles is the score of the respective substring of length between 8 and 10. Only the substring with the highest score is a candidate match of the SSP at position i of S. Such a scoring algorithm performs $\mathcal{O}(n)$ window shifts for scanning a sequence of length n. Since in the worst case an SSP can consist of only one single-strand profile, which is aligned to target substrings using dynamic programming in $\mathcal{O}(m^2)$ time, the scoring algorithm requires $\mathcal{O}(nm^2)$ time. To improve its practical running time, *ERPIN* can assign score cutoffs to the profiles constituting an SSP. Using these cutoffs, a certain profile is only aligned to a target substring if the score obtained from the previously aligned profiles exceeds a cutoff.

ERPIN allows to use pseudocounts to avoid overfitting of the SSP by employing two position independent substitution matrices M^1 and M^2 similar to RIBOSUM in the *RSEARCH* program [74]. However, they are computed differently from RIBOSUM. In these matrices, an entry M^1_{cd} contains a value for substituting base $c \in \mathcal{A}$ by base $d \in \mathcal{A}$ and M^2_{xy} contains a value for substituting base pair $x \in \mathcal{A}^2$ by base pair $y \in \mathcal{A}^2$. Hence, M^1 has 4×4 entries and M^2 has 16×16 entries.

More precisely, the value stored in M^1_{cd} is the sum of the scalar product of the base counts observed in a large "training" alignment, i.e. $\sum_{i=1}^{w} Q_{i,c}Q_{i,d}$, where w is the number of columns of the concatenated profiles of the SSP and $Q_{i,c}$ is the number of bases c in one column of the training alignment modeled by column i of the concatenated profiles. The entries are normalized so that for all d, $\sum M^1_{cd} = 1$. M^2_{xy} is analogously computed for base pairs x and y. With these computed matrices M^1 and M^2, *ERPIN* then scores target sequences with a reformulated frequency profile for each helix and single-strand profile. The frequency profile for an unpaired position i of the input alignment becomes

$$P'_{i,c} = \sum_{d \in \mathcal{A}} M^1_{cd} P_{i,d}. \tag{2.8}$$

The frequency profiles for paired positions are reformulated analogously.

2.5 Covariance models

Given a multiple sequence alignment of related RNAs (i.e. an RNA family) annotated with a consensus secondary structure, a covariance model (CM) of the alignment can be used for searching sequence databases for homologous RNAs and computing multiple sequence-structure alignments. CMs extend the concept of profile hidden Markov models (pHMMs) [75, 39], which are very prominent in the field of protein homology search [76, 77]. CMs, like pHMMs, contain position specific information about the conservation of the columns of the multiple sequence alignment. However, they are more complex than pHMMs, capturing not only primary sequence but also secondary structure information of the respective RNA family. This is achieved by treating base paired positions of the sequence alignment as dependent units, in contrast to pHMMs where each position of the alignment is treated independently.

Covariance models are the formulation of profile stochastic context-free grammars (SCFGs) to model RNAs introduced by Eddy and Durbin [78, 39]. Therefore, to understand covariance models, it is important to understand the underlying concept of grammars, which is reviewed next. We remark that SCFGs for RNA analysis were independently introduced by Sakakibara et al. [79, 80].

Foundations of context-free grammars

Natural and computer languages present regularities, which are formalized and studied by generative grammars. These grammars have rules that define not only how strings are generated, but also allow to determine whether existing strings could have been generated by a specific grammar. These properties led to a wide application of generative grammars for the analysis of biological sequences [39]. We define a generative grammar as follows.

Definition 9 A generative grammar is a tuple $(N, \Sigma, P, \mathsf{S})$ where:

- N is a finite set of abstract *nonterminal symbols* not appearing in strings generated by the grammar. By convention, these symbols are uppercase.
- Σ is a finite set of *terminal symbols*, consisting e.g. of characters denoting a base from alphabet $\mathcal{A}$. By convention, these symbols are lowercase.
- P is a finite set of *production rules* $(\Sigma \cup N)^* N^+ (\Sigma \cup N)^* \to (\Sigma \cup N)^*$, where $(\Sigma \cup N)^*$ denotes zero or more occurrences of $\Sigma \cup N$ and N^+ denotes at least one occurrence of a nonterminal symbol of N. One says that the left-hand side of the rule *produces*, *generates*, or *emits* the right-hand side.
- $\mathsf{S} \in N$ is a *start symbol* allowing for the application of production rules from P.

Terminal and nonterminal symbols are also referred to as simply *terminals* and *nonterminals*.

Generative grammars of particular interest for the analysis of RNA sequences are *context-free grammars* (CFGs), which are formalized by Chomsky [81, 82] as follows.

Definition 10 A context-free grammar is a generative grammar $(N, \Sigma, P, \mathsf{S})$, such that each production rule of P satisfies $\alpha \to \beta$, where $\alpha \in N$ and $\beta \in (\Sigma \cup N)^*$. That is, the left-hand side of the production rule must consist of one nonterminal only.

As an example, a CFG for generating strings with an equal number of 0s and 1s is ($\{\mathsf{S}\}$, $\{0, 1\}$, $\mathsf{S} \to 0\mathsf{S}1\mathsf{S} \mid 1\mathsf{S}0\mathsf{S} \mid \varepsilon$, S), where '|' means *or*. To generate a string from this grammar, we apply, beginning with nonterminal start symbol S, successive production rules from P, replacing step by step the left-hand side of the rule with the right-hand side. This can be repeated until the generated string contains only terminal symbols. The successive application of production rules that transforms the start symbol S into a string, replacing at each step the left-hand side of the rule with the right-hand side, is called a *derivation* of the string from the grammar. A derivation of an example string 00011101 generated by the above grammar can be represented as

$$\mathsf{S} \Rightarrow 0\mathsf{S}1\mathsf{S} \Rightarrow 00\mathsf{S}1\mathsf{S}10\mathsf{S}1\mathsf{S} \Rightarrow 000\mathsf{S}1\mathsf{S}1101 \Rightarrow 00011101.$$

To determine whether a given string could have been generated by a given CFG, we build a derivation of the string or show that no such derivation exists. The latter means that the string cannot be generated with the grammar. To build a derivation, we begin at its right-hand side and apply production rules backwards until we obtain only the start symbol S. The process of building a derivation for a given sequence is called *parsing*, while a sequence of production rules generating the sequence in question is called a *parse* of the sequence. If the CFG allows to build more than one parse for the same sequence, the CFG is *ambiguous*.

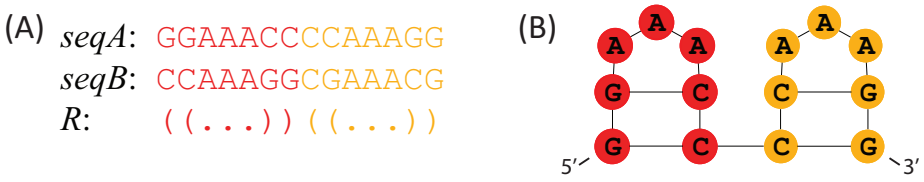


Figure 2.6: (A) Two example RNA sequences with a consensus secondary structure string R. (B) Drawing of the respective branching secondary structure with bases from sequence *seqA*. In both (A) and (B), the first (second) stem-loop substructure in 5' to 3' direction is highlighted in red (orange).

Using context-free grammars to model RNA primary and secondary structure

A CFG can be used to model both the primary sequence and the secondary structure of RNAs. This is accomplished with different nonterminals and production rules, including a nonterminal for the simultaneous generation of two terminals corresponding to a base pair and nonterminals for the generation of unpaired bases. For example, consider sequences *seqA* and *seqB* given in Figure 2.6 (A) along with a consensus secondary structure string R. To define a CFG generating these sequences and respective consensus secondary structure, we use nonterminals P for an emission of a *base pair* $(a, b) \in \mathcal{A} \times \mathcal{A}$ and production rule $\mathsf{P} \rightarrow aWb$, where W is any nonterminal symbol from the set of nonterminals N. For an emission of a single base on the *left* of a nonterminal, we use nonterminals L and production rule $\mathsf{L} \rightarrow aW$. Observe that the consensus secondary structure branches into two stem-loops (see Figure 2.6 (B)). Therefore, we also use a nonterminal B denoting *bifurcation* (i.e. *branching*) allowing an emission of two nonterminals with a single production rule. To denote the *start* and *end* of the structure and structural elements like stem-loops, we use nonterminals S and E, respectively. We can now define the complete set of nonterminals as $N = \{\mathsf{S}, \mathsf{L}, \mathsf{P}, \mathsf{B}, \mathsf{E}\}$, the set of terminals as $\Sigma = \{a, b\}$, and the set of production rules as $P = \bigcup_{W \in N} \{\mathsf{S} \rightarrow W, \mathsf{L} \rightarrow aW, \mathsf{P} \rightarrow aWb\} \cup \{\mathsf{B} \rightarrow \mathsf{SS}, \mathsf{E} \rightarrow \varepsilon\}$. Using this CFG, we give in Figure 2.7 (A) the parses generating *seqA* and *seqB* with bases assigned to terminals a and b. Note that a parse of a sequence and structure can be represented in the form of a *parse tree*, as shown in Figure 2.7 (B) for *seqA*. By traversing the tree top down, we can obtain *seqA*. Note also that, given only the tree, we can precisely obtain the structure string R of *seqA* by looking at the topology of the tree and observing that each base beside nodes with nonterminal L is unpaired and that bases beside nodes with nonterminal P are paired. This representation of a sequence-structure parse as a parse tree will be useful for defining CMs.

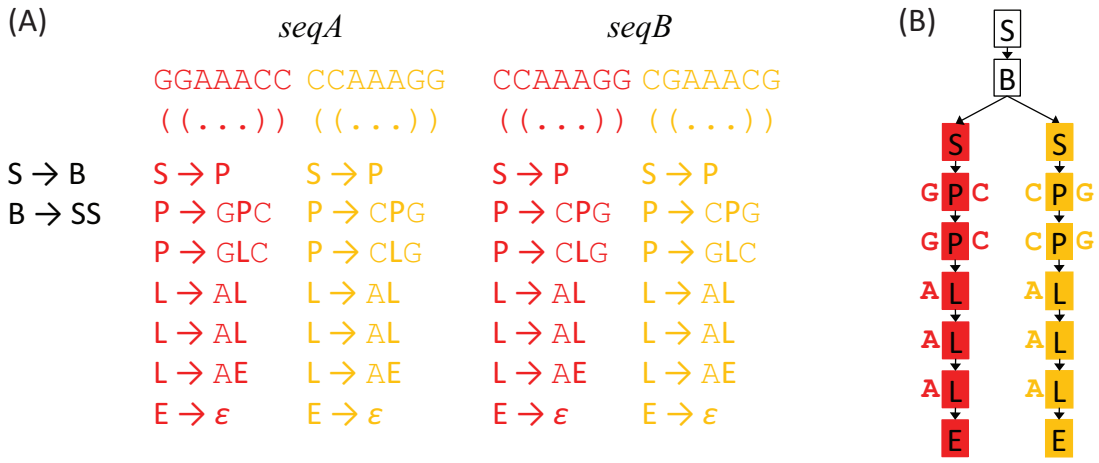


Figure 2.7: (A). Parses of sequences *seqA* and *seqB* and respective secondary structure for the CFG given in the main text with bases assigned to terminal symbols. The first two production rules on the left are common to the parses of both sequences. (B) Parse tree of sequence *seqA* and its secondary structure. Colors correspond to the respective stem-loop substructures.

From stochastic context-free grammars to covariance models

Commonly, more than one parse can generate the same sequence and structure of an RNA using a given CFG. That is, the CFG is ambiguous. Such a parse, as described until here, cannot be preferred over another due to the absence of a measure of quality or scoring. However, for the analysis of RNA sequences, we are not simply interested in determining whether a sequence can be parsed by the grammar. We need a grammar allowing to model the primary and secondary structure of an RNA family provided as a multiple alignment of its members, which can be used to parse and score a target sequence. This is possible with *stochastic context-free grammars* (SCFGs), the probabilistic variants of (CFGs), defined as follows.

Definition 11 A *stochastic context-free grammar* (SCFG) is a CFG $G = (N, \Sigma, P, \mathsf{S})$ that assigns to each production rule $\lambda \in P$ a probability $\varphi(\lambda) : \lambda \to \mathbb{R}$. For any $\alpha \in N$,

$$\sum_{i=1}^{k} \varphi(\alpha \to \beta_i) = 1$$

must hold, where $\beta_1, \beta_{...}, \beta_k$ are all the possible productions from α. The probability $P(S, \pi \mid G)$ that a sequence S using a parse tree π is generated given G is the product of all probabilities $\varphi(\alpha \to \beta)$ for all used $\alpha \to \beta$ in π. The probability $P(S| G)$ that sequence S is generated given G is the sum over $P(S, \pi \mid G)$ for all possible parse trees π that generate S.

The idea of applying SCFGs for RNA analysis is to use it to build a model from an RNA family that can parse and score target sequences. Target sequences or specific parses from the model generating the sequences with a high probability receive a high score. This score will suggest a possible homology between the RNA family from which the model was built and the target sequence.

A limitation of SCFGs is that an emission of a terminal or nonterminal symbol only depends on the available nonterminal symbol and production rule. That is, SCFGs do not contain information about the columns of the alignment of the query RNA, such as base frequencies in each column or the alignment length. This hinders the use of SCFGs for RNA homology search. This limitation is overcome with the formulation of SCFGs to model RNAs called covariance models (CMs). Like the CFG of the example above, CMs contain nonterminal symbols for base emissions and structure modeling. To incorporate position specific information about the input multiple sequence alignment, like it is done in pHHMs, repetitive nonterminals for generating the primary and secondary structure are connected via transitions. A transition from a nonterminal to another has a certain probability. Base emitting nonterminals have a direct correspondence to one unpaired alignment column or to two paired columns. Therefore, they are also assigned base emission probabilities reflecting the distribution of the bases observed in the specific column(s). All nonterminals of a CM are called *states*.

Given a structure-annotated RNA multiple sequence alignment, to construct a CM we must, in a first step, define its structure topology connecting its states. In a second step, we compute the state emission and transition probabilities.

The CM topology is based on a tree-like structure resembling the consensus secondary structure of the sequences in the input alignment. In fact, this structure, called *guide tree*, is the parse tree of the consensus structure. Because the guide tree represents the consensus of both the structure and sequence of an RNA family, certain columns of the alignment are ignored, e.g. columns consisting mostly of gaps. Here, we assume that the consensus columns, i.e. columns that are not ignored, are given. For an example of a guide tree built from an RNA alignment annotated with a consensus secondary structure, see Figures 2.8 (A) and (B). The guide tree has different types of nodes. The first five node types we list below are strictly required to define the tree topology and do not have a direct relation to alignment columns.

1. A ROOT node is used at the top of the tree. See an example in Figure 2.8 (B).
2. BIF nodes are used for bifurcations (i.e. branching) of multiple stem-loops and multi-branch loops. Observe in the example in Figure 2.8 (A) that the consensus structure $\widehat{R}$ contains two base pairs $(2, 4)$ and $(7, 9)$. These base pairs induce each a stem-loop substructure described by a branch of the guide tree in subfigure (B). Hence, these stem-loops cause a bifurcation of the tree into two branches.

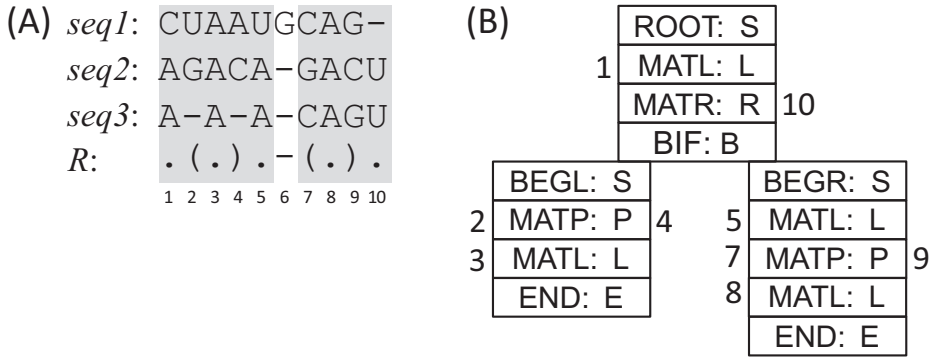


Figure 2.8: (A) Example of an RNA multiple alignment consisting of sequences *seq1*, *seq2*, and *seq3* annotated with a consensus secondary structure string R. Consensus alignment columns are highlighted in gray. (B) Guide tree of the alignment in (A) resembling its consensus secondary structure and alignment columns. The numbers beside each node in the tree indicate the corresponding column of the alignment. S, L, R, B, P, and E within each node are the associated states.

3. BEGL nodes are used at the beginning of a left branch of a bifurcation. In the example in Figure 2.8, the left branch is the first stem-loop from left to right (5' to 3' direction) to which base pair $(2, 4)$ belongs.
4. BEGR nodes are used at the beginning of a right branch of a bifurcation.
5. END nodes are used at the end of the tree or branches of bifurcations.

The following three types of nodes correspond directly to one or two alignment columns.

1. MATP (match base pair) nodes correspond each to two base-paired columns of the alignment. As an example, see in the guide tree in Figure 2.8 (B) the two MATP nodes corresponding to the two base pairs $(2, 4)$ and $(7, 9)$ in $\widehat{R}$ in subfigure (A).
2. MATL (match base leftwise) nodes correspond to a column of the alignment of an unpaired base on the left-hand side of a base pair. MATL nodes are also used for columns within base pairs, e.g. loops, and columns of an alignment without base pairs. As an example, the MATL nodes in the guide tree in Figure 2.8 (B) correspond to alignment columns 1, 3, 5, and 8.
3. MATR (match base rightwise) nodes correspond to a column of the alignment of an unpaired base on the right-hand side of a base pair. In the example in Figure 2.8, the only MATR node corresponds to position 10 of the alignment. Note that position 5 of the alignment is considered for modeling to be on the left of the second stem-loop from left to right and, consequently, it is modeled with a MATL node.

To enable the generation of a sequence by the guide tree, its nodes are assigned each a state, i.e. a nonterminal symbol. For emissions of single bases and base pairs there are three states. These are

node	state (guide tree)	production rule	states (CM)
MATP	P	$\mathsf{P} \rightarrow aWb$	MP, ML, MR, IL, IR, D
MATL	L	$\mathsf{L} \rightarrow aW$	ML, IL, D
MATR	R	$\mathsf{R} \rightarrow Wb$	MR, IR, D
ROOT	S	$\mathsf{S} \rightarrow W$	S, IL, IR
BIF	B	$\mathsf{B} \rightarrow \mathsf{SS}$	B
BEGL	S	$\mathsf{S} \rightarrow W$	S
BEGR	S	$\mathsf{S} \rightarrow W$	S, IL
END	E	$\mathsf{E} \rightarrow \varepsilon$	E

Table 2.1: Nodes that build up a guide tree for the construction of a covariance model, state associated to each node in the guide tree, corresponding production rule, and states assigned to each type of node in a covariance model. W is a symbol from the set of nonterminals $\{\mathsf{P}, \mathsf{L}, \mathsf{R}, \mathsf{S}, \mathsf{B}, \mathsf{E}\}$ and a and b are terminal symbols representing an arbitrary base. Table adapted from [83].

state P, assigned to MATP nodes for generating *base pairs*, and states L and R, assigned to MATL and MATR nodes, respectively, for generating bases on the *left-* and *right-hand* side of the state (also called *leftwise* and *rightwise* base generation). Since there can be 4×4 different base pairs, P has 16 emission probabilities, and L and R have each 4 emission probabilities. There are also three states which do not emit bases but are required for the nodes defining the guide tree topology. These are state B (*bifurcation*), assigned to BIF nodes, state S (*start*), assigned to ROOT, BEGL, and BEGR nodes, and state E (*end*) assigned to END nodes. B, S, and E have emission probability 1, because B and S can only emit nonterminal symbols and E can only emit the empty string ε. The types of guide tree nodes with each associated state and production rule are summarized in Table 2.1. Observe also in Figure 2.8 (B) the guide tree example with states assigned to each type of node.

A CM must allow for variations of the sequence and structure relative to the input RNA multiple sequence alignment, so that it can be used to parse putative homologous sequences not occurring in the input alignment. However, a guide tree can only represent the single RNA whose primary and secondary structure corresponds to the consensus of the alignment. To obtain a CM from a guide tree, we assign each node not only one but multiple states. These are essentially the same states used in the guide tree, but some are expanded to various "specialized" states to differ between base emissions that correspond to a base match or an insertion. Base emitting states receive a prefix M and a prefix I indicating a match and an insertion, respectively. States P, L, and R, therefore, become MP, ML, MR, IR, and IL. A new state D is also created to model base deletion. In summary, the nodes of a CM and their assigned states become as follows.

- Node MATL: states ML, IL, and D (match leftwise, insert leftwise, and deletion, respectively).

- Node MATR: states MR, IR, and D (match rightwise, insert rightwise, and deletion, respectively).
- Node MATP: states MP, ML, MR, IL, IR, and D. Here, ML and MR indicate a match of the base occurring on the 5' and 3' side of the pair, respectively; IL and IR indicate an insertion on the 5' and 3' side of the pair, respectively.
- Node BIF: state B (bifurcation).
- Node ROOT: states S, IL, and IR (start, insert leftwise, insert rightwise, respectively).
- Node BEGL: state S (start).
- Node BEGR: states S and IL (start and insert leftwise, respectively).
- Node END: state E (end).

All node types of a CM and respective states are summarized again in Table 2.1.

Once the nodes are assigned their respective states, states are connected via transitions. States may transition to all insert states of the same node and to all non-insert states of the next node. Insert states have a state transition to themselves. In nodes modeling base pairs, insert states IL have a transition to insert states IR but not vice versa. B states transition to two S states and E states do not transition do any state. The final CM (until here without calculated state transition and emission probabilities) is a directed graph without cycles, except self transitions of insert states. For an example of a CM showing its state transitions, see Figure 2.9. We note that, using a CM and its guide tree, each sequence in the input multiple sequence alignment can be converted unambiguously to a parse tree. In this parse tree, the bases of the sequence and also the gaps, as they appear in the alignment, are assigned to states of the CM. Conversely, the CM is able to generate the input sequences via a traversal of the CM parse tree of each sequence beginning at its root state and ending at its end states. See Figure 2.10 (A) for an example of a CM traversal and Figure 2.10 (B) for the respective parse tree generating a sequence.

The second step in the construction of a CM is to compute the state transition and emission probabilities. For this computation, we count, in the parse trees of the sequences in the input alignment, the number of times each particular transition and emission occurs. Let $A_{k,q}$ be the number of counted transitions from a state k to a state q. The transition probability from a state k to a state q is defined as

$$a_{k,q} = \frac{A_{k,q}}{\sum_{q'} A_{k,q'}} \tag{2.9}$$

where q' is any state to which a transition from k is possible. As an example, observe the transition probabilities in Figure 2.11 (A) computed from the parse trees in Figure 2.10 (B). Now let $E_q(b)$ be the number of observed emissions of a base b in state q. The emission probability of base b in state k is defined as

$$e_k(b) = \frac{E_k(b)}{\sum_{b'} E_k(b')} \tag{2.10}$$

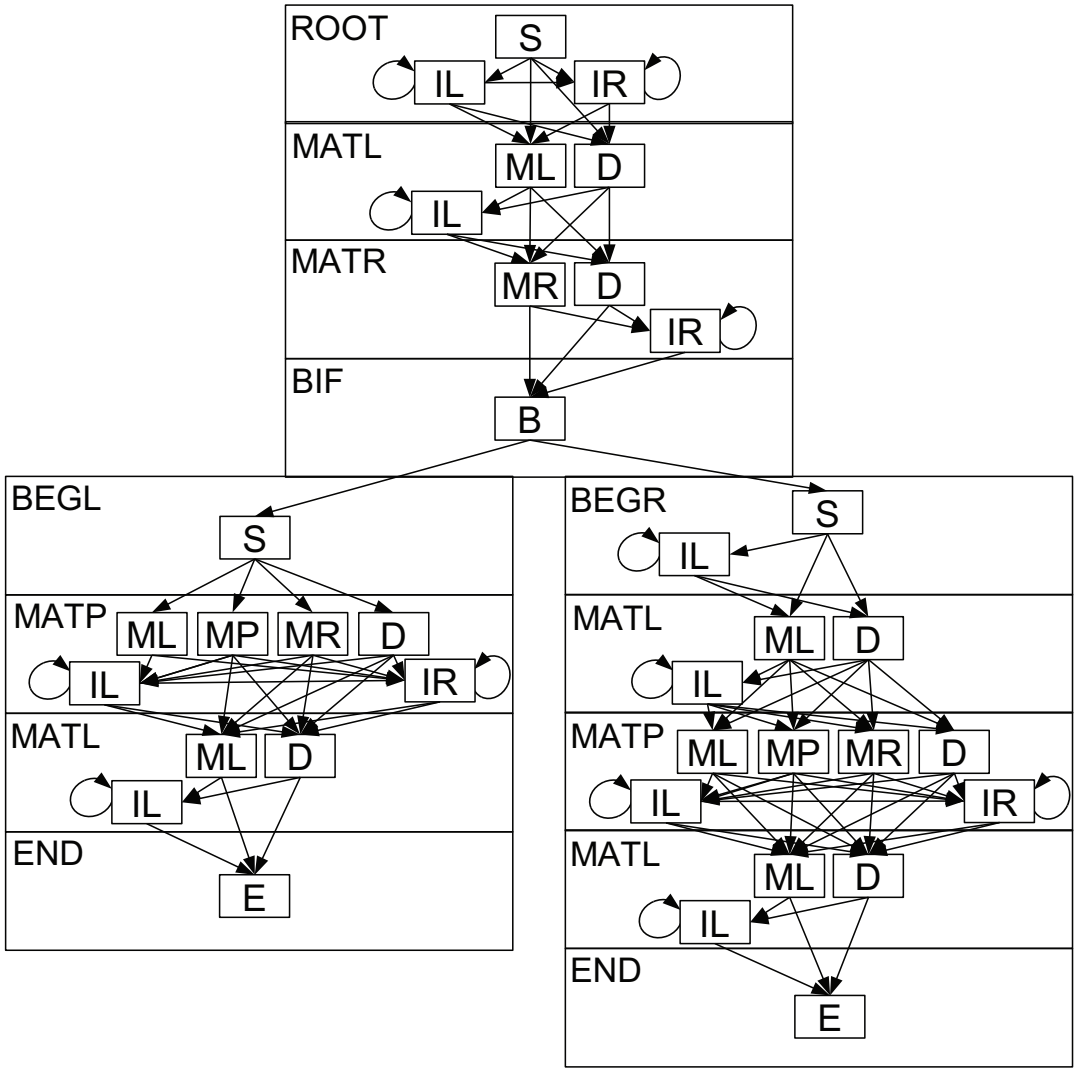


Figure 2.9: Covariance model obtained from the guide tree shown in Figure 2.8 (B). The names of the states in small rectangles, which are mostly derived from the names of the states in the guide tree by adding prefix M (match) and I (insertion), are grouped according to the type of node. The arrows indicate the allowed state transitions.

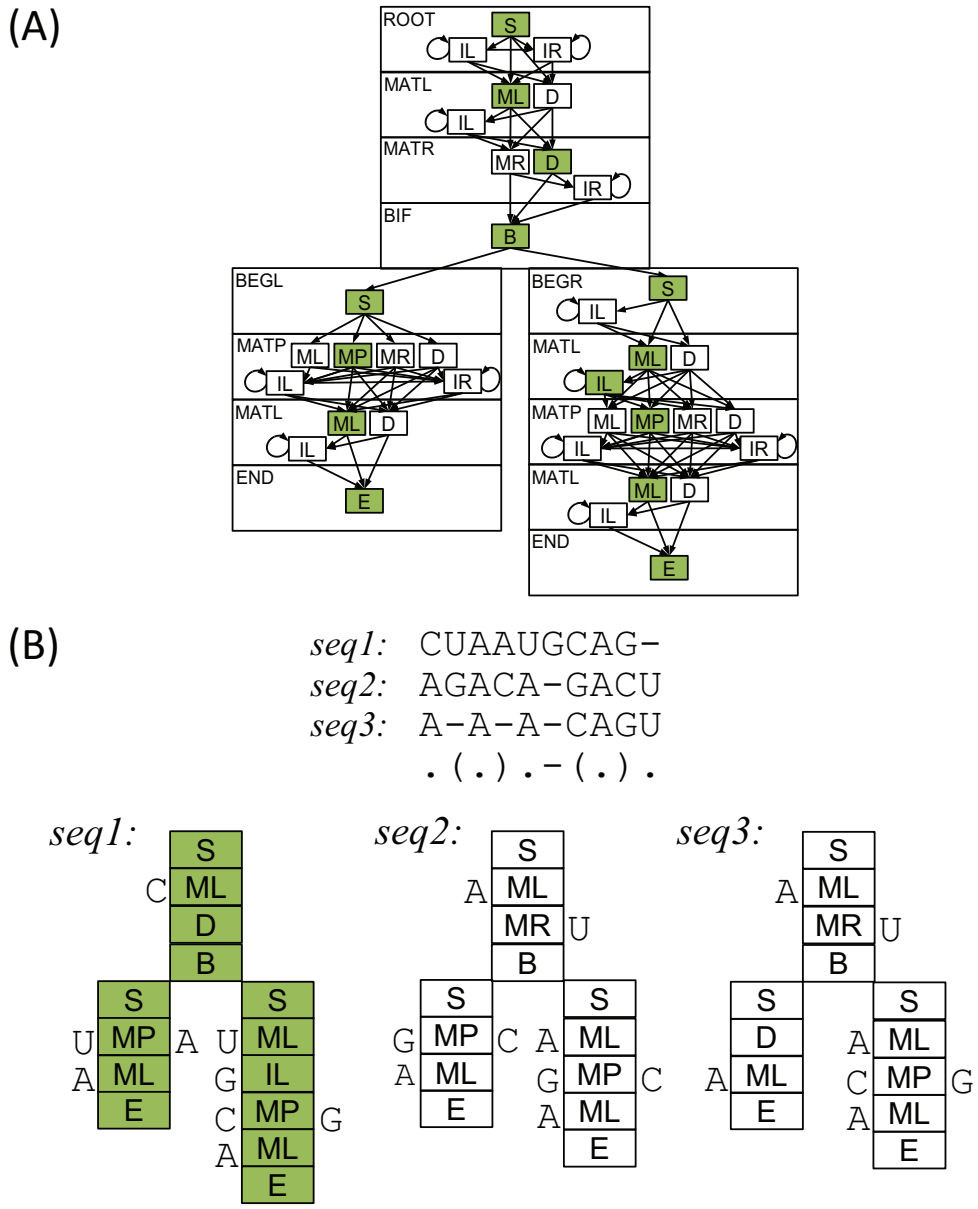


Figure 2.10: (A) Covariance model from Figure 2.9 highlighting in green the path from the root to its ending states that leads to the parse tree of sequence *seq1* from Figure 2.8 and shown again here in (B). (B) Parse trees of sequences *seq1*, *seq2*, and *seq3* for the given covariance model. The characters beside each node are the bases emitted by the node state. Note that a parse tree corresponds to exactly one path in the covariance model, as shown for the parse tree of *seq1* highlighted in green.

where b' is any base from $\mathcal{A}$. Observe, as an example, the emission probabilities in Figure 2.11 (B) computed from the parse trees in Figure 2.10 (B).

Parameters $a_{k,q}$ and $e_k(b)$ are the *maximum likelihood estimators* for the CM [39]. That is, they maximize the probability that the CM generates the sequences of the input alignment.

A problem that can occur with this simple computation of transition and emission probabilities is the overfitting of the CM when it is used for homology search. In the extreme case, transitions and emissions not occurring in the input alignment receive probability zero. This can be observed in the example in Figure 2.11. Consequently, instead of penalizing e.g. the occurrence of a base in the target sequence corresponding to a position in the model where that base was not seen, the parse of the complete target sequence will also have probability zero and therefore be forbidden. Consequently, the target sequence will not be considered homologous. To avoid this, pseudocounts called *priors* are added to $A_{k,q}$ and $E_q(b)$. If all priors are set equally, they are said to be *uninformative*. Uninformative priors, however, can disproportionately affect models built from only a few sequences. A better choice is to compute *informative* biologically motivated priors, using e.g. mixture Dirichlet densities [84]. These rely on the base distribution typically occurring in columns of multiple alignments of a larger dataset. Mixture Dirichlet densities, combined with the observed counts in the input multiple sequence alignment, were shown to considerably improve the sensitivity and specificity of CMs [85].

Using covariance models for RNA homology search and alignment computation

Consider a target RNA sequence S and a CM θ built from a query RNA family. If S can be generated from θ with a reasonably high probability $P(S, \pi \mid \theta)$ using CM parse tree π, then the usual assumption is that S and the query family are probably homologous. $P(S, \pi \mid \theta)$ is obtained by simply multiplying the state emission and transition probabilities observed during a traversal of the parse tree π that generates S. Alternatively, we can compute $P(S| \theta)$ by summing over $P(S, \pi \mid \theta)$ for all possible parse trees π that generate S. We remark that, commonly, the transition and emission probabilities are converted into log-odds scores and the sequence is then scored by summing up these scores.

In practice, in homology search sequence S can have a large length n, making it more sensible to score substrings S' of S of a certain length $m < n$. To score these substrings, a solution is to consider the parse tree π^* that generates each S' with the highest probability, i.e. for any π, $P(S, \pi \,|\, \theta) \leq P(S, \pi^* \,|\, \theta)$. Given S and CM θ, substrings S' of S can be scored using a CM version of the Cocke-Younger-Kasami (*CYK*) algorithm [86, 87, 88, 39], which simultaneously finds the "best" parse tree π^* for each substring. Another way to score substrings S' is to sum up the scores of all parse trees that generate S', obtaining the score equivalent of $P(S|\theta)$. This can be done using the *Inside* algorithm [39]. *Inside* and *CYK* use dynamic programming and require $\mathcal{O}\left(nm^3\right)$ time and

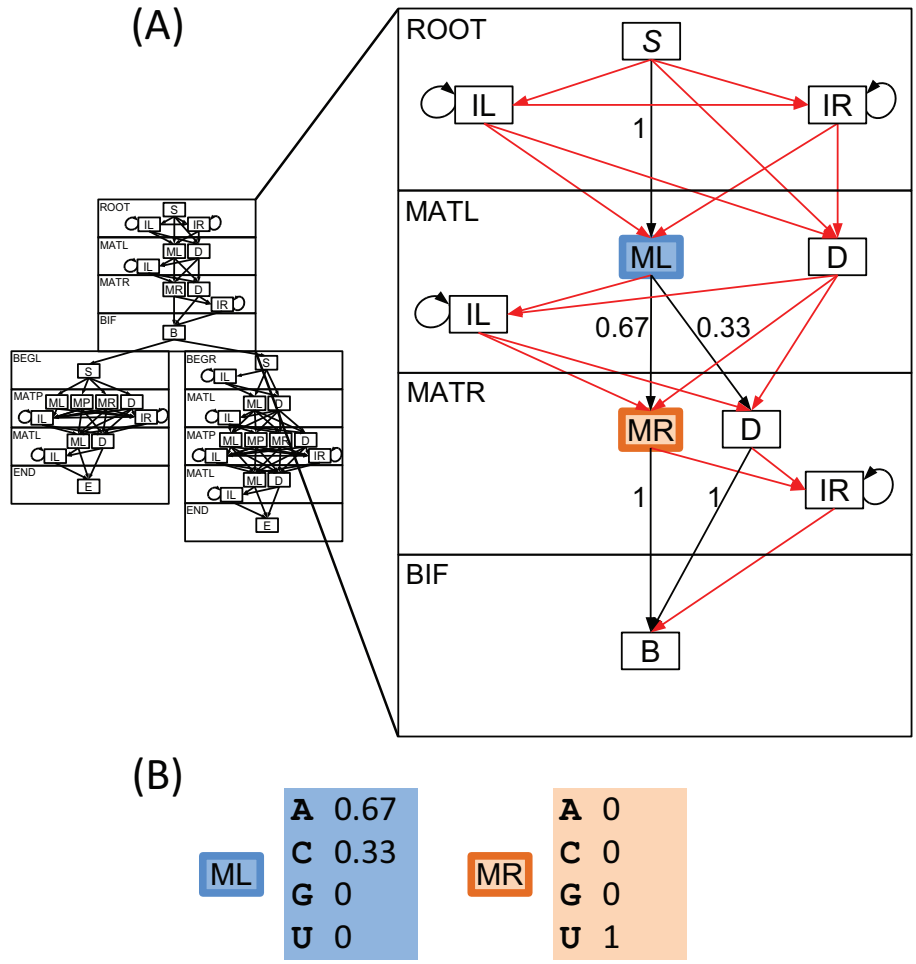


Figure 2.11: (A) First four nodes of the covariance model in Figure 2.9 showing its state transition probabilities. State transitions in red have probability zero. (B) Base emission probabilities of the colored states ML and MR for this covariance model. The transition and emission probabilities are computed from the parse trees in Figure 2.10 (B), remarking that the emission probabilities follow the distribution of the bases observed in the corresponding columns of the multiple alignment used to build the covariance model.

$\mathcal{O}\left(m^3\right)$ space. Notably, *CYK* and *Inside* are analogous to the *Viterbi* and *Forward* algorithms [39] for (profile) hidden Markov models.

The *CYK* algorithm suggests that a given CM can be used to compute a multiple sequence alignment. For instance, consider a set of RNAs without secondary structure annotation. For each of these sequences, we can compute a CM parse tree, from which the columns of each sequence in the alignment can directly be read. Hence, computing a multiple sequence alignment consists in computing the best parse tree of each sequence, accomplished with the *CYK* algorithm.

Software using covariance models

A major disadvantage of CMs is the large time complexity of the *CYK* and *Inside* algorithms. For this reason, CMs are often used in combination with pre-filters that use only sequence or some amount of structure information of the query RNA. For example, an initial step in building an RNA family of the Rfam database [34] is searching the Rfam sequence database with *Blast* [35] using query sequences from the family's seed alignment. The sequences below some E-value threshold are then searched using the *Infernal* software [89, 40] with the CM built from the family's seed alignment annotated with secondary structure. Other tools apply different search strategies. *RaveNnA* [90] converts CMs into pHHMs to take advantage of the reduced complexity of pHHMs. Multi-segment *CYK* (*MSCYK*) [91] simplifies CMs for the computation of ungapped structural alignments. Structure-based query-dependent banding (QDB) [85] accelerates CM searches by performing computationally expensive recursions only within bands of the dynamic programming matrix where the optimal alignment is likely to lie.

The most prominent software using CMs is *Infernal* [89, 40]. It consists of tools for the CM construction and database search of a query RNA multiple sequence alignment annotated with a consensus secondary structure. It can also be used to make sequence- and structure-based RNA sequence alignments. The CM constructed by *Infernal* follows the description above. State transition and base emission probabilities incorporate mixture Dirichlet priors for more sensitive and specific searches. Since the algorithms for database searches using CMs are too slow for practical use, *Infernal* uses a filtering pipeline comprising two main stages. The first stage consists in applying filters based on hidden Markov models (HMMs), i.e. pure sequence-based filters, whereas the second stage consists in searching the (sub)sequences surviving the first stage with CMs. As of version 1.0 of *Infernal*, the first stage consists of a single filtering step relying on an implementation of pHHMs as in the *RaveNnA* tool mentioned above. The second stage consists of two steps. First, accelerated CM searches are performed using query-dependent banding (QDB). And second, CM searches use the slower but more specific *Inside* algorithm. As of this writing, the current version 1.1 of *Infernal* incorporates a more sophisticated combination of algorithms in the filtering pipeline. In the first stage, *Infernal* scores sequences using the following HMM-based algorithms: (1) *SSV* (*Single Segment Viterbi*) computes and extends high-scoring ungapped alignments, (2) *Viterbi* with gaps, and (3) local *Forward*, i.e. the Forward algorithm operating in local mode.

These three algorithms, which are also implemented in the *HMMER3* software package [92], are responsible for large speedups compared to previous versions of *Infernal*. In the second stage, the (sub)sequences that survived each filter of the first stage are scored using banded versions of the *CYK* and *Inside* algorithms and are finally processed with the standard *Inside* algorithm. For a more detailed description of the *Infernal* filtering pipeline, see the *Infernal* manual [93]. We note that, for accelerated searches of query RNAs without base pairs, *Infernal* applies only HMM-based filters, avoiding the CM-based stage. This is possible, since CMs mainly differ from HMMs by their ability to model base pairs. In the absence of these, CMs and HMMs present the same sensitivity. Obviously, filters are a trade-off between speed and sensitivity. Because stringent filters accelerate search but can eliminate from the search space potentially high scoring sequences, they must be used with caution. For a discussion about the filters implemented in *Infernal* 1.0, their tuning and effect on search, see chapter 4 of [94].

Another program using CMs is *RSEARCH* [74]. An important difference to *Infernal* is that it builds a CM from a single query RNA sequence annotated with secondary structure. Hence, the CM topology is obtained from the guide tree of the unique sequence rather than from a multiple alignment consensus. Because no emission and transition probabilities can be computed from the single query RNA, *RSEARCH* uses a position independent substitution matrix called RIBOSUM and gap penalties to score target sequences. RIBOSUM has 4×4 and 16×16 entries for the substitution of single bases and base pairs, respectively, which are computed analogously to entries in BLOSUM matrices used in protein searches [95]. That is, they are log-odds scores of the base frequencies observed in alignments of homologous RNAs. The gap penalties are computed using a standard affine gap penalty formulation as $\alpha + \beta n$, where α is the gap opening penalty, β is a gap extension, and n is the size of the gap. Despite the different used scoring, *RSEARCH* uses for homology search the same *CYK* algorithm as *Infernal*. Since *RSEARCH* enables the search for homologous RNAs from a single query RNA, we remark, as in [94], that it partially addresses the problem of model overfitting discussed above. However, its performance is largely influenced by the quality of the used RIBOSUM matrix (see [74]).

The Rfam RNA family database

Rfam [34] is a database of families of homologous non-coding RNAs sharing sequence and structure information, primarily created for genome annotation. Each family is represented by a seed and a full alignment annotated with a consensus secondary structure and a CM. The seed alignment contains representative sequence members of the family and is hand-curated or experimentally validated from published literature. From the seed alignment, a CM is built using the *Infernal* software. To search the Rfam sequence database, called Rfamseq, for putative homologous sequences, one first applies *BlastN* [35] using sequences from the seed alignment. The sequences in Rfamseq surviving an E-value threshold are then searched with the built CM. Found putative homologues are aligned using the CM and merged with the seed alignment to form the full alignment. Rfam was

first released in 2002 containing 25 families (version 1.0), whereas the latest release from 2012 contains 2,208 families (version 11.0). This tremendous growth became possible by the automatic maintenance of the full alignments enabled by *Infernal* using CMs. But perhaps, more importantly, it can also be credited to the search time improvements of *Infernal* achieved by incorporating a filtering pipeline as described above. This allowed to take into account many new non-coding RNAs reported since 2002.

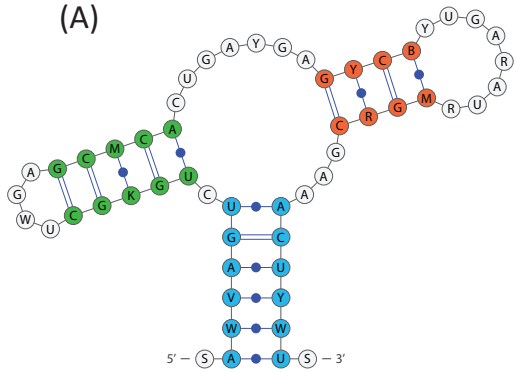
2.6 Descriptor-based search methods

Descriptor-based RNA homology search methods provide a language for defining RNA motif descriptors, here also called *sequence-structure patterns*, containing primary and secondary structure properties of an RNA family. The pattern for a family must be predefined using e.g. information from an externally computed multiple sequence-structure alignment of the specific RNA family. In addition to a language, these tools provide a method to search with the patterns in large sequence databases.

Sequence-structure patterns supported by the tools in this category can, in general, describe all RNA secondary structure elements (see Figure 1.1). The patterns normally consist of strings of IUPAC characters including ambiguous symbols, e.g. N meaning any base from alphabet $\mathcal{A}$ (formally, $\varphi(N) = \mathcal{A}$), and of base pairing information about these characters. These strings of IUPAC characters indicate which bases can *match* designated positions within the pattern, whereas the base pairing information further constrains matching bases. For example, allowing only complementary bases to form pairs, two paired positions encoded with character N have only 6 combinations of possible matching bases instead of 4×4 for two unpaired positions. Commonly, a number of allowed errors such as base mismatches can be specified. Overall, a sequence-structure pattern describes a subset of strings from $\mathcal{A}^* \setminus \{\varepsilon\}$ matching the pattern. The goal of descriptor-based search methods is to find all occurrences of the substrings in this subset in a target database.

One of the most popular tools in this category is *RNAMotif* [42]. A sequence-structure pattern in *RNAMotif*'s descriptor language consists basically of paired and unpaired elements. A single-stranded element corresponding to continuous unpaired positions is described with the expression ss and a helical element, e.g. the stem of a stem-loop, is described with $h5$ and $h3$ denoting the 5' and 3' sides of the stem. A pattern is defined as a list of such structural elements stating their relative positions within the pattern; see an example in Figure 2.12 (B). Immediately following each element, the user can specify in parenthesis the sequence information of the element, its minimum and maximum length, a number of allowed mismatches, among other information. A description of *RNAMotif*'s descriptor language is available in a detailed manual [96]. Previous tools in this category are *RNAMOT* [97], a variant implementation of it called *RNABOB* [98], and *PatScan* [99]. Patterns for these tools are also defined as a list of paired and unpaired elements, with some differences in syntax and flexibility. For instance, *RNABOB* allows for mismatches in paired and unpaired

(A)



(B) *RNAMotif*

```
parms
 wc += gu;

descr
 ss(seq="^S$")
 h5(minlen=6, maxlen=8)
  ss(seq="^C$")
  h5(mismatch=1, seq="^UGKGC$")
   ss(mismatch=2, seq="^UWGA$")
  h3(seq="^GCMCA$")
  ss(seq="^CUGAYGA$")
  h5(seq="^GYCB$")
   ss(seq="^N\{0,3\}YUGARAUR$")
  h3(seq="^MGRC$")
  ss(seq="^GAA$")
 h3
 ss(seq="^S$")
```

(C) *RNABOB*

```
s1 h1 s2 h2 s3 h2' s4 h3 s5 h3' s6 h1' s7

s1 0 S
h1 0:0 **NNNNNN:NNNNNN**
s2 0 C
h2 1:1 UGKGC:GCMCA
s3 2 UWGA
s4 0 CUGAYGA
h3 0:0 GYCB:MGRC
s5 0 [3]YUGARAUR
s6 0 GAA
s7 0 S
```

Figure 2.12: (A) Consensus primary and secondary structure of RNA family Hammerhead ribozyme (type III) (Rfam Acc.: RF00008). (B) Sequence-structure pattern in *RNAMotif*'s descriptor language capturing primary and secondary structure properties of this family. The pattern is relaxed to match between 6 and 8 complementary base pairs in the positions corresponding to the stem drawn in blue. In addition, the pattern allows for mismatches in the structural elements specified with the keyword *mismatch* and for up to 3 insertions in the loop closed by the stem drawn in orange. (C) Sequence-structure pattern in *RNABOB*'s descriptor language for the same RNA family. After the list of structural elements defining their position within the pattern, each element is more precisely defined. The single or pair of numbers, e.g. 1:1, following an identifier of an unpaired or paired element, respectively, is the number of allowed mismatches for the respective element. This pattern also allows to match between 6 and 8 complementary base pairs in the positions corresponding to the stem drawn in blue, where * means 0 or 1 N. Similarly, "[3]" used in element *s5* means 0 to 3 Ns.

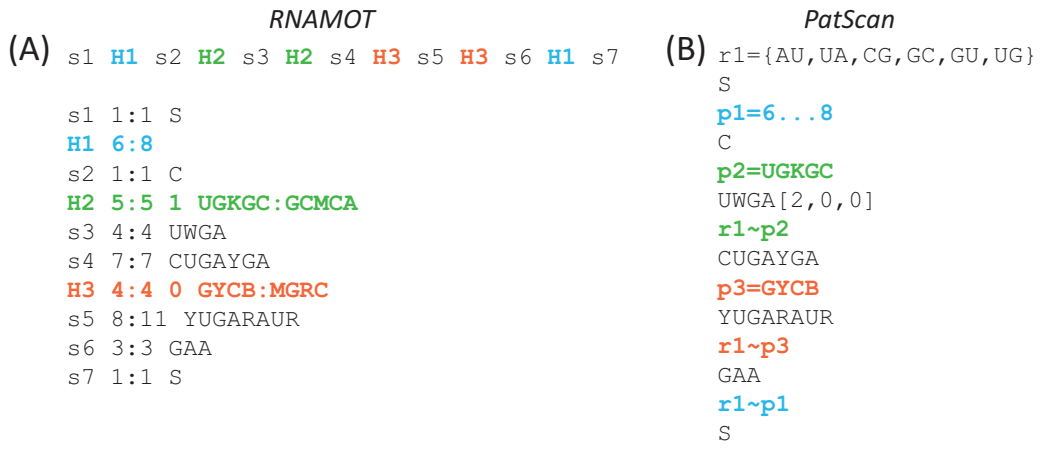


Figure 2.13: (A) Sequence-structure pattern in *RNAMOT*'s descriptor language for the RNA structure shown in Figure 2.12 (A). Despite sharing a similar syntax with *RNABOB*, here the pair of numbers, e.g. 1:1 or 6:8, following each element identifier is the minimum and maximum length of the element (instead of a number allowed mismatches as in *RNABOB*). In *RNAMOT*, mismatches are only allowed in paired elements, specified after the pair of numbers denoting possible lengths of the element. In this example, 1 mismatch is allowed for the paired element *H2* in green. (B) Sequence-structure pattern in *PatScan*'s descriptor language. $r1$ defines a set of allowed base pairs. Unpaired elements are simply given as a string of IUPAC characters (not preceded by character p) optionally followed by three numbers in brackets meaning, in this order, a number of allowed mismatches, insertions, and deletions. As an example, matches to unpaired element UWGA can contain up to 2 mismatches. Strings preceded by a p are paired, whereas their complement is specified with $\sim p$. A range, e.g. 6...8 specified for element $p1$, is the minimum and maximum length of the respective element.

elements, whereas *RNAMOT* only supports mismatches in paired elements. For an example of a pattern for *RNABOB*, *RNAMOT*, and *PatScan*, see Figures 2.12 (C), 2.13 (A), and 2.13 (B), respectively. Another tool, *Palingol* [100], provides a powerful descriptor language to model primary and secondary properties of an RNA molecule. However, despite being powerful, its language complexity may discourage its use by biologists. Except for *RNABOB*, all these tools provide a method to score and rank matches, e.g. to prefer matches with a lower number of mismatches (*RNAMotif* and *RNAMOT*), longer helices (*RNAMOT*), or minimum free energy (*RNAMOT*). *PatScan* and *Palingol* can score matches using position weight matrices typically reflecting base frequencies in columns of a multiple sequence alignment of the sought RNA family.

The search for occurrences of a given sequence-structure in a target sequence is performed in a scanning fashion. For each position of the sequence, tools like *RNAMotif* and *RNAMOT* try to sequentially match each paired and unpaired element in the pattern. If every element can be matched, then an occurrence of the pattern can be scored and reported. Note that the same element sometimes matches different substrings due to allowed errors, e.g. mismatches, and variable length specified by the user. Hence, if an element cannot be matched using any of its variations, the tools step back to a previously matched element, try to match it with e.g. a different length, and proceed with the next element in a recursive manner. Due to the scanning of the target sequence, all tools in this category have a running time that scales at least linearly in the size of the sequence.

2.7 Concluding remarks on existing RNA homology search methods

Given one or more homologous query RNAs belonging to the same family and a target sequence to be searched for sequence and structure similarities with the query, the choice for a specific RNA homology search method (see a summary in Table 2.2) can depend on various properties of the query and the target as follows.

- *Number of homologous query sequences*. If two or more homologous sequences are available, it is important that information from the primary and secondary structure of all members of the family can be combined into one query model for homology search. This is supported by the models used by *LocARNAscan*, *Infernal*, *ERPIN*, as well as the descriptor-based methods *RNAMotif*, *RNABOB*, *RNAMOT*, *PatScan*, and *Palingol*. In the case of only one query sequence, all these methods can still be used, whereas *RSEARCH* and *ERPIN* may better balance sensitivity and specificity by using position independent substitution matrices. In this second scenario, also methods performing pairwise comparisons may be applied.
- *Local or global sequence and structure similarity*. If two RNA sequences to be compared are expected to contain sequence and structure similarities throughout their extension, then methods capable of performing global sequence-structure alignments can be appropriate. These

Method	Description	Time
Comparative methods (Sankoff-style simultaneous alignment and folding)		
Foldalign [65, 43]	Computes pairwise local or global sequence-structure alignment simplifying Sankoff's algorithm [64] by (1) not allowing for branching structures or (2) using a heuristic for pruning the used dynamic programming matrices	$\mathcal{O}\left(n^4\right)$
Dynalign [66]	Computes a pairwise global sequence-structure alignment simplifying Sankoff's algorithm by limiting the span M between aligned bases	$\mathcal{O}\left(n^3M^3\right)$
PMcomp [67]	Computes pairwise global sequence-structure alignment saving running time by using precomputed base pairing probabilities from each individual sequence	$\mathcal{O}\left(n^4\right)$
LocARNA [44]	*PMcomp* successor for computing pairwise local alignment saving additional running time by ignoring base pairings with low probability	$\mathcal{O}\left(n^2+m^2\right)$
LocARNAscan [68]	Scanning variant of the *LocARNA* method suitable for searching for relatively short homologs in a larger sequence database of known base pairings probabilities. Limits the span L between aligned base pairs	$\mathcal{O}\left(nmL^2\right)$
Comparative methods (requiring known secondary structure)		
MARNA [69]	Computes a multiple alignment of a set of RNAs with known structure by using (1) the algorithm of Jiang *et al.* [70] for pairwise sequence-structure alignment and (2) the multiple sequence alignment tool *T-Coffee* [71]	$\mathcal{O}\left(m^2n^2\right)$
RNAforester [72]	Computes a pairwise local or global alignment of secondary structures represented as trees of maximal degree d	$\mathcal{O}\left(mnd^2\right)$
Method using secondary structure profiles (not covariance models)		
ERPIN [73, 41]	Builds a secondary structure profile from a multiple sequence alignment annotated with a consensus secondary structure, which it then uses for database searches	$\mathcal{O}\left(nm^2\right)$
Methods using secondary structure profiles (covariance models)		
Infernal [89, 40]	Builds a covariance model from a multiple sequence-alignment annotated with a consensus secondary structure, which it can then use for database searches	$\mathcal{O}\left(nm^3\right)$
RSEARCH [74]	Builds a covariance model from a single structure-annotated sequence and uses it combined with a position independent substitution matrix to score target sequences	$\mathcal{O}\left(nm^3\right)$
Descriptor-based methods		
RNAMotif [42]	Scans a target database searching for matches of a pattern provided by the user as a list of structural elements, each of which can allow for mismatches, insertions, and deletions of single bases or base pairs. Can score matched substrings by number of mismatches or matches of specific bases	$\mathcal{O}\left(nm\right)$
RNABOB [98]	Similar to *RNAMotif* using its own descriptor language. Cannot rank matches	$\mathcal{O}\left(nm\right)$
RNAMOT [97]	Similar to *RNAMotif* using its own descriptor language, but allows for mismatches only in paired elements. Can rank matched substrings by number of mismatches, helix length, and minimum free energy	$\mathcal{O}\left(nm\right)$
PatScan [99]	Similar to *RNAMotif* using its own descriptor language. Can score matches using position weight matrices	$\mathcal{O}\left(nm\right)$
Palingol [100]	Similar to *RNAMotif* using its own descriptor language. Can score matches using position weight matrices	$\mathcal{O}\left(nm\right)$

Table 2.2: Summary of RNA homology search methods. Column "Time" refers to the time required by the methods to compare two RNAs of length m and n or to search a sequence database of length n.

include *Foldalign*, *Dynalign*, *PMcomp*, *MARNA*, *RNAforester*, *Infernal*, and *RSEARCH*. If, however, the two sequences appear to be highly dissimilar and the intention is to discover shorter common motifs, then methods for local sequence-structure alignments can be recommended. These include *Foldalign*, *LocARNA*, *RNAforester*, *Infernal*, and *RSEARCH*. Note that some methods can operate to compute both global and local alignments.

- *Length*. Global and local alignments are a sensible way to compare sequences of similar length. However, the high time complexity of methods based on covariance models and of variants of the Sankoff algorithm enforces a limit on the length of the sequences. This limitation affects programs like *Foldalign*, *Dynalign*, and *LocARNA* despite their running time improvements compared to the original algorithm of Sankoff. In a different scenario, one is interested in searching for occurrences of relatively short sequence-structure patterns or motifs in large sequence databases. This can be accomplished by computing semi-global alignments, aligning the complete motif model to substrings of the target sequence. Semi-global alignment computation is supported by *LocARNAscan*, *Infernal*, *RSEARCH*, and *ERPIN*. Although offering good sensitivity and specificity, all these methods suffer from high computational demands and often can only be used with heuristics or filters that affect their sensitivity. Consequently, they are not well-suited for rapid database searches. An alternative is to search with sequence-structure patterns using descriptor-based methods.
- *Knowledge about secondary structure*. If the secondary structure of the RNAs to be compared is not known, then the Sankoff-style methods are most appropriate. These methods can combine information from the input sequences to infer a common secondary structure. Methods like *MARNA* and *RNAforester* can also be used, but the quality of the produced alignments can suffer from the poor quality of the folding of single RNAs. However, as already noted, all these methods suffer from high running time complexities.

These properties of query and target RNAs along with the respective applicable homology search methods are summarized in Table 2.3. From this analysis of the methods it is easy to conclude that a major limitation of these methods is their large running time, which is at least quadratic. While descriptor-based methods are more appropriate for this task, they still have a running time that scales at least linearly in the size of the sequence database. This makes searching ever-growing databases challenging. Furthermore, these methods can poorly handle mutations occurring on both the sequence and structure levels. For instance, allowed insertions, deletions, or mispairings must be predefined by the user at specific positions of the patterns. This can lead to insensitive searches, since RNAs often present low sequence conservation throughout their extension.

In the following chapters, we present novel methods for efficient matching of RNA sequence-structure patterns. In this way, we address the limitations of current methods by enabling fast sensitive and specific searches in large databases.

	many query sequences	one query sequence	global similarity	local similarity	similar length	very different lengths	known structure	unknown structure	speed
Foldalign		✓	✓	✓	✓			✓	−−
Dynalign		✓	✓		✓			✓	−−
PMcomp		✓	✓		✓		✓	✓	−
LocARNA		✓		✓	✓		✓	✓	−
LocARNAscan	✓					✓	✓	✓	−
MARNA	✓		✓		✓		✓		−
RNAforester			✓	✓	✓		✓		−
Infernal	✓		✓		✓	✓	✓		−
RSEARCH		✓	✓		✓	✓	✓		−
ERPIN	✓	✓				✓	✓		+
RNAMotif	✓	✓				✓	✓		+
RNABOB	✓	✓				✓	✓		+
RNAMOT	✓	✓				✓	✓		+
PatScan	✓	✓				✓	✓		+
Palingol	✓	✓				✓	✓		+

Table 2.3: Possible properties of hypothetically given query and target RNAs to be compared and suitable homology search methods for this task. For homology searches in large sequence databases, one prerequisite is the capability of the method to compare relatively short sequence-structure patterns or models of the query with substrings of a larger target database. Methods with this capability are indicated with a green check mark in column “very different lengths”. Another prerequisite are short running times, informally denoted with column “speed”. For more details, see main text.

3 Fast index-based bidirectional search for RNA sequence-structure patterns

3.1 Introduction

In the previous chapter, we described different approaches for sensitive and specific RNA homology search. We saw that comparative methods or methods using secondary structure profiles are too slow for searching large databases. Therefore, descriptor-based methods are best suited for rapid searches. However, since the running time of these methods also scales at least linearly in the size of the target sequence database, searching with these tools is also challenging when it comes to large databases. A solution with sublinear running time would require index data structures. Still, widely used index structures like suffix trees [51] or arrays [52] or the FM-index [53] perform badly on typical RNA sequence-structure patterns, because they cannot take advantage of the RNA structure information.

Here, we present a fast descriptor-based method and software for RNA sequence-structure pattern matching. The method consists of initially building an affix array [101], i.e. an index data structure of the target database. Affix arrays cope well with structural pattern constraints by allowing for an efficient matching order of the bases constituting the pattern. Structurally symmetric patterns like stem-loops can be matched inside out, such that first the loop region is matched and, in subsequent extensions, pairing positions on the boundaries are matched consecutively. Because the matched substring is extended to the left and to the right, this pattern matching scheme is known as *bidirectional search*. Unlike traditional left-to-right search where the two substrings constituting the stem region of the pattern are matched sequentially, in bidirectional search base complementarity constraints are checked as early as possible. This leads to a significant reduction of the search space that has to be explored and in turn to a reduced running time. We note that bidirectional search for RNA sequence-structure patterns was also presented by Mauri et al. in [102]. However, their method uses affix trees [103] instead of the more memory efficient affix arrays. Affix trees require with approximately 45 bytes per input symbol more than twice the memory of affix arrays (18 bytes per input symbol), making their application infeasible on a large scale. Moreover, their method traverses the affix tree in a breadth-first manner, leading to a space requirement that grows exponentially with increasing reading depth. We instead employ a depth-first search algorithm whose space requirement is only proportional to the length of the searched substring.

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

The description of our method closely follows [104].

3.2 Formal preliminaries

To formalize the concept of affix arrays and their application for bidirectional search of RNA sequence-structure patterns, we complement our definitions given above with the following definitions.

Definition 12 We denote the *reverse sequence* of a sequence $S = S[1]S[2]\dots S[n]$ with $S^{-1} = S[n]S[n-1]\dots S[1]$. The k–th *reverse prefix* of S is the k–th suffix of S^{-1}. For $1 \leq k \leq n$, S_k denotes the k–th suffix of S and $S_k^{-1} = (S^{-1})_k$ denotes the k–th reverse prefix of S.

Definition 13 A *sequence pattern* is a sequence $P \in (\mathcal{A} \cup \Phi)^*$, recalling that $\mathcal{A} = \{\text{A, C, G, U}\}$ and $\Phi = \{\text{R, Y, M, K, W, S, B, D, H, V, N}\}$. Let m denote its length $|P|$. An *occurrence* of P in a sequence S is a position i, $1 \leq i \leq n$, such that $S[i+k-1] \in \varphi(P[k])$ for all $1 \leq k \leq m$.

Definition 14 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.

Definition 15 A *match* or *occurrence* of $\mathcal{Q}$ of length m in an RNA sequence S is an occurrence i of P in S, such that for all base pairs $(l, r) \in \widehat{R} : (S[i+l-1], S[i+r-1]) \in \mathcal{C}$, where $\mathcal{C}$ is the set of complementary bases defined above. Note that, with this definition, here we only allow complementary bases to form base pairs. We define, in addition, $\mathcal{CS}$ as a mapping of a character $c \in \Phi \cup \mathcal{A}$ to the set of its complementary characters in $\mathcal{A}$, i.e. $\mathcal{CS}(c) = \{d \in \mathcal{A} |\ \exists e \in \varphi(c) : (d, e) \in \mathcal{C}\}$.

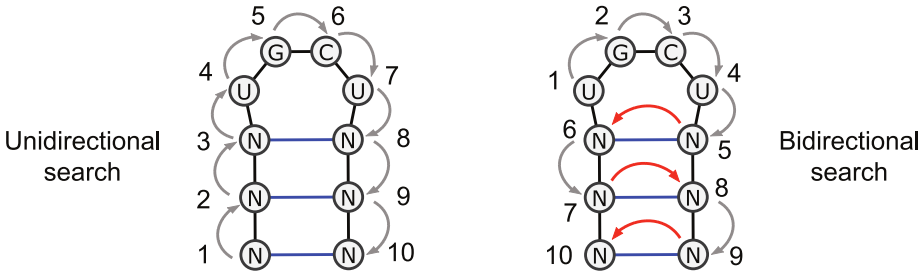


Figure 3.1: Unidirectional (left) and bidirectional (right) searches for the RNA sequence-structure pattern (RSSP) $\mathcal{Q} = (P, R)$ with P = NNNUGCUNNN and R = (((. . . .))), which represents a stem-loop structure of length $m = 10$. The numbers indicate the order in which the pattern characters are matched against the target sequence. In the unidirectional search, the characters are matched in a single direction, beginning (ending) with a character in $\varphi(P[1])$ ($\varphi(P[m])$). In the bidirectional search, the loop region of the pattern can be matched first. Then, pairing bases are matched consecutively by switching the search direction, represented by the red arrows.

In the following, structures described by RSSPs are non-branching. We also note that, for stating the space requirements of our index structures, we assume that $|S| < 2^{32}$, such that sequence positions and lengths can be stored in 4 bytes.

3.3 The affix array data structure

In [101] the theoretical concept of an index data structure called *affix array* is described. This index structure supports efficient unidirectional as well as bidirectional searches and is more space efficient than the affix tree [103, 105]. The term *unidirectional search* refers to the search for occurrences of a sequence pattern where the pattern characters are compared with sequence characters in a left-to-right (right-to-left) order, i.e. the already compared (matched) prefix (suffix), of the pattern is extended to the right (left). Notably, a change of the direction is not possible.

When searching for occurrences of sequence-structure patterns, however, unidirectional search cannot exploit the complementarity condition on base paired pattern positions. To utilize this condition as effectively as possible, both positions of a base pair need to be accessed immediately after each other. This is enabled by *bidirectional search*, which refers to methods where the direction of the match extension can be changed freely. Figure 3.1 illustrates the order of the character comparisons of a sequence-structure pattern in the unidirectional and bidirectional searches.

Until now, affix arrays have received little attention in bioinformatics. Presumably, this has been due to the lack of an open and robust implementation. As a consequence, their potential for efficient database search with RSSPs has hardly been recognized and the details of this data structure are

not widely known in the field. Therefore, we briefly recall the basic ideas of the affix array, which constitutes the central component of our *Structator* approach.

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

The affix array data structure of a sequence S is composed of six tables, namely $\mathsf{suf}_{\mathsf{F}}$ and $\mathsf{suf}_{\mathsf{R}}$, $\mathsf{lcp}_{\mathsf{F}}$ and $\mathsf{lcp}_{\mathsf{R}}$, and $\mathsf{aflk}_{\mathsf{F}}$ and $\mathsf{aflk}_{\mathsf{R}}$. They are called *suffix*, *longest common prefix*, and *affix link arrays* of S^{F} and S^{R}, respectively. Table $\mathsf{suf}_{\mathsf{R}}$ is also known as *reverse prefix array*. suf_X is an array of integers in the range 1 to $n+1$ specifying the lexicographic order of the $n+1$ suffixes of the string $S^X\$$. That is, $S^X_{\mathsf{suf}_X[1]}, S^X_{\mathsf{suf}_X[2]}, ..., S^X_{\mathsf{suf}_X[n+1]}$ is the sequence of suffixes of $S^X\$$ in ascending lexicographic order. Each of the tables $\mathsf{suf}_{\mathsf{F}}$ and $\mathsf{suf}_{\mathsf{R}}$ requires $4n$ bytes and can be constructed in $\mathcal{O}(n)$ time and space [106]. In practice non-linear time [107, 108] construction algorithms are often used as they are faster and require less space.

lcp_X is a table in the range 1 to $n+1$ such that $\mathsf{lcp}_X[1] = 0$, and $\mathsf{lcp}_X[i]$ is the length of the longest common prefix between $S^X_{\mathsf{suf}_X[i-1]}$ and $S^X_{\mathsf{suf}_X[i]}$ for $1 < i \leq n+1$. Each of the tables $\mathsf{lcp}_{\mathsf{F}}$ and $\mathsf{lcp}_{\mathsf{R}}$ requires n bytes and store entries with value up to 255, whereas occasional larger entries are stored in an exception table using 8 bytes per entry [109]. More space efficient representations of the lcp table are possible (see [110]). The construction of $\mathsf{lcp}_{\mathsf{F}}$ and $\mathsf{lcp}_{\mathsf{R}}$ can be accomplished in $\mathcal{O}(n)$ time and space given $\mathsf{suf}_{\mathsf{F}}$ and $\mathsf{suf}_{\mathsf{R}}$ [111]. In contrast to [101] where affix arrays were described using a terminology derived from tree-like data structures, we explain the underlying concepts of this data structure in terms of intervals in the suffix array suf_X. Two important concepts of affix arrays are suffix-intervals and lcp-intervals. An interval $[i..j]$ representing the set of suffixes $S^X_{\mathsf{suf}_X[i]}, ..., S^X_{\mathsf{suf}_X[j]}$, $1 \leq i \leq j \leq n+1$, of *width* $j-i+1$, is a ***suffix-interval*** in suf_X with *depth (prefix length)* $\ell \in \{0, \ldots, n\}$, or ***$\ell$-suffix-interval***, denoted $\ell - [i..j]$, if and only if the following three conditions hold:

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

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

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

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

i	$\mathsf{suf_F}[i]$	$\mathsf{lcp_F}[i]$	$\mathsf{aflk_F}[i]$	S^{F}_i
1	3	0	1	AGCUGCUGCUGCA
2	1	1		AUAGCUGCUGCUGCA
3	15	1		A
4	14	0	4	CA
5	11	1	5	CUGCA
6	8	4	6	CUGCUGCA
7	5	7		CUGCUGCUGCA
8	13	0	4	GCA
9	10	2	5	GCUGCA
10	7	5	6	GCUGCUGCA
11	4	8		GCUGCUGCUGCA
12	2	0	12	UAGCUGCUGCUGCA
13	12	1	5	UGCA
14	9	3	6	UGCUGCA
15	6	6		UGCUGCUGCA
16	16	0		

$(S^{\mathrm{R}}_i)^{-1}$	$\mathsf{aflk_R}[i]$	$\mathsf{lcp_R}[i]$	$\mathsf{suf_R}[i]$	i
AUAGCUGCUGCUGCA	1	0	1	1
AUA		1	13	2
A		1	15	3
AUAGC	8	0	11	4
AUAGCUGC	9	2	8	5
AUAGCUGCUGC	10	5	5	6
AUAGCUGCUGCUGC		8	2	7
AUAG	8	0	12	8
AUAGCUG	9	1	9	9
AUAGCUGCUG	10	4	6	10
AUAGCUGCUGCUG		7	3	11
AU	12	0	14	12
AUAGCU	9	1	10	13
AUAGCUGCU	10	3	7	14
AUAGCUGCUGCU		6	4	15
		0	16	16

Figure 3.2: Affix array for S = AUAGCUGCUGCUGCA. Some lcp-intervals are marked by rectangles and the affix links from an lcp-interval to its reverse interval are represented by arcs. The solid arc points in two directions, from the lcp-interval $q = 5 - [9..11]$ in $\mathsf{suf_F}$ (on the left-hand side) to its reverse interval $q^{-1} = 5 - [5..7]$ in $\mathsf{suf_R}$ (on the right-hand side) and vice versa. That is, $q = (q^{-1})^{-1}$ (see Lemma 2). The dotted arc points in only one direction, from the lcp-interval $q = 4 - [5..7]$ in $\mathsf{suf_F}$ to its reverse interval $q^{-1} = 5 - [5..7]$ in $\mathsf{suf_R}$. In this case, the reverse of q^{-1} is $(q^{-1})^{-1} = 5 - [9..11]$, and $q \neq (q^{-1})^{-1}$.

- A location in the index structure suf_X by interval borders i and j and depth ℓ. For an example, see the yellow marked region in Figure 3.2 which corresponds to the suffix-interval $4 - [5..7]$ in $\mathsf{suf_F}$.
- A (lexicographically ordered) sequence of suffixes $S^X_{\mathsf{suf}_X[i]}, \ldots, S^X_{\mathsf{suf}_X[j]}$. For an example, consider the lexicographically ordered sequence $S^{\mathrm{F}}_{\mathsf{suf_F}[5]} = \mathrm{CUGCA}, \ldots, S^{\mathrm{F}}_{\mathsf{suf_F}[7]} = \mathrm{CUGC}$-UGCUGCA of suffixes in the suffix-interval $4 - [5..7]$ in $\mathsf{suf_F}$ in Figure 3.2.
- A substring of S^X of length ℓ, namely $\delta_X(\ell - [i..j])$. That is, for the suffix-interval $4 - [5..7]$ in $\mathsf{suf_F}$ in Figure 3.2, $\delta_{\mathsf{F}}(4 - [5..7]) = \mathrm{CUGC}$.
- The occurrences of this substring in S^X, namely at positions $\mathsf{suf}_X[i], \ldots, \mathsf{suf}_X[j]$. To give an example, consider Figure 3.2 and observe that substring CUGC occurs at positions $\mathsf{suf_F}[5] = 11$, $\mathsf{suf_F}[6] = 8$, and $\mathsf{suf_F}[7] = 5$ in S^{F} = AUAGCUGCUGCUGCA.

For unidirectional left-to-right search of some pattern in S it is sufficient to process lcp-intervals only in $\mathsf{suf_F}$. For bidirectional pattern search using affix arrays, described in detail in the next section, we employ information from table $\mathsf{suf_F}$ as well as $\mathsf{suf_R}$. Therefore, we need to associate information of one table to the other. This is done by linking intervals via tables $\mathsf{aflk_F}$ and $\mathsf{aflk_R}$. We observe that there exists a mapping between lcp-intervals in $\mathsf{suf_F}$ and $\mathsf{suf_R}$. This is stated by the following proven lemma [101].

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

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

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

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

$$\mathsf{home}_X([i..j]) = \begin{cases} i, \text{ if } \mathsf{lcp}_X[i] \geq \mathsf{lcp}_X[j+1], \\ j, \text{ otherwise}, \end{cases} \tag{3.1}$$

where $\ell - [i..j]$ is an lcp-interval in suf_X. Hence, the home position is one of two boundary positions. Strothmann [101] shows that $\mathsf{home}_X([i..j]) \neq \mathsf{home}_X([i'..j'])$ for different lcp-intervals $\ell - [i..j]$ and $\ell' - [i'..j']$.

Table aflk_X of string $S^X\$$ with total length $n+1$ can now be defined as a table in the range 1 to $n+1$ such that $\mathsf{aflk}_X[\mathsf{home}_X(q)] = i'$, where q is an lcp-interval in suf_X and i' is the left border of the reverse interval $q^{-1} = [i'..j']$ in $\mathsf{suf}_{\overline{X}}$. We refer to the entries in table aflk_X as *affix links*. Tables $\mathsf{aflk}_{\mathsf{F}}$ and $\mathsf{aflk}_{\mathsf{R}}$ occupy $4n$ bytes each. They can be computed by traversing the lcp-intervals in suf_X while simultaneously looking for the corresponding reverse lcp-intervals in $\mathsf{suf}_{\overline{X}}$. Locating reverse lcp-intervals can be accelerated by skp-tables. These tables, introduced in Beckstette *et al.* [54] and hereinafter referred to as $\mathsf{skp}_{\mathsf{F}}$ and $\mathsf{skp}_{\mathsf{R}}$, can be constructed in linear time [112] and allow one to quickly skip intervals in suf_X (for details, see [54]). The construction of tables $\mathsf{aflk}_{\mathsf{F}}$ and $\mathsf{aflk}_{\mathsf{R}}$ takes $\mathcal{O}\left(n^2\right)$ time. Although the use of skp-tables requires additional $2 \times 4n$ bytes of memory, they considerably reduce the construction times of tables $\mathsf{aflk}_{\mathsf{R}}$ and $\mathsf{aflk}_{\mathsf{R}}$ in practice. We note that Strothmann [101] describes a linear time construction algorithm for tables $\mathsf{aflk}_{\mathsf{F}}$ and $\mathsf{aflk}_{\mathsf{R}}$, which employs suffix link and child-tables [109] and an additional table. Altogether these tables require at least additional $7n$ bytes of space. Moreover, even without applying the skp-table based acceleration, Strothmann states that the quadratic time construction algorithm is fast in practice. An example of the affix array for sequence S = AUAGCUGCUGCUGCA highlighted with some of its lcp-intervals connected to the respective reverse interval via the aflk_X table is shown in Figure 3.2.

Because affix links in table aflk_X are only defined for lcp-intervals but not suffix-intervals in general, which we require in bidirectional search, we introduce the concept of *affix-intervals*. Affix-intervals are similar to affix nodes as defined in [101]. An affix-interval in suf_X is a triple $v = \langle k, q, X \rangle$, where k is an integer designated *context* of v and q is a suffix-interval in suf_X.

An affix-interval $v = \langle k, q, X\rangle$ in suf_X, with $q = \ell - [i..j]$, $\ell > 0$, $-m < k < \ell$, describes a substring $\omega_X(v)$ of S^X of length $\ell - k$, defined as the k-th suffix of $\delta_X(q)$, i.e. $\omega_X(v) = S^X[\mathsf{suf}_X[i] + k..\mathsf{suf}_X[i] + \ell - 1]$. At the same time v identifies all occurrences of $\omega_X(v)$ in S^X, namely the positions $\mathsf{suf}_X[i] + k, \ldots, \mathsf{suf}_X[j] + k$. For $v = \langle k, q, X\rangle$, we therefore also use the notation $\overrightarrow{v} = \omega_F(v)$ if $X = \mathsf{F}$ and $\overrightarrow{v} = \omega_R(v)^{-1}$ if $X = \mathsf{R}$. As an example, consider the affix-interval $v = \langle 1, 4 - [5..7], \mathsf{F}\rangle$ in suf_F of the affix array shown in Figure 3.2. In this case, $k = 1$, $q = 4 - [5..7]$, and $X = \mathsf{F}$. v identifies all occurrences of substring $\overrightarrow{v}$ = UGC in S^F at positions $\mathsf{suf}_\mathsf{F}[5] + 1 = 12$, $\mathsf{suf}_\mathsf{F}[6] + 1 = 9$, and $\mathsf{suf}_\mathsf{F}[7] + 1 = 6$. Observe that $\overrightarrow{v}$ = UGC is the first suffix of $\delta_\mathsf{F}(q)$ = CUGC due to context $k = 1$.

3.4 Searching RNA databases with affix arrays

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

3.4.1 Unidirectional traversal of affix arrays

Let $P = P[1] \ldots P[m] \in (\mathcal{A} \cup \Phi)^m$ be a sequence pattern to be searched in S in a unidirectional left-to-right way using information from table suf_F only. To search for P, we call the procedure *unidir-search* of Figure 3.3 by *unidir-search*$([1..|S|+1], P, 1)$. Therefore, in step 0 we start searching for the characters in $\varphi(P[1])$ in the suffix-interval $q_0 = 0 - [1..n+1]$ in suf_F, which represents all suffixes of $S\$$. In each step k, $k \geq 0$, we locate the $k+1$-suffix-intervals q_k of maximal width, such that $P[1..k+1]$ matches $\delta_\mathsf{F}(q_k)$. For each $d \in \varphi(P[k+1])$, this step is performed by two binary searches in the suffix-interval $q_{k-1} = \ell - [i..j]$ for $q_k = (\ell + 1) - [i'..j']$, $i \leq i' \leq j' \leq j$, $j' - i'$ maximal, and $S[\mathsf{suf}_\mathsf{F}[i'] + k + 1] = d$. With a binary search we locate i' and with another we locate j'.

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

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

```
1 if k = |P| + 1 then
2 |   report match at positions suf_F[i], ..., suf_F[j]
3 |   return
4 else
5 |   foreach q' such that d ∈ φ(P[k]) and δ_F((k+1) − [i..j]) = δ_F(k − [i..j])d do
6 |   |   unidir-search(q', P, k + 1)
7 |   end
8 end
```

Figure 3.3: Unidirectional search algorithm for searching for a sequence pattern $P \in (\mathcal{A} \cup \Phi)^*$. Given the suffix array suf_F of S, the procedure enumerates all occurrences of P in S when called by *unidir-search*$([1..|S|+1], P, 1)$. In line 5, the suffix-interval q' is located by binary search in $\mathcal{O}(\log n)$.

3.4.2 Bidirectional traversal of affix arrays

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

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

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

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

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

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

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

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

The following cases require switching the search direction.

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

The operation fails if v_x cannot be determined.

3.4.3 RNA sequence-structure pattern matching using affix arrays

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

For further considerations, we will assume a special 'canonical' form for RSSPs, which we define in the following. Independently of a sequence S, each RSSP describes a set of pattern instances, i.e. the set of potential substrings matching the pattern. Often, there are several patterns that describe the same set of instances. For example, the pattern (UNUACACGNR, (((. . . .)))) describes the same set of instances as (UNUACACGNR, ((. . . . . .))) since the additional base pair $(3, 8)$ in (((. . . .))) does not make the pattern more specific. We will define a pattern to be structure minimal if there is no, in this sense, equivalent pattern containing a true subset of the base pairs. An RSSP $\mathcal{Q} = (P, R)$ is *structure minimal* if and only if for all base pairs $(i, j) \in \widehat{R}$ it holds that

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

Furthermore, a general pattern is called *inconsistent* if it does not have any instance. Formally, a pattern is *consistent* if and only if for each base pair (i, j) it holds that $\varphi(P[i]) \cap \mathcal{CS}(P[j]) \neq \emptyset$ and $\varphi(P[j]) \cap \mathcal{CS}(P[i]) \neq \emptyset$. An example of an inconsistent RSSP is $\mathcal{Q} = (P, R)$ with P = UAUACACGAN and $R =$ ((. . . . . .)). $\mathcal{Q}$ is not consistent because there is a base pair $(2, 9) \in \widehat{R}$ but the bases $P[2] =$ A and $P[9] =$ A are not complementary, i.e. $(A, A) \notin \mathcal{C}$. An example of a structure minimal and consistent RSSP is (UNUACACGNR, ((. . . . . .))). Note that a pattern can be transformed into an equivalent structure minimal pattern and checked for consistency in $\mathcal{O}(m)$ time. For complexity considerations, we can therefore safely assume that patterns are consistent and structure minimal.

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

Following this idea, Mauri and Pavesi [102] presented an algorithm for matching RNA stem-loop structures using affix trees. This algorithm explores the search space in a breadth-first manner, so memory use grows exponentially with increasing depth. Instead of an affix tree, we employ the more space efficient affix array data structure and use a depth-first search algorithm which only

requires space for the search proportional to the length of the substring searched. The depth-first search for all occurrences of a stem-loop RSSP $\mathcal{Q} = (P, R)$ is performed by calling procedure *bidir-search* of Algorithm 2 (see Figure 3.4). Note that we explicitly support bulges and internal loops in the stem-loop pattern, i.e. we do not require perfect stacking of the base pairs but allow general non-branching structures.

In our algorithm, we switch the search direction only once per base pair when matching the stem region of the pattern, thus halving the number of lookups in the affix link tables compared to a naive algorithm without this optimization. This was also observed by Strothmann [101] whose algorithm did not support RSSPs containing bulges and internal loops.

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

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

3.4.4 An example of bidirectional RNA sequence-structure pattern search

As an example of bidirectional search for RSSPs using affix arrays, we search for the RSSP $\mathcal{Q}$ in the sequence S given in Figures 3.1 and 3.2, respectively. We recall that $\mathcal{Q} = (P, R)$ with P = NNNUGCUNNN and R = (((. . . .))) represents a stem-loop structure of length $m = 10$ and S = AUAGCUGCUGCUGCA has length 15. We start matching P in S by calling procedure *bidir-search* of Algorithm 2 as *bidir-search*$(\langle 0, 0-[1..16], \mathsf{F}\rangle, 3, 4)$. That is, the algorithm matches the first position $P[4] = \mathrm{U}$ of the loop region in left-to-right direction. Given that $\mathsf{X} = \mathsf{F}$ and $i < j$ (i.e. $1 < 16$) hold, it locates interval $v_x = \langle 0, 1-[12..15], \mathsf{F}\rangle$ with $\overrightarrow{v_x} = \mathrm{U}$ via binary search in the interval $0-[1..16]$ of suf_F. Analogously, the following recursive calls of *bidir-search* perform right c-extensions of $u = \mathrm{U} = P[4..4]$ with characters $P[5] = \mathrm{G}$, $P[6] = \mathrm{C}$, and $P[7] = \mathrm{U}$, by searching in the intervals $1-[12..15]$, $2-[13..15]$, and $3-[13..15]$, respectively. After these extensions, the algorithm has located the affix-interval $v_x = \langle 0, 4-[14..15], \mathsf{F}\rangle$ representing all occurrences of $\overrightarrow{v_x} = \mathrm{UGCU}$ in S such that $\overrightarrow{v_x}$ matches $u = P[4..7]$. We set $v = v_x$. Next, the algorithm performs a right c-extension of u with the pairing position $c \in \varphi(P[8] = \mathrm{N})$. Therefore, it enumerates all

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

```
1  if r < 1 and r' > m then
2  |  report match at positions suf_X[i] + k, ..., suf_X[j] + k
3  |  return
4  else if r ≥ 1 and r' ≤ m and R[r] = '(' and R[r'] = ')' then
5  |  if X = R then
   |  |  // perform left extension first
6  |  |  foreach v' such that d ∈ φ(P[r]) and →v' = d→v do
7  |  |  |  foreach v'' such that e ∈ φ(P[r']) and (d,e) complementary and →v'' = →v'e do
8  |  |  |  |  bidir-search(v'', r − 1, r' + )
9  |  |  |  end
10 |  |  end
11 |  else
   |  |  // perform right extension first
12 |  |  foreach v' such that e ∈ φ(P[r']) and →v' = →ve do
13 |  |  |  foreach v'' such that d ∈ φ(P[r]) and (d,e) complementary and →v'' = d→v' do
14 |  |  |  |  bidir-search(v'', r − 1, r' + 1)
15 |  |  |  end
16 |  |  end
17 |  end
18 else if r' ≤ m and R[r'] = '.' and (X = F or r < 1 or R[r] ≠ '.') then
19 |  foreach v' such that d ∈ φ(P[r']) and →v' = →vd do
20 |  |  bidir-search(v', r, r' + 1)
21 |  end
22 else if r ≥ 1 and R[r] = '.' then
23 |  foreach v' such that d ∈ φ(P[r]) and →v' = d→v do
24 |  |  bidir-search(v', r − 1, r')
25 |  end
26 end
```

Figure 3.4: Bidirectional recursive RSSP matching using an affix array. Procedure *bidir-search* finds all matches of a given RSSP (P, R), beginning the pattern extensions from any position in the loop region or any position in a completely unpaired pattern. In each call, parameter v denotes the affix-interval representing matches of the pattern substring $P[r+1..r'-1]$, $1 \leq r \leq r' \leq m$ satisfying the structural constraints imposed by $R[r+1..r'-1]$. The procedure takes care to change the search direction only as often as necessary, in particular it changes the direction only once per base pair.

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

3.4.5 Complexity analysis

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

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

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

Algorithm 1 accomplishes the search for an unambiguous pattern $\mathcal{Q}$ on the suffix array suf_F using binary search for locating intervals in $\mathcal{O}(m \log n + z)$ time, where z is the number of occurrences of P in S. We remark that this time bound can be lowered at the price of higher memory consumption to $\mathcal{O}(m + \log n + z)$ [52] or even $\mathcal{O}(m + z)$ [113, 109] time by using additional precomputed information.

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

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

On each path from the root to the leaf the algorithm performs $m - 2p$ c-extensions and at most one switch of the search direction for matching the $m - 2p$ unpaired characters. Then, it performs $2p$

c-extensions and p switches of the direction for matching the base paired positions. Therefore, we count the total number of c-extensions as

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

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

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

Instead of performing the binary search over the suffix tables, one can use the child-tables introduced by Abouelhoda *et al.* in [109] to determine the child intervals and switch the search direction in constant time. The child-tables, however, add at least $2n$ bytes to the index and require additional involved index construction. As the child-tables improve the worst case behavior but, on the other hand, require more space, we analyze the complexity with and without these tables (i.e. with tables suf_X, lcp_X, and aflk_X only).

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

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

2. Case $i < j$ and $k = 0$. The algorithm needs to locate the reverse affix-interval $v' = \langle k', \ell' - [i'..j'], \overline{X} \rangle$ of v. Interval boundaries $i' = \mathsf{aflk}_X[\mathsf{home}_X([i..j])]$ and $j' = i' + (j - i)$ of v' are computed in constant time. By definition, computing the reverse affix-interval of v requires knowing ℓ_{lcp}. Then, $\ell' = \ell_{\text{lcp}}$ and $k' = \ell' - \ell$. Without child-tables, we determine ℓ_{lcp} by computing the length of the longest common prefix between $S^X_{\mathsf{suf}_X[i]}$ and $S^X_{\mathsf{suf}_X[j]}$. It suffices to perform $\ell_{\text{lcp}} - \ell + 1 = k' + 1$ character comparisons only, since both suffixes $S^X_{\mathsf{suf}_X[i]}$ and $S^X_{\mathsf{suf}_X[j]}$ share a common prefix of at least length ℓ. With the help of child-tables, ℓ_{lcp} is determined in constant time [109].

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

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

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

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

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

For the worst case that all strings matching the pattern actually occur as substrings in S, the sequence S must have a certain minimal length. In the case of $p = 0$, the possible matches are the words in $\mathcal{A}^m$ and a sequence that contains all these matches is called $|\mathcal{A}|$-ary *de Bruijn* sequence of order m [114] without wrap-around, i.e. a *de Bruijn* sequence with its first $m - 1$ characters concatenated to its end. Such a sequence was shown to have a length of $n_0 = |\mathcal{A}|^m + m - 1$. As a consequence, the worst case requires $n \geq n_0$.

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

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

However, it can be clearly seen from this analysis that the worst case is based on extremely pessimistic assumptions that are almost contrary to the expected application. 1.) It is assumed that a pattern consists of wildcards N only. In the expected application, however, patterns will often specify bases in the loop region, which is of particular benefit for our algorithm. 2.) Sequences, like

the *de Bruijn* sequence, that contain all possible matches of an average sized pattern will be rare in practice. E.g. it could be assumed that a sequence that contains all possible matches of a pattern Q with p base pairs (and $P =$N... N) is at least as long as the $|\mathcal{A}|$-ary *de Bruijn* sequence of order m, since one expects no significant bias for the specific complementarity due to $\widehat{R}$ over all substrings of length m. However, $E_{m,p} = |\mathcal{A}|^{m-p}|\mathcal{C}|^p = 4^{m-2p}6^p = 4^m/(16/6)^p$ is even for small p much smaller than $n_0 = 4^m + m - 1$. For example, four base pairs (i.e., $p = 4$) reduce the time bound by a factor of $(16/6)^4 \approx 50$ and eight base pairs reduce time by a factor of about 2500.

3.4.6 A bidirectional search algorithm supporting variable length RSSPs

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

Before describing the algorithm, we define this extension of RSSPs. A *variable-length RSSP* $\mathcal{Q}$ consists of an RSSP (P, R) and parameters *maxleftloopextent (mllex)*, *maxrightloopextent (mrlex)*, and *maxstemlength (msl)*. *mllex* and *mrlex* denote the maximum number of respective left and right extensions of the loop region specified in R and msl denotes the maximum number of base pairs in the stem. The minimum length of occurrences of $\mathcal{Q}$ is $m = |P| = |R|$. For examples of variable-length RSSPs, see Figure 3.5 (E) until (H).

To keep the code simple, we split the original algorithm into two procedures. (i) First the loop region of a given variable-length RSSP $\mathcal{Q}$ is matched with procedure *bidir-search-loop* (see Algorithm 3, Figure 3.6). (ii) Next, the stem region is matched with procedure *bidir-search-stem* (see Algorithm 4, Figure 3.7). Note that *bidir-search-stem* is very similar to Algorithm 2. Prior to the search for $\mathcal{Q}$, the following variables are set: *loopstart*, *minloopstart*, *loopend*, *maxloopend*, *minbps*, and *maxbps*. These variables store the following information. *loopstart* (*loopend*) stores the position of the base occurring in the left-most (right-most) position of the loop described by the structure string R in 5' to 3' direction, *minloopstart* $=$ *loopstart* $-$ *mllex*, *maxloopend* $=$ *loopend* $+$ *mrlex*, and *minbps* (*maxbps* $=$ *msl*) is the minimum (maximum) number of base pairs occurring in $\mathcal{Q}$. It holds: *minloopstart* $\leq$ *loopstart* $\leq$ *loopend* $\leq$ *maxloopend*. Note that *minloopstart* can be negative. As an example, let $R =$ (((. . . .))), *mllex* $= 5$, and *mrlex* $= 1$. Then *loopstart* $= 4$, *minloopstart* $= -1$, *loopend* $= 7$, *maxloopend* $= 8$, and *minbps* $= 3$. To match $\mathcal{Q}$, procedure *bidir-search-loop* is initially called as *bidir-search-loop*$(\langle 0, 0 - [1..n + 1], \mathsf{F}\rangle, r_0 - 1, r_0,$ true$)$, where $\langle 0, 0 - [1..n + 1], \mathsf{F}\rangle$ is an affix-interval, r_0 is any position in the loop region of $\mathcal{Q}$, and parameter true states that the pattern can be extended to the right. Procedure *bidir-search-loop* calls *bidir-search-stem* whenever substrings of minimum length *loopend* $-$ *loopstart* $+ 1$ matching the loop in the searched database are found. If $\mathcal{Q}$ has no base pairs, i.e. $msl = 0$, it instead immediately reports the matching

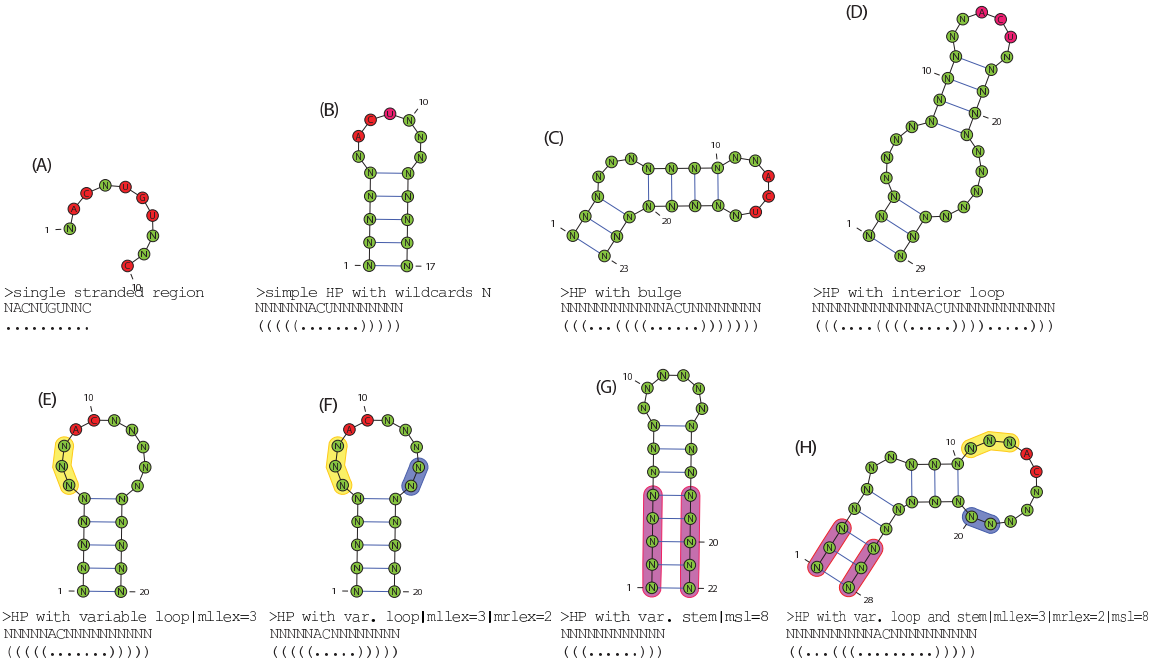


Figure 3.5: Supported structural patterns and corresponding pattern definitions in *Structator* syntax (see complete syntax description in Appendix A). Unambiguous nucleotides are marked in red. Positions containing ambiguous nucleotides, denoted here with character N, are marked in green and can contain any nucleotide from $\mathcal{A}$. Maximal allowed left and right extensions of the loop region of a pattern as specified by parameters *maxleftloopextent (mllex)* and *maxrightloopextent (mrlex)* are marked in yellow and blue, respectively. Allowed possible extensions of a pattern's stem region as specified by parameter *maxstemlength (msl)* are marked in purple. As an example for the semantics of the parameter *msl* consider pattern (G): it matches all substrings of the searched sequence that are able to fold into a stem-loop structure with loop length 6 and stem length between 3 and 8. For further details, see corresponding text.

positions. This is reflected by calling *bidir-search-stem*$(v', loopstart-1, loopend+1, 0)$, where v' is the affix-interval representing all occurrences of substring $\overrightarrow{v}'$ in the searched database matching the loop region of $\mathcal{Q}$, positions *loopstart* $-$ 1 and *loopend* $+$ 1 denote the inner-most base pair (*loopstart* $-$ 1, *loopend* $+$ 1) of the pattern, and 0 is the number of currently matched base pairs. Procedure *bidir-search-stem* reports matching positions of $\mathcal{Q}$ whenever the boundaries of the RSSP are reached or $minbps \leq bpcount \leq maxbps$ holds.

3.5 RNA secondary structure descriptors based on multiple ordered RSSPs

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

We address search for complex structured RNA families with the new concept of RNA secondary structure descriptors (SSD for short). SSDs use the information of multiple ordered RSSPs derived from the decomposition of an RNA's secondary structure or from the consensus secondary structure of a multiple sequence-structure alignment of related RNAs into stem-loop-like structural elements. Such consensus secondary structures for multiple RNAs can be computed with a variety of programs following one of the three strategies introduced in [63]. Namely: (A) alignment of the sequences followed by joint folding [115, 116, 117, 118], (B) Sankoff style [64] simultaneous alignment and folding [119, 44, 120, 121], and (C) individual folding of the sequences followed by alignment of their structures [69, 122, 123]. In the following we make the concept of SSDs more precise. Let $A = A_1, A_2, \ldots, A_L$ be a sequence of non-overlapping alignment blocks. These alignment blocks are excised from a multiple sequence(-structure) alignment and represent regions of the molecule that fold into stem-loop-like structures or remain unfolded. The indexing from 1 to L reflects their order of occurrence in the alignment. Hence A represents a sequential decomposition of the molecule's secondary structure (in $5' \rightarrow 3'$ direction) into regions, each of which can be described by an RSSP. See Figure 3.8 (A) for an example.

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

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

Figure 3.6: Bidirectional recursive matching of the loop region of a variable-length RSSP using an affix array. Procedure *bidir-search-loop* searches for an RSSP (P, R) defined with additional variables *maxleftloopextent (mllex)* and *maxrightloopextent (mrlex)* denoting the maximum number of left and right extensions of the loop specified in R, respectively, and *maxstemlength (msl)* denoting the maximum number of base pairs. Used variables *loopstart*, *minloopstart*, *loopend*, and *maxloopend* are preset according to structure string R, *mllex*, and *mrlex* (see text). *bidir-search-loop* calls procedure *bidir-search-stem* (see Algorithm 4) whenever substrings of minimum length *loopend* − *loopstart* + 1 matching the loop are found.

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

```
1  if (r < 1 and r' > m) or (minbps ≤ bpcount ≤ maxbps) then
2  |  report match at positions sufX[i] + k, ..., sufX[j] + k
3  end
4  if (minbps ≤ bpcount < maxbps) or (r ≥ 1 and r' ≤ m and R[r] = '(' and R[r'] = ')')
   then
5  |  if minbps ≤ bpcount < maxbps then
6  |  |  chr  = 'N'
7  |  |  chr' = 'N'
8  |  else
9  |  |  chr  = P[r]
10 |  |  chr' = P[r']
11 |  end
12 |  if X = R then
   |  |  // perform left extension first
13 |  |  foreach v' such that d ∈ φ(chr) and v'→ = d v→ do
14 |  |  |  foreach v'' such that e ∈ φ(chr') and (d, e) complementary and v''→ = v'→ e do
15 |  |  |  |  bidir-search-stem(v'', r − 1, r' + 1, bpcount + 1)
16 |  |  |  end
17 |  |  end
18 |  else
   |  |  // perform right extension first
19 |  |  foreach v' such that e ∈ φ(chr') and v'→ = v→ e do
20 |  |  |  foreach v'' such that d ∈ φ(chr) and (d, e) complementary and v''→ = d v'→ do
21 |  |  |  |  bidir-search-stem(v'', r − 1, r' + 1, bpcount + 1)
22 |  |  |  end
23 |  |  end
24 |  end
25 else if r' ≤ m and R[r'] = '.' and (X = F or r < 1 or R[r] ≠ '.') then
26 |  foreach v' such that d ∈ φ(P[r']) and v'→ = v→ d do
27 |  |  bidir-search-stem(v', r, r' + 1, bpcount)
28 |  end
29 else if r ≥ 1 and R[r] = '.' then
30 |  foreach v' such that d ∈ φ(P[r]) and v'→ = d v→ do
31 |  |  bidir-search-stem(v', r − 1, r', bpcount)
32 |  end
33 end
```

Figure 3.7: Bidirectional recursive matching of the stem region of a variable-length RSSP using an affix array. Procedure *bidir-search-stem* is called by procedure *bidir-search-loop* (see Algorithm 3) and extends substrings $\overrightarrow{v}$ matching the loop region of the RSSP (P, R) to substrings matching also the stem. Used variables *minbps* and *maxbps* are preset according to structure string R and variable *maxstemlength (msl)* (see text).

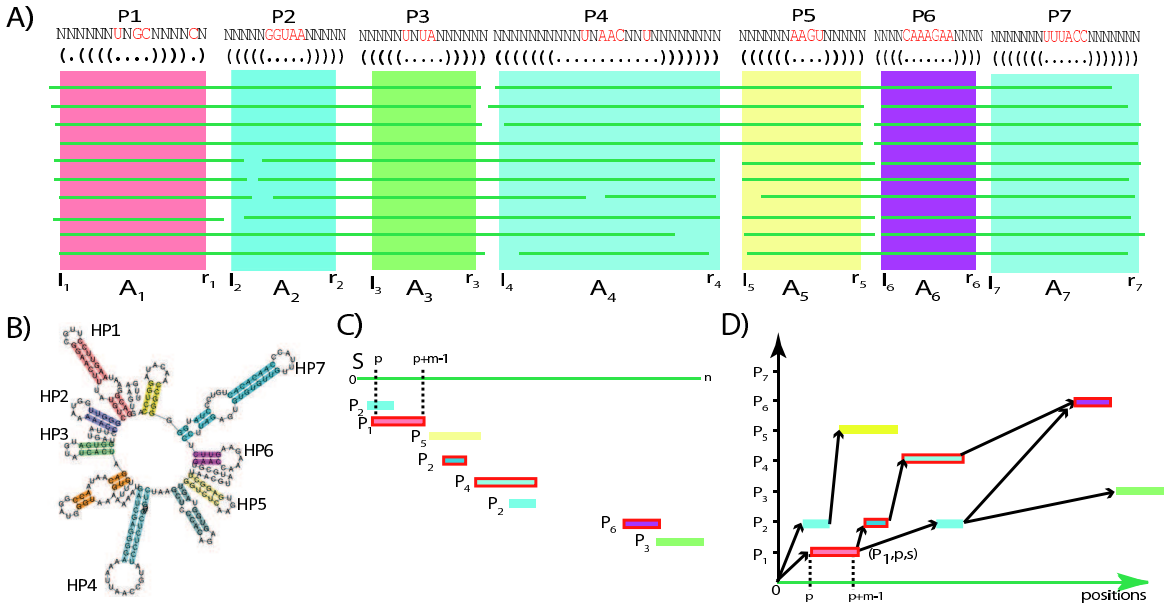


Figure 3.8: (A) Non-overlapping alignment blocks of stem-loop regions excised from a multiple sequence-structure alignment and derived sequence-structure patterns. Since $l_i \leq r_i < l_j \leq r_j$ and sequence regions $S[l_i \ldots r_i]$ fold into stem-loop structures for $1 \leq i \leq j \leq 7$, $A = A_1, A_2, A_3, A_4, A_5, A_6, A_7$ is an ordered sequence of non-overlapping alignment blocks suitable to construct an RNA secondary structure descriptor $\mathcal{R} = \mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4, \mathcal{Q}_5, \mathcal{Q}_6, \mathcal{Q}_7$. The sequence-structure patterns $\mathcal{Q}_i, i \in [1, 7]$ of $\mathcal{R}$ given on top of their underlying alignment blocks describe the seven marked stem-loops shown in the RNA secondary structure (B) of the Citrus tristeza virus replication signal (Rfam: RF00193). (C) Matches of RSSPs $\mathcal{Q}_i, i \in [1, 7]$, on sequence S, sorted in ascending order of their start position. (D) Graph-based representation of the matches of $\mathcal{Q}_i, i \in [1, 7]$. An optimal chain of collinear non-overlapping matches is determined by computing an optimal path in the directed acyclic graph. Observe that not all edges in the graph are shown in this example and that the optimal chain (indicated here by their red marked members) is not necessarily the longest possible chain.

An SSD $\mathcal{R}$ of length L is a sequence of L RSSPs $\mathcal{R} = \mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_L$ where $\mathcal{Q}_i$ denotes the RSSP describing A_i, $i \in [1, L]$. The order $\ll$ of the RSSPs in $\mathcal{R}$ is imposed by the order of the corresponding alignment blocks. By l_i and r_i we denote the start and end positions of A_i in the multiple alignment, respectively. In practice, $\mathcal{R}$ can be obtained from multiple sequence-structure alignments of related RNA sequences (i.e., of an RNA family) as they are available in databases like Rfam [124, 125]. A match to $\mathcal{R}$ is a non-overlapping sequence of matches for some or all of the RSSPs in $\mathcal{R}$ in their specified order.

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

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

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

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

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

$$gcsc\,(\mathcal{C}) = \sum_{i=1}^{k} \alpha(\mathcal{Q}_{j_i}). \tag{3.2}$$

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

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

$$lcsc\,(\mathcal{C}) = \sum_{i=1}^{k-1} (\alpha(\mathcal{Q}_{j_i}) - g\left((\mathcal{Q}_{j_i}, p_i), (\mathcal{Q}_{j_{i+1}}, p_{i+1})\right)) + \alpha(\mathcal{Q}_{j_k}) \tag{3.3}$$

where

$$g\left((\mathcal{Q}_{j_i}, p_i), (\mathcal{Q}_{j_{i+1}}, p_{i+1})\right) = \left|(p_{i+1} - p_i) - (l_{j_{i+1}} - r_{j_i})\right|. \tag{3.4}$$

To solve the local chaining problem we use our own implementation of a fast local chaining algorithm described in [126] with modified gap costs. While the algorithm of [126] penalizes gaps by

the sum of their lengths, our solution is based on the difference between their observed lengths (in the chain of matches) and their expected lengths (as given by the multiple alignment of the family); confer Equation (3.4). This algorithm runs in $\mathcal{O}(q \log q)$ time, where q is the size of $\mathcal{MS}$.

To solve the global chaining problem we use an efficient chaining algorithm running in $\mathcal{O}(q)$ time. This algorithm is described in [104].

3.6 Implementation and computational results

We implemented (1) the algorithms necessary for affix array construction, (2) the fast bidirectional search of RSSPs using affix arrays as sketched in Algorithm 2 (hereinafter called *BIDsearch*), (3) an online variant operating on the plain sequence (hereinafter called *ONLsearch*) for validation of *BIDsearch* and reference benchmarking, and (4) integrated with the search algorithms the efficient global and local chaining algorithms. Algorithm *ONLsearch* shifts a window of length $m = |RSSP|$ along the sequence of length n to be searched and compares the substring inside the window with the RSSP from left to right until a mismatch occurs. Hence, it runs in $\mathcal{O}(nm)$ time in the worst and $\mathcal{O}(n)$ time in the best case. Algorithms *BIDsearch* and *ONLsearch* were implemented in the program *afsearch*. The *afconstruct* program makes use of routines from the *libdivsufsort2* library (see `http://code.google.com/p/libdivsufsort/`) for computing the suf_F and suf_R tables in $\mathcal{O}(n \log n)$ time. For the construction of the lcp_F and lcp_R tables we employ our own implementation of the linear time algorithm of [111]. Tables aflk_F and aflk_R are constructed in $\mathcal{O}(n^2)$ worst-case time with fast practical construction time due to the use of the skip tables skp_F and skp_R [54]. The programs were compiled with the GNU C compiler (version 4.3.2, optimization option -O3) and all measurements were performed on a Quad Core Xeon E5410 CPU running at 2.33 GHz, with 64 GB main memory (using only one CPU core). To minimize the influence of disk subsystem performance the reported running times are user times averaged over 10 runs. Allowed base pairs were canonical Watson-Crick (A, U), (U, A), (C, G), (G, C), and wobble (G, U), (U, G), unless stated otherwise.

Affix array construction times

In a first experiment we constructed the affix array for genomes of selected model organisms of different sizes and stored it on disk. We measured the total running times needed by *afconstruct* to construct each table comprising the affix array. See Figure 3.9 for the results of this experiment. The total size for each table is given in Table 3.1. Construction times were in the range of 25 minutes for the *C.elegans* genome containing ~ 100 megabases to 15.7 hours for the ~ 2 gigabase genome of the megabat *P.vampyrus*.

We also measured the running time of *afconstruct* to construct the affix array for a set of $3{,}192{,}599$ RNA sequences with a total length of ~ 622 MB compiled from the full alignments of all Rfam

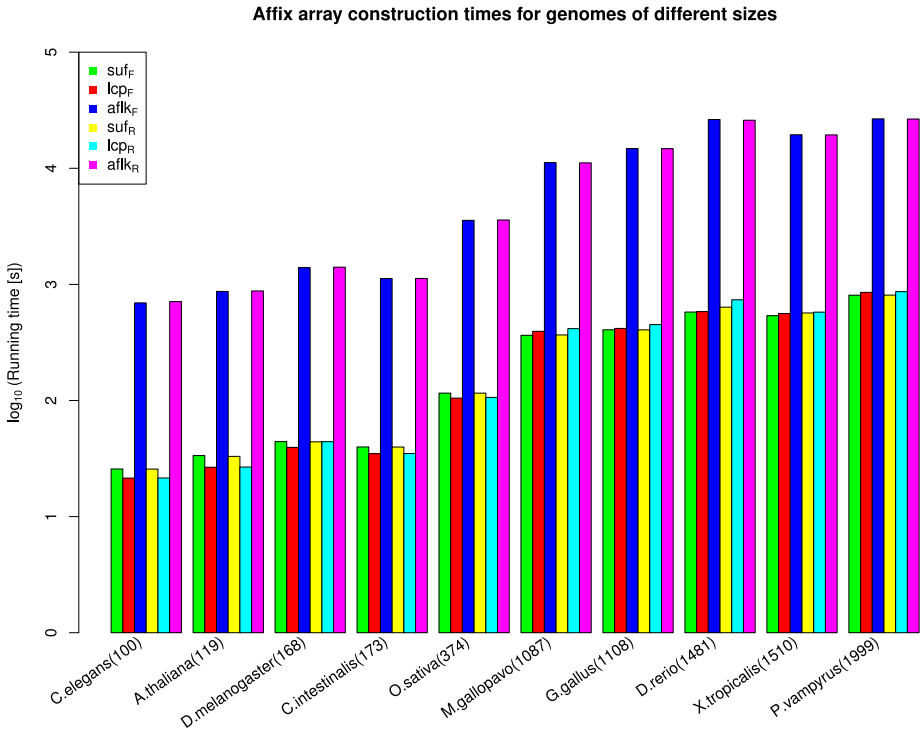


Figure 3.9: Experiment 1: Running times for affix array construction for genomes of different model organisms. Genome sizes are given for each organism in megabases in brackets. We measured the running time in seconds for all tables the affix array consists of (y-axis, $\log_{10}$ scale). Total construction times were in the range of ~ 25 minutes for *C.elegans* up to 15.7 hours for *P.vampyrus*.

release 10.0 families. The construction and storage on disk required 126 minutes. In the following we refer to this dataset as RFAM10 for short.

Influence of loop length on search performance

In a second experiment we investigated the influence of the loop length and the number of unambiguous characters in the loop of an RSSP on the running time of *BIDsearch* and *ONLsearch*. For this experiment we constructed artificial RSSPs with a fixed stem length of 7 and a loop length l varying from 3 to 20. For each loop length, we also varied the number of consecutive unambiguous characters q from 0 to 4. For $q = 0$ this means that the RSSP contains structural constraints only. That is, for $q = 0$ and $l = 5$ the used RSSP matches all substrings that are able to fold into a stem-loop structure with loop length 5 and stem length 7. Such a pattern is written in dot-bracket notation as `((((((.....))))))`. Allowed base pairs were (A, U), (U, A), (C, G), and (G, C). We measured the time needed by *BIDsearch* and *ONLsearch* to search for these patterns in the RFAM10 dataset. Results are given in Figure 3.10. In this experiment *BIDsearch* performed very well and was faster than *ONLsearch* for all parameter combinations. We also investigated the

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

Table 3.1: Sizes in megabytes of the different tables the affix array consists of for the genomes used in Experiment 1. $\mathsf{lcpe_F}$ and $\mathsf{lcpe_R}$ are the exception tables storing entries with value larger than 255 which cannot be stored in tables $\mathsf{lcp_F}$ and $\mathsf{lcp_R}$, respectively. In tables $\mathsf{lcpe_F}$ and $\mathsf{lcpe_R}$, each entry consumes 8 bytes.

influence of different stem length (data not shown here) and found that the impact on the total running time is negligible. We observe that the advantage of *BIDsearch* over *ONLsearch* decreases with increasing loop length l for fixed q. We explain this behavior with the increasing number of affix-intervals that have to be processed for finding all different substrings of the sequences that match the RSSP. However, even for an RSSP with loop length $l = 20$ containing only structural constraints ($q = 0$), *BIDsearch* is still faster than *ONLsearch*. We further notice that the number of unambiguous characters in the loop region has a strong influence on the running time of *BIDsearch*. That is, by specifying only a few conserved nucleotides in the RSSP's loop region, the running time of *BIDsearch* is reduced dramatically. For an example of this effect, see the running times of *BIDsearch* in Figure 3.10 for parameters $l = 15$ and $q \in \{2, 3, 4\}$. This renders *BIDsearch* in particular useful for searching with RSSPs with moderate loop length or existing sequence conservation in the loop region. The speedup factors measured in this experiment were in the range from 1.001 to 78.1 for $q = 0$ and from 9.28 to 11×10^3 for $q = 4$. Table 3.2 gives more details on the speedups of *BIDsearch* over *ONLsearch* for all investigated combinations of q and l.

Searching large sequence databases

To measure the performance of *BIDsearch* for non-artificial real-world RSSPs, we manually compiled a set of 397 RSSPs describing 42 highly structured RNA families taken from the RFAM10 database. These were all families with a consensus secondary structure containing at least 5 stem-loop substructures. We measured the running time needed by *BIDsearch*, *ONLsearch*, and the widely used tools *RNAMotif* [42] and *RNABOB* [98] to search for these 397 RSSPs in the RFAM10 dataset. As expected, all tools delivered identical results. However, while it took *BIDsearch* less than 50 seconds to search for the 397 patterns as shown in Table 3.3, *RNABOB* and *RNAMotif* needed more

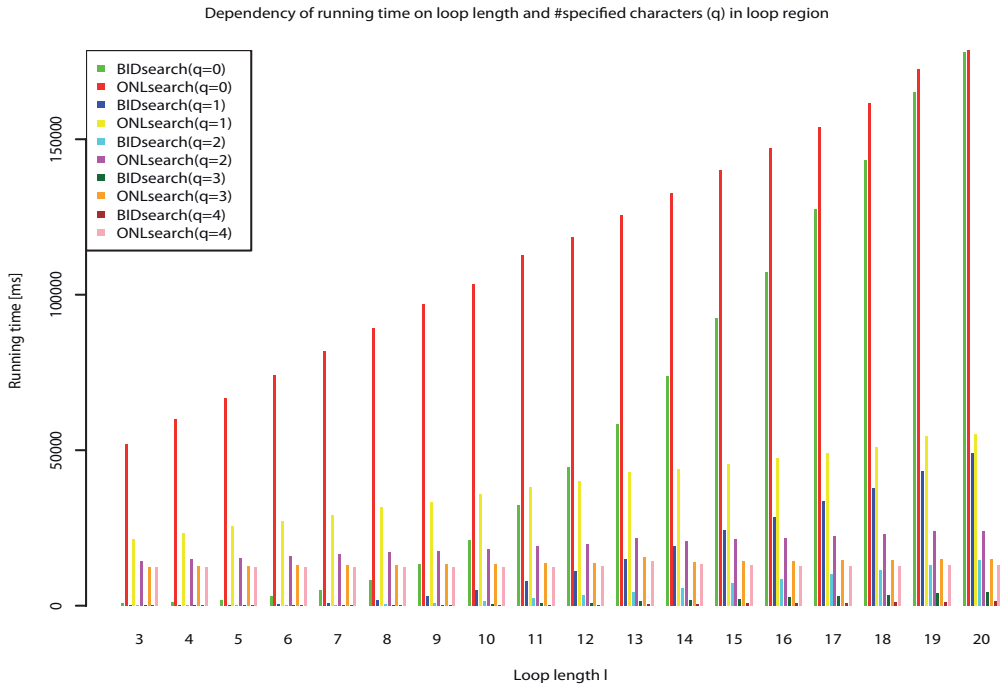


Figure 3.10: Experiment 2: Influence of loop length and number of unambiguous characters in loop region on total running time of *BIDsearch* and *ONLsearch*. We measured the running time in milliseconds to search with artificial RSSPs with loops of varying length $l \in \{3, \ldots, 20\}$ on $\sim$ 622MB of RNA sequence data. For each loop length l we also varied the number $q \in \{0, \ldots, 4\}$ of unambiguous nucleotides in the loop. The used RSSPs had a fixed stem length of 7. For more details on this experiment see corresponding text.

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

Table 3.2: Experiment 2: Obtained speedup of *BIDsearch* over *ONLsearch* for different loop length $l \in \{3, \ldots, 20\}$ and number of unambiguous characters in the loop region $q \in \{0, \ldots, 4\}$. For the parameter combination $l = 3, q = 4$ also one character of the stem was specified.

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

Table 3.3: Experiment 3 (A): Running times in seconds needed by the programs to search for 397 RSSPs describing 42 RFAM10 families in $\sim$ 622 megabases of RNA sequence data. For each program the speedup factor of *BIDsearch* over the particular program is given in brackets.

than 2.5 and 3.2 hours respectively to complete the same task. This made for a speedup factor of 196.5 (254.7) for *BIDsearch* over *RNABOB* (*RNAMotif*). Even if we include the time needed for affix array construction, *BIDsearch* is still faster than *RNABOB* and *RNAMotif*.

We also investigated the distribution of speedup factors obtained by *BIDsearch* when searching for the 397 RSSPs. We observed that *BIDsearch* is more than 50,000 times faster than *RNABOB* and *RNAMotif* for the majority of the patterns and that the total search time required by *BIDsearch* is dominated by only a small number of patterns. These patterns describe large unconserved loop regions. See Figure 3.11 for a graphical visualization of the distribution of speedup factors.

Scaling behavior of bidirectional pattern search using affix arrays

In a further experiment we investigated the scaling behavior of *BIDsearch* and *ONLsearch* for an increasing size of sequences to be searched. For this, we searched with different RSSPs on random subsets of RFAM10 of different sizes and measured the running time for both algorithms. The results are given in Figure 3.12. Here pattern1 is an RSSP containing only structural constraints. It describes a stem-loop with loop length 4, stem length 10 and no specified nucleotides in the loop region. The RSSP pattern2 (pattern3) only differ from pattern1 by containing one (two consecutively) unambiguous nucleotides in the loop region.

In this experiment *BIDsearch* clearly showed a sublinear scaling behavior, whereas *ONLsearch* scaled only linearly. It took *BIDsearch* only 566.8 (pattern1), 133.8 (pattern2), and 37.1 (pattern3) milliseconds to search the whole RFAM10 dataset. The obtained speedups of *BIDsearch* over *ONLsearch* were in the range from 4.63 (*1MB subset*) to 104.79 (*full* RFAM10) for pattern1, from 12.23 (*1MB subset*) to 223.18 (*full* RFAM10) for pattern2, and from 35.0 (*1MB subset*) to 618.37 (*full* RFAM10) for pattern3. We observe again that the specification of only one or two nucleotides in an RSSP's loop region considerably reduces the running time of the *BIDsearch* algorithm.

RNA family classification by global chaining of RSSP matches

To demonstrate the effect of global chaining of RSSP matches, we searched with an SSD built for the Rfam family of OxyS RNAs (Acc.: RF00035). OxyS is a small 109-nucleotide long non-coding RNA which is included in response to oxidative stress in *E.coli* [127]. Members of this family fold

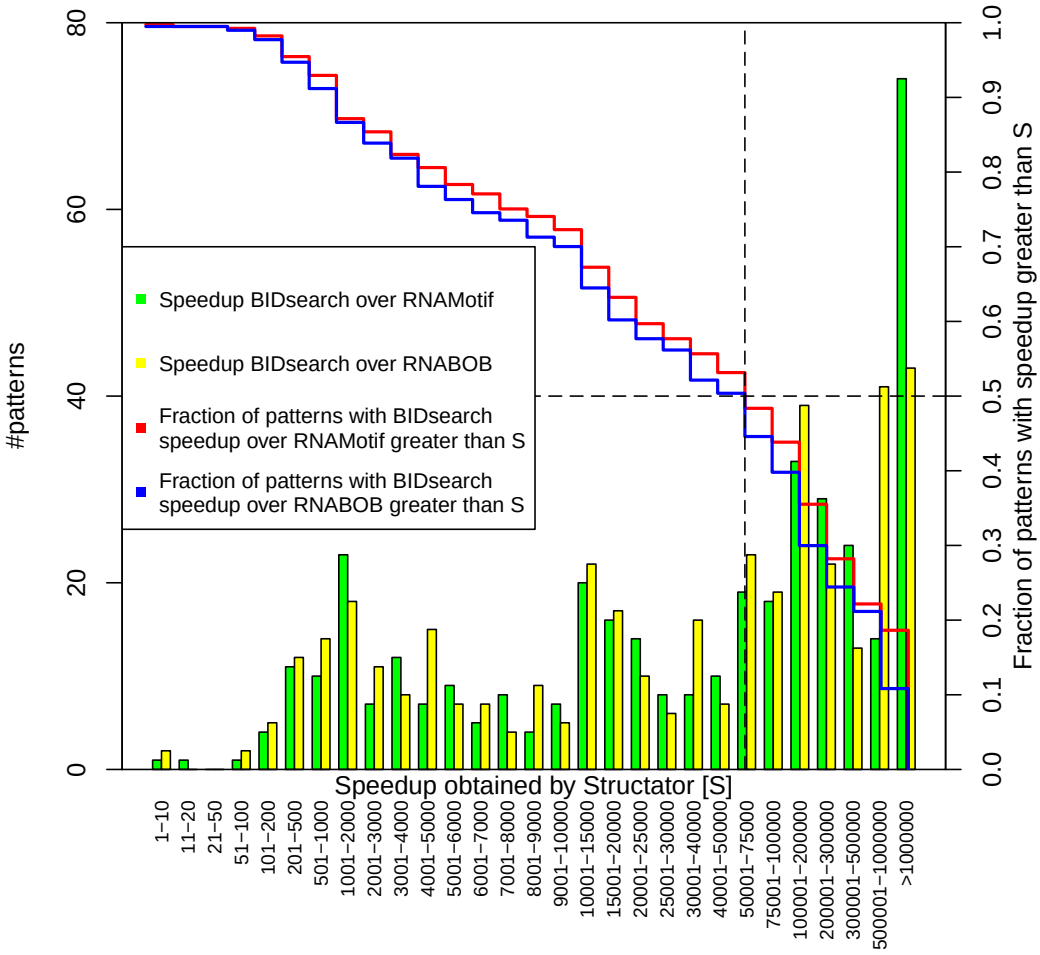


Figure 3.11: Distribution of speedup factors of *BIDsearch* over *RNABOB* (yellow) and *RNAMotif* (green) when searching for 397 RSSPs in RFAM10 consisting of ~ 622 megabases of RNA sequence data. The red and blue curves show the values of one minus the empirical cumulative distribution function of the speedup factors distributions. That is, for a given speedup factor S they show the fraction of RSSPs for which *BIDsearch* obtained a speedup greater than S over *RNAMotif* (red curve) and *RNABOB* (blue curve), respectively. We observed that *BIDsearch* is more than $50{,}000$ times faster than *RNABOB* and *RNAMotif* for the majority of the patterns (see intersection point of dashed lines). Moreover, the total search time required by *BIDsearch* is dominated by only a small number of patterns describing large unconserved loop regions.

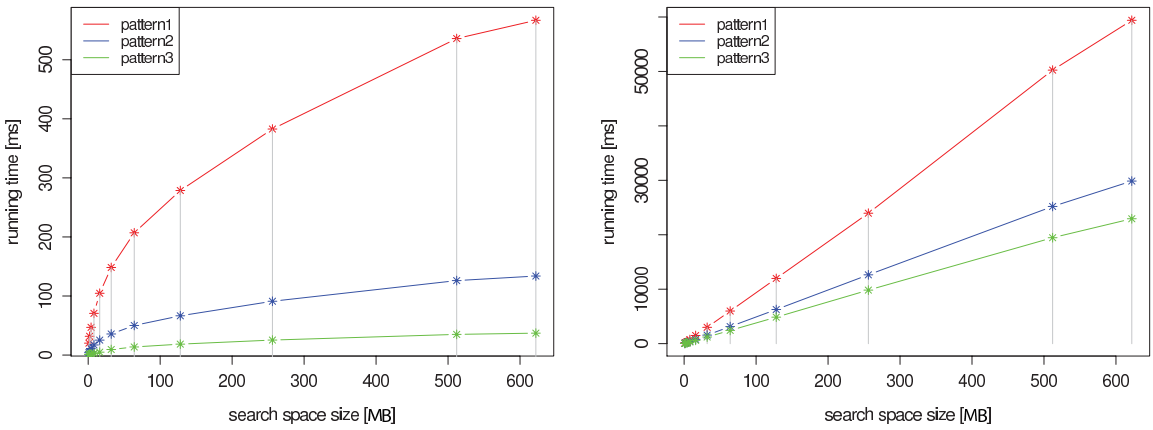


Figure 3.12: Scaling behavior *BIDsearch* (left) and *ONLsearch* (right). We measured the running time needed to search with three different patterns on random subsets of RFAM10 of different sizes. For details, see main text.

into a characteristic secondary structure consisting of three stem-loop substructures, referred to as HP1, HP2, and HP3 in Figure 3.13 (C). From the three stem-loops we derived three descriptors called RSSP1, RSSP2, and RSSP3, which constitute the SSD describing this family. We note that in this experiment the RSSPs were constructed to guarantee high specificity and thus to minimize the number of false positives. For the SSD specified in *Structator* syntax, see Figure 3.13 (A). Searching for this SSD in RFAM10, *Structator* delivers 8,619 matches for RSSP1, 1,699 matches for RSSP2, and 142,219 matches for RSSP3. Instead of reporting these matches, *Structator* computes high-scoring global chains for each sequence containing matches to all three RSSPs. The chains and the sequences they occur in are reported in descending order of the chain score. This procedure resulted in 61 sequences, all belonging to the OxyS family which contains 115 members in total. Hence, by considering only high-scoring chains all the spurious RSSP matches were eliminated. We also described the same three stem-loops in a format compatible with *RNAMotif* (see Figure 3.13 (B)). A search on RFAM10 with this descriptor returned exactly the same 61 sequences. However, *Structator* operating in *BIDsearch* (*ONLsearch*) mode with subsequent global chaining of RSSP matches needed only 3.9 (122.5) seconds to identify all family members, whereas *RNAMotif* needed 84.7 seconds. The search times for *Structator* include 0.05 seconds required for the chaining.

We also employed global chaining to detect members of the structurally more complex family of Citrus tristeza virus replication signal (Rfam Acc.: RF00193). Therefore we built an SSD comprising 8 RSSPs, describing 8 of 10 stem-loops the molecule is predicted to fold into. For more information on the molecule's secondary structure and the used descriptor, see Figure 3.14. Using *Structator* operating in *BIDsearch* (*ONLsearch*) mode and global chaining of RSSP matches it took only 1.3 (138.7) seconds to search RFAM10 with this SSD, where 0.06 seconds were required for the chaining. The computed global chains with a minimum length of 5, computed from the 184,199

(A)
```
>RSSP1|maxrightloopextent=1|maxleftloopextent=1|maxmispair=6|weight=1
NNNNNNNNNNNNNNNNNNNNNNACCCNUNANNNNNNNNNNNNNNNNN
(((((((.((((.((.((..........)).)).)..)))))))))))
>RSSP2|maxrightloopextent=5|weight=1
GNNNNNCUCACNN
((((.....))))
>RSSP3|maxmispair=2|maxrightloopextent=2|weight=1
NNGGANCUNNNNNNNNNNN
(((((((.....)))))))
```

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

(C)

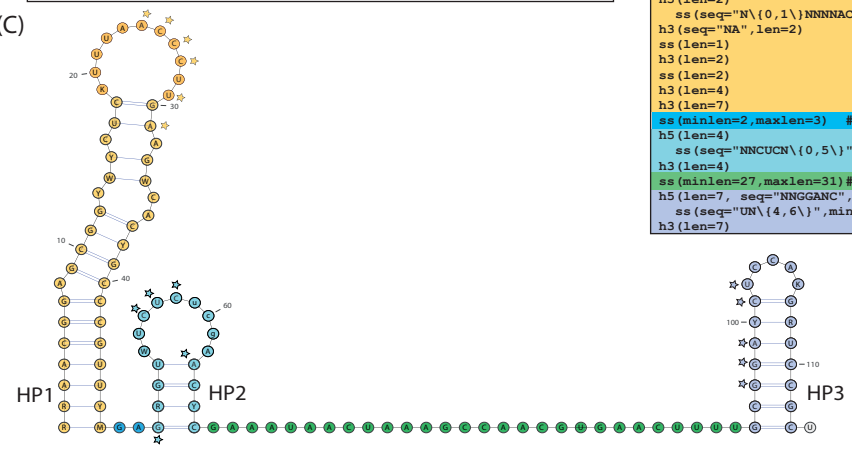


Figure 3.13: (A) Secondary structure descriptor for the family of OxyS RNAs in *Structator* syntax. The SSD consists of RSSPs RSSP1, RSSP2, and RSSP3 describing the three stem-loop structures (HP1, HP2, and HP3, see (C)) of this small non-coding RNA. (B) *RNAMotif* descriptor for the same structural elements. (C) Consensus secondary structure of the OxyS RNA family as drawn by *VARNA* [128]. Sequence information (non-wildcard nucleotides) used in both descriptors are marked with an asterisk. Observe that both descriptors use predominantly structure and very little sequence information.

single RSSP matches, were ranked according to their global chain score. We observe that the sequences containing the 37 highest scoring chains are exactly all 37 members of the family.

In addition we measured the performance of *Structator* using global chaining for RNA family classification with manually compiled SSDs for 42 Rfam families. For the results of this experiment see Table 3.4.

Searching whole genomes using local chains of RSSP matches

As an example of searching a complete genome or whole chromosomes for non-coding RNAs, we searched for the RNA gene Human accelerated region 1F (HAR1F) on both strands of the human genome sequence. HAR1F is one of 49 regions in the human genome that differ significantly from highly conserved regions of the chimpanzee [129]. The consensus structure of the HAR1F family in Rfam (Acc.: RF00635) contains three stem-loop regions, denoted HP1, HP2, and HP3 in Figure 3.16 (A). From these regions, we built an SSD for the family with RSSPs RSSP1, RSSP2, and RSSP3, shown in Figure 3.16 (B). Since we were searching on complete chromosomes, we only wanted to consider RSSP matches that occurred at a similar distance to each other w.r.t. to the

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

Table 3.4: Results of *Structator* searches on RFAM10 (1,446 families; 3,192,599 sequences) using SSDs describing 42 Rfam families. The manually compiled SSDs used in this experiment are available on the *Structator* website. They were designed to be highly specific and consist of 397 RSSPs in total with an average of 9.45 RSSPs per SSD. These are the same 397 RSSPs used in section "Searching large sequence databases". Columns 2, 3, 4, and 5 show the number of sequences containing high-scoring global chains, the numbers of true positives (TP), false positives (FP), and false negatives (FN), respectively. Sensitivity is computed as $\frac{\#TP}{\#TP+\#FN}$, specificity as $\frac{\#TN}{\#TN+\#FP}$, accuracy as $\frac{\#TP+\#TN}{\#TP+\#FP+\#FN+\#TN}$, and precision as $\frac{\#TP}{\#TP+\#FP}$. Observe that these values strongly depend on the used SSD. The number of RSSPs constituting an SSD is given in column 10. Column 11 shows the minimal required length of a chain to be considered a matching chain. Total running times of *Structator* operating in *BIDsearch* and *ONLsearch* mode are given in columns 12 and 13, respectively. Column 14 shows *BIDsearch*'s speedups over *ONLsearch*. The running time required for chaining of RSSP matches is listed in column 15. Observe that the sum of running times does not match the times needed for searching with the 397 single RSSPs reported above because here each SSD was searched using a separate *Structator* program call.

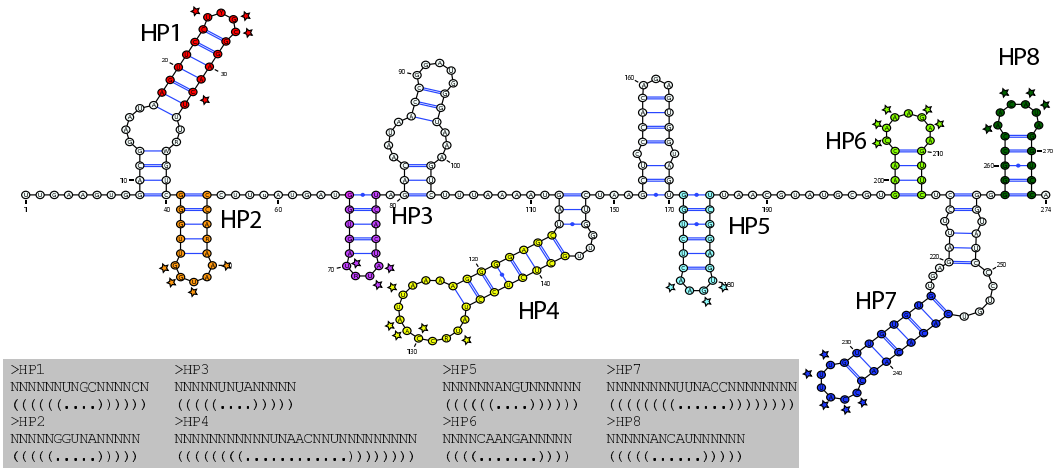


Figure 3.14: Consensus secondary structure of the CTV_rep_sig family (RFAM Acc.: RF00193) visualized with the *VARNA* program [128] and SSD in *Structator* syntax describing this family. The 8 given RSSPs correspond to the colored stem-loops HP1 - HP8. Positions at which sequence information is used in the descriptor are marked with an asterisk.

distances of the corresponding descriptors in the SSD. Therefore, unlike in the previous experiment where we searched for global chains of RSSP matches, we now computed high-scoring local chains. Gap costs were computed according to Equation (3.4) and we used an RSSP weight $\alpha(\mathsf{RSSP}_i) = 10$, for $1 \leq i \leq 3$. Affix array construction for all human chromosomes was accomplished in 12.6 hours by *afconstruct*. We searched with *Structator* for the three RSSPs and found $15{,}090$, $1{,}578$, and $14{,}491$ matches for RSSP1, RSSP2, and RSSP3, respectively. For these RSSP matches we computed local high-scoring chains (see Figure 3.16 (D)). Chains $\mathcal{C}$ were ranked according to their local chain score $lcsc\,(\mathcal{C})$. We observed that the highest-scoring chain corresponds to the correct location of the gene on chromosome 20. Using *BIDsearch* (*ONLsearch*) this task needed 3.1 (633.4) seconds only, including 0.02 seconds for the chaining. *RNAMotif* also found a single match corresponding to the correct location of the gene, but needed 274.7 seconds. See Figure 3.15 for the used *RNAMotif* descriptor.

Comparison of two implementations of bidirectional pattern search using affix arrays

We measured the speedup of *Structator* running in *BIDsearch* mode over *ONLsearch* and compared the results with previously reported measurements [101]. Because the implementation used by Strothmann [101] is not available (personal communication), we calculated relative speedups based on the absolute running times reported in [101]. We note that the measurements of [101] were performed on different hardware. This can, according to our experiments, significantly influence the performance of *BIDsearch*. See Table 3.5 for the results of the comparison of *BIDsearch*

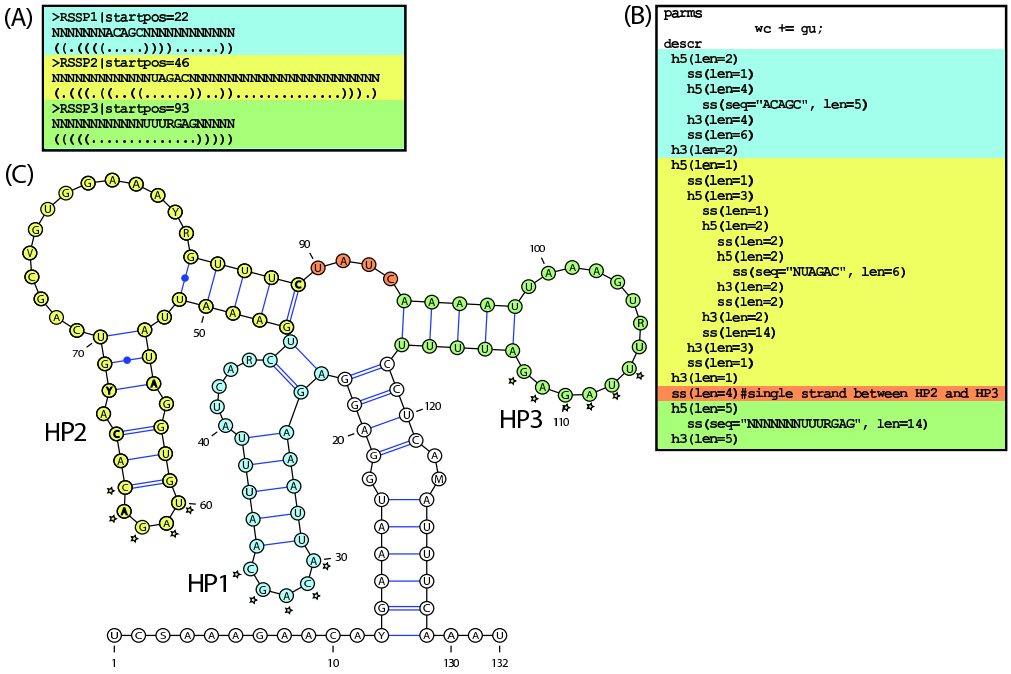


Figure 3.15: (A) SSD for HAR1F RNA family consisting of RSSP1, RSSP2, and RSSP3 in *Structator* syntax. RSSPs were built from stem-loops HP1, HP2, and HP3 shown in (C). (B) *RNAMotif* descriptor for the same structural elements. Secondary structure drawing shown in (C) was generated with *VARNA* [128].

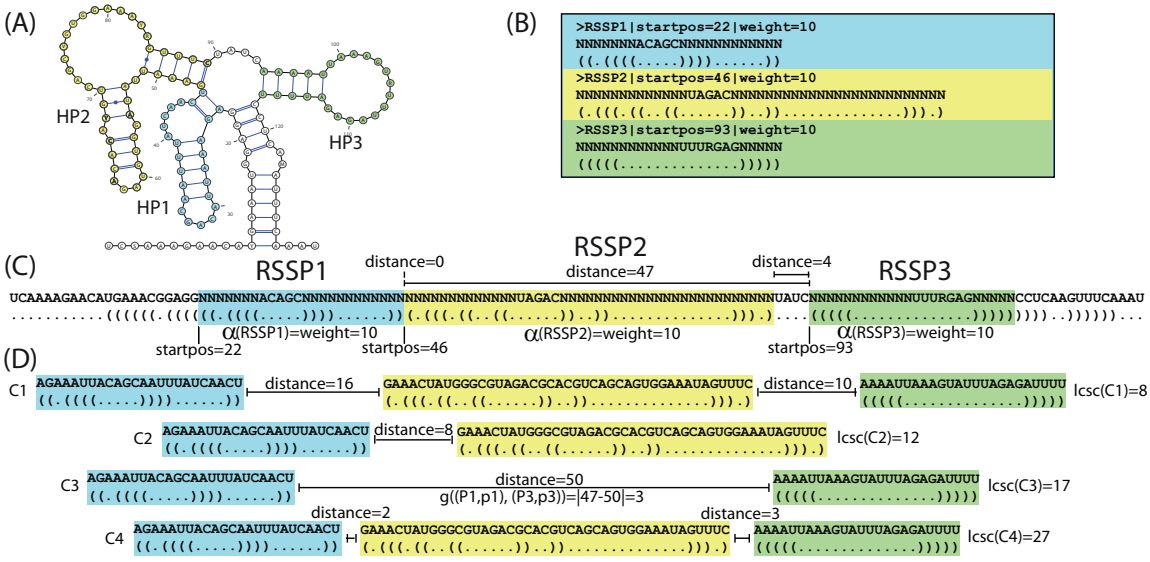


Figure 3.16: (A) Consensus secondary structure visualized with the *VARNA* program of the HAR1F RNA family showing stem-loops HP1, HP2, and HP3. (B) SSD consisting of RSSP1, RSSP2, and RSSP3 in *Structator* syntax describing the three stem-loop regions of HAR1F. (C) Regions of HAR1F described by the RSSPs, including distances $l_{i+1} - r_i$, $1 \leq i < 3$, between neighbored RSSPs and RSSP weights $\alpha(\mathsf{RSSP}_i)$, $1 \leq i \leq 3$. (D) Examples of local chains $\mathcal{C}_i$, $1 \leq i \leq 4$ found with the SSD, showing, in each chain, the distance between RSSP matches and their local chain score $lcsc\,(\mathcal{C}_i)$. Gap cost computation according to Equation (3.4) is shown exemplary for the two RSSP matches of chain $\mathcal{C}_3$.

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

Table 3.5: Comparison of speedup of *Structator*'s *BIDsearch* over *ONLsearch* (column *B*vs.*O*) and the speedup of affix array based search over searching on the plain text as reported in [101] (column *STR*). The respective search times of *BIDsearch* (column *BID*) and *ONLsearch* (column *ONL*) are shown in milliseconds. For *P. vampyrus* only measurements for *Structator* are available.

with the method of [101]. For a description of the used RSSPs see [101]. The search was performed in the genomes of *P. horikoshii* (GenBank Acc.: NC_000961, 1.7 MB) and *E. coli* (GenBank Acc.: AC_000091, 4.5 MB), which were also used in [101]. Additionally we included with *P. vampyrus* (GenBank Acc.: ABRP00000000, 1.9 GB) a larger eukaryotic genome in this experiment.

Surprisingly, with the RSSPs ACloop(5), ACloop(10), and ACloop(15) taken from [101], which describe a loop consisting of 5 (10 and 15) repetitions of AC, the speedup of the affix array based method of [101] decreased with increasing loop length. This is a behavior which is opposite to our observations (see Figure 3.10). We also noticed that *BIDsearch* obtained a higher speedup when searching for RSSP Hpin2 in *E. coli* than the method of [101] but not when searching in the smaller genome of *P. horikoshii*. This observation remains unclear and cannot be further investigated due to unavailability of the implementation used in [101].

Comparison with an implementation of bidirectional pattern search using a compressed data structure

In the last experiments we compared *Structator*'s running time using using *BIDsearch* with the time needed by a recently published bidirectional pattern search implementation for the same task. The implementation of [130], to which we refer as *BWI*, uses a compressed data structure called bidirectional wavelet index. We remark that *BWI* can only search with a small set of hard-coded patterns, i.e., the user cannot use it to search with his/her own patterns. Moreover, unlike *Structator*, which provides a full command line interface with many configurable options (see section about the software package), *BWI* reports neither matching substrings nor matching positions (which is known to be the most time consuming part when querying compressed index structures [53]). It only outputs the search time of individual patterns and the number of matches. Thus, it serves rather as a prototype implementation of the concepts introduced in [130]. Nevertheless, since it also makes

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

Table 3.6: Search time comparison between *Structator*'s *BIDsearch* and an implementation, here called *BWI*, of bidirectional search using the wavelet tree data structure described in [130]. Search times are in milliseconds. The last row shows the speedup of *BIDsearch* over *BWI*.

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

Table 3.7: Size in megabytes of the bidirectional wavelet index (BWI) [130] for different genomes.

use of bidirectional search, we compared *BWI* with *Structator* using *BWI*'s hard-coded patterns. See Table 3.6 for the results. Details of the database and patterns are as previously described [130]. We noticed that *BIDsearch* was faster than *BWI* for matching all patterns by up to factor 2, hence making it preferable when speed is most important. However, we note that *BWI*'s compressed wavelet index consumes significantly less memory than *Structator*'s affix array index, which would make *BWI* preferable in cases where space consumption is critical. See Table 3.7 for the memory required by *BWI*'s index for different genomes.

3.7 *Structator* software package

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

afconstruct implements all algorithms necessary for affix array construction, namely a lightweight suffix sorting algorithm for construction of the suffix arrays suf_F and suf_R, the algorithm for construction of tables lcp_F and lcp_R [111], and the algorithm for computation of the affix link tables aflk_F and aflk_R. The program constructs all or if necessary only some of the tables of the affix array for a target database provided in FASTA format and stores them on disk. Therefore the program

can also be used to compute only the tables needed for a traditional enhanced suffix array [109]. *afconstruct* can handle RNA as well as DNA sequences. Moreover, it supports the transformation of input sequences according to user-defined (reduced) alphabets and allows the index construction for transformed sequences. Such personalized alphabets are easily specified in a text file.

afsearch is the program for performing structural pattern matching. That is, it searches (ribo)nucleic acid sequence databases for entries that can adopt a particular secondary structure. For an overview of the supported RNA sequence-structure patterns (RSSPs), see Figure 3.5. The simplest RSSP describes a single-stranded region, where ambiguous (not well-conserved) nucleotides can be specified with IUPAC characters. All ambiguous IUPAC characters are hard-coded in *afsearch*, e.g. N standing for nucleotides A, C, G, and U (and T) and R standing for A and G. Besides fixed-length RSSPs with or without ambiguous characters (Figure 3.5 (A) until (D)), also RSSPs describing loop or stem regions of variable size (Figure 3.5 (E) until (H)) are supported. More precisely, one can specify with parameters *maxleftloopextent (mllex)* and *maxrightloopextent (mrlex)* a variable number of allowed extensions to the left (nucleotides marked in yellow in Figure 3.5 (E)) and/or to the right (nucleotides marked in blue in Figure 3.5 (F)) for the specified loop pattern. Variable stem sizes can be addressed with parameter *maxstemlength (msl)* (see regions marked in pink in Figure 3.5 (G)). Also supported is the combination of variable loop and stem size (see Figure 3.5 (H)) and a maximal number of allowed mispairings in the stem region. All these different RSSPs can be specified by the user in a text file which use, as shown in Figure 3.5, an expressive but easy to understand pattern syntax. For additional details on the supported patterns see the corresponding section in the *Structator* user manual. *afsearch* also permits user-defined base pairing rules. That is, the user can define an arbitrary subset from $\mathcal{A} \times \mathcal{A}$ as valid pairings. This ensures a maximum of flexibility. For example, the standard canonical Watson-Crick pairings as well as non-standard pairings such as G-U can be specified.

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

RSSP matches can be reported directly by *afsearch* or can be used as input for the computation of high-scoring global or local chains of matches. Computed chains resemble the order of the RSSPs given in the pattern file and are reported in descending order of their chain score. This allows the description of complex secondary structures with our new concept of secondary structure descriptors (SSDs). This is done by simply specifying a series of RSSPs in the pattern file describing the stem-loop substructures the RNA molecule is composed of in the order of their occurrence in

5' to 3' direction. To incorporate different levels of importance or significance of an RSSP into SSD models and subsequently in the computation of chain scores, RSSP specific weights can be defined in the pattern file. This is particularly useful in the context of RNA family classification where the used SSD may be derived from a multiple sequence-structure alignment or a consensus structure-annotated multiple sequence alignment. Here, it permits the assignment of higher weights to RSSPs describing highly conserved functionally important structural elements occurring in a family of RNAs, and lower weights to RSSPs describing less conserved substructures that occur only in certain members of the family.

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

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

3.8 Discussion and concluding remarks

We have presented a method for fast index-based search of RNA sequence-structure patterns (RSSPs), implemented in the *Structator* software. As part of the software, we give the first publicly available implementation of bidirectional pattern search using the affix array data structure. For the majority of biologically relevant RSSPs, our implementation of *BIDsearch* shows superior performance over previous programs. In a benchmark experiment on the Rfam database, *BIDsearch* was faster than *RNAMotif* and *RNABOB* by up to two orders of magnitude. Furthermore, in a comparison between *BIDsearch* and the program of [130], which works on compressed index data structures, *BIDsearch* was faster by up to 2 times. We observed that for RSSPs with long unconserved loop regions, the advantage of *BIDsearch* over *ONLsearch* decreases. For such cases, *Structator* can also employ *ONLsearch* on the plain sequence data. As a further contribution, we presented for the first time a detailed complexity analysis of bidirectional search using affix arrays. While bidirectional search does not improve the worst-case time complexity compared to online search, in practice it runs much faster than online search algorithms and the running time scales sublinearly with the length n of the searched sequences.

Our implementation of the affix array data structure requires only $18n$ bytes of space. This is a significant space reduction compared to the $\sim 45n$ bytes needed for the affix tree. With the program *afconstruct* we present for the first time a command line tool for the efficient construction and persistent storage of affix arrays that can also be used as a stand-alone program for index construction. We note that bidirectional search with an affix array is also possible using $10n$ bytes of space as observed in [131] after the publication of our work. This is achieved by avoiding the storage of the affix link tables. However, this approach requires the computation of affix links during the search

for structural patterns and, consequently, increases the number of binary searches in the suffix and reverse prefix arrays.

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

4 Fast approximate search for RNA sequence-structure patterns

4.1 Introduction

Our *Structator* tool presented in the former chapter addresses a fundamental drawback of previous descriptor-based RNA homology search methods, i.e. the fact that their running times scale at least linearly in the size of the target sequence database. *Structator*, in contrast, achieves sublinear running time by using the affix array index data structure, which allows to perform bidirectional pattern search and efficiently handle the structural constraints of the patterns.

However, 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 *Structator*. 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 [132] 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 an 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.* [70], 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

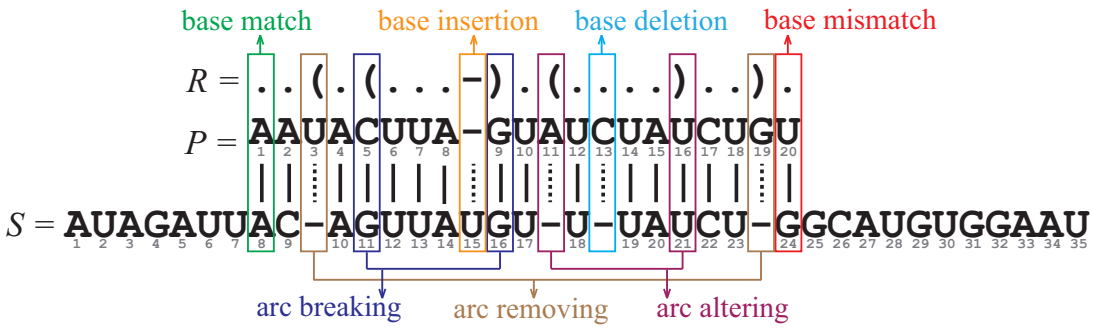


Figure 4.1: Example of a semi-global alignment of a sequence-structure pattern $\mathcal{Q} = (P, R)$ and an RNA sequence S and involved sequence-structure edit operations. Continuous (dashed) lines indicate match (gap) alignment edges from A_{match} (A_{gap}).

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 the *Structator* tool, our new algorithms 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 [104, 126]. 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.

The description of our algorithms closely follows [133]. For the used notation, please see the formal preliminaries in Chapters 2 and 3.

4.2 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 $\{(x, -) \mid x \in [1..m] \wedge \nexists y, (x, y) \in A_{\text{match}}\} \cup \{(-, y) \mid y \in [p..q] \wedge \nexists x, (x, y) \in A_{\text{match}}\}$. See Figure 4.1 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 $S[l]$, $1 \leq k \leq m$, $p \leq l \leq q$, are:

- *base mismatch*, with cost ω_{m}, which occurs if there is an edge $(k,l) \in A_{\text{match}}$ and $S[l] \notin \varphi(P[k])$;
- *base match*, with cost zero, which occurs if there is an edge $(k,l) \in A_{\text{match}}$ and $S[l] \in \varphi(P[k])$;
- *base deletion*, with cost ω_{d}, which occurs if $(k,-) \in A_{\text{gap}}$; and
- *base insertion*, also with cost ω_{d}, which occurs if $(-,l) \in A_{\text{gap}}$.

The possible edit operations on base pairs were first introduced by Jiang *et al.* [70] and are defined as follows. Let (k_1,k_2) be a base pair in $\widehat{R}$ and l_1 and l_2, $p \leq l_1 < l_2 \leq q$, 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 4.1. 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 4.1 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 4.1 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

$$dist(\mathcal{Q}, S[p..q]) = \min\{dist_A(\mathcal{Q}, S[p..q]) \mid A \text{ is an alignment of } \mathcal{Q} \text{ and } S[p..q]\} \tag{4.1}$$

where

$$
\begin{aligned}
& dist_A(\mathcal{Q}, S[p..q]) = \\
& \sum_{(k,l)\in A, R[k]=., S[l]\notin\varphi(P[k])} \omega_{\mathrm{m}} && \text{base mismatch} \\
+ & \sum_{(k,-)\in A, R[k]=.} \omega_{\mathrm{d}} && \text{base deletion} \\
+ & \sum_{(-,l)\in A} \omega_{\mathrm{d}} && \text{base insertion} \\
+ & \sum_{(k_1,k_2)\in\widehat{R},(k_1,l_1)\in A,(k_2,l_2)\in A,(S[l_1],S[l_2])\notin\mathcal{C}} \omega_{\mathrm{b}} && \text{arc breaking} \\
+ & \sum_{(k_1,k_2)\in\widehat{R},(k_1,l_1)\in A,(k_2,-)\in A} \omega_{\mathrm{a}} && \text{arc altering} \\
+ & \sum_{(k_1,k_2)\in\widehat{R},(k_2,l_2)\in A,(k_1,-)\in A} \omega_{\mathrm{a}} && \text{arc altering} \\
+ & \sum_{(k_1,k_2)\in\widehat{R},(k_1,-)\in A,(k_2,-)\in A} \omega_{\mathrm{r}} && \text{arc removing.}
\end{aligned}
\tag{4.2}
$$

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

$$dist(\mathcal{Q}, S[p..q]) \leq \mathcal{K} \text{ and } m - d \leq |S[p..q]| \leq m + d \leq n. \tag{4.3}$$

We call every substring $S[p..q]$ satisfying Equation (4.3) 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.

4.2.1 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 $1 \leq p \leq q \leq n$ 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.* [70], 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:

$$\chi(i,j) = \begin{cases} 0 & \text{if } T[j] \in \varphi(P[i]) \quad \text{(base match)} \\ 1 & \text{otherwise.} \quad\quad\quad\quad\quad\quad\quad\quad \text{(base mismatch)} \end{cases} \tag{4.4}$$

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:

$$comp(i,j) = \begin{cases} 0 & \text{if } (T[i], T[j]) \in \mathcal{C} \quad \text{(complementary)} \\ 1 & \text{otherwise.} \quad\quad\quad\quad\quad\quad\quad\quad \text{(not complementary)} \end{cases} \tag{4.5}$$

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:

$$row(i) = \begin{cases} i' & \text{if } (i', i) \in \widehat{R} \text{ and } 1 < i' < i < m \text{ and } R[i+1] = . \text{ and } R[i'-1] \neq (\\ 0 & \text{if } (i, i') \in \widehat{R} \text{ and } R[i+1] = . \\ i & \text{otherwise.} \end{cases} \tag{4.6}$$

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

$$DP_k(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ DP_k(0, j-1) + \omega_{\mathrm{d}} & \text{if } i = 0 \text{ and } j > 0 \\ DP_k(row(i-1), 0) + \omega_{\mathrm{d}} & \text{if } i > 0 \text{ and } j = 0 \\ \min \left\{ \begin{array}{l} DP_k(row(i-1), j) + \omega_{\mathrm{d}} \\ DP_k(i, j-1) + \omega_{\mathrm{d}} \\ DP_k(row(i-1), j-1) + \chi(i,j)\omega_{\mathrm{m}} \end{array} \right\} & \text{if } i > 0 \text{ and } j > 0 \end{cases} \tag{4.7}$$

2. If $R[i] \neq$. (paired base), then

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

$$DP_k(i,j) = \begin{cases} DP_k(row(i-1), 0) + \omega_{\mathrm{r}} & \text{if } j = 0 \\ \min \left\{ \begin{array}{l} DP_k(row(i-1), j-1) + \chi(i, j+k)\omega_{\mathrm{m}} + \omega_{\mathrm{a}} \\ DP_{k+1}(row(i-1), j-1) + \chi(i', k)\omega_{\mathrm{m}} + \omega_{\mathrm{a}} \\ DP_k(row(i-1), j) + \omega_{\mathrm{r}} \\ DP_k(i, j-1) + \omega_{\mathrm{d}} \\ DP_{k+1}(i, j-1) + \omega_{\mathrm{d}} \\ DP_{k+1}(row(i-1), j-2) + (\chi(i, j+k) + \chi(i', k+1))\omega_{\mathrm{m}} + \\ comp(k+1, j+k)\omega_{\mathrm{b}}, \text{ if } j > 1 \end{array} \right\} & \text{if } j > 0 \end{cases} \tag{4.8}$$

   (b) If (a) holds and either $R[i'-1] =$. or $R[i'-1] =$), compute in addition to Equation (4.8)

$$DP_k(row(i), j) = \begin{cases} DP_k(row(i'-1), 0) + DP_k(i, 0) & \text{if } j = 0 \\ \min \left\{ DP_k(row(i'-1), j') + DP_{k+j'}(i, j-j') \mid 0 \leq j' \leq j \right\} & \text{if } j > 0 \end{cases} \tag{4.9}$$

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 determined 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 4 *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 $T_{l+1}[1..m'-l] = S[p+l..q]$, $0 \leq l < m'$, 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 4, our algorithm computes only the last column of the DP matrices for every shifted window substring (see the example in Figure 4.2) 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 [133] where we for the first time describe *ScanAlign*, Will *et al.* [68] 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.

4.2.2 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.

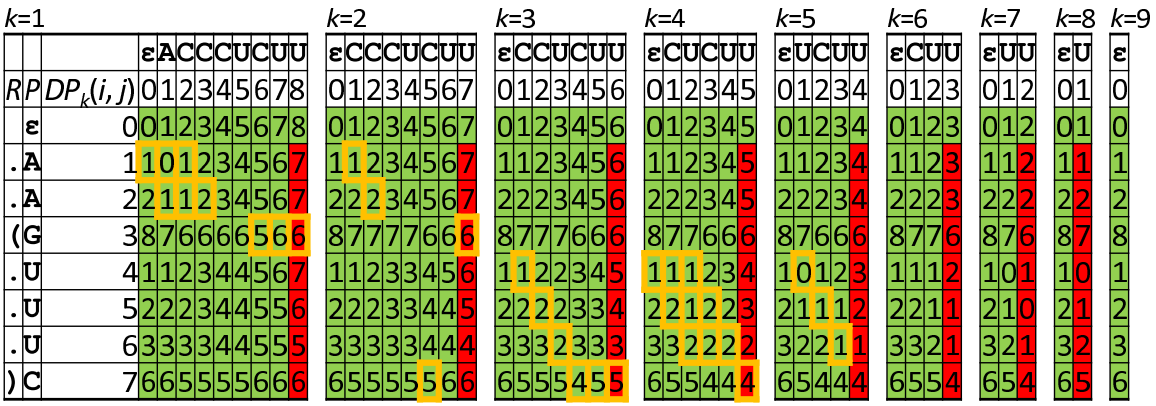


Figure 4.2: DP tables for the sequence-structure alignment computation of RSSP $\mathcal{Q} = (\text{AAGUUUC}, \texttt{. . (. . .)})$ and window substring $T = \text{ACCCUCUU}$ when scanning a sequence S with algorithm *ScanAlign*. Only the entries in red have to be computed for each window shift, whereas 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: $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = 1$, $\omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 2$, and $\omega_{\mathrm{r}} = 3$.

- 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 $\mathcal{Q}[x..y]$, $1 \leq x \leq y \leq m$, satisfies exactly one of the following conditions.

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 4.3.
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 4.3, 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 4.3, namely $\mathcal{Q}[5..9]$, $\mathcal{Q}[11..16]$, and $\mathcal{Q}[3..19]$.

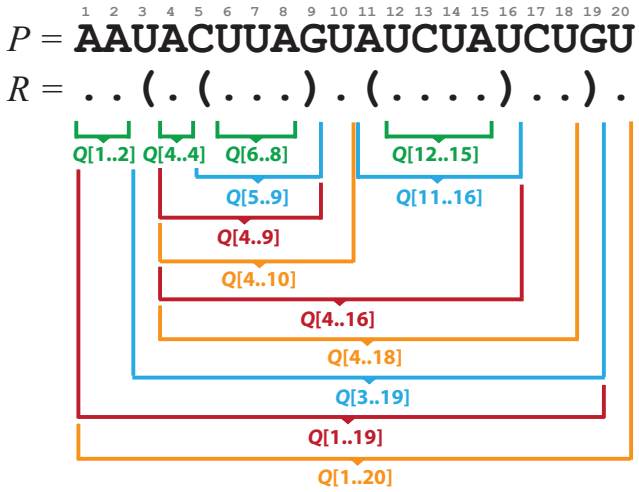


Figure 4.3: Regions of RSSP $\mathcal{Q} = (\text{AAUACUUAGUAUCUAUCUGU}, \text{. . (. (. . .) . (. . . .) . .) .})$ according to conditions 1 (green), 2 (orange), 3 (blue), and 4 (red) described in the text.

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 4.3, 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 4.3, 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 (4.7).
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 (4.7).

3. Row y is computed by recurrence (a) of Equation (4.8).
4. Row $row(y)$ is computed by recurrence (b) of Equation (4.8).

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 [134].
- 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 5 *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 $dist(\mathcal{Q}[x..y], T_{x+d'}[1..l+z]) > \mathcal{K}$, then, for every d'', $0 \leq d'' \leq d$, $dist(\mathcal{Q}, T[1..m' - d'']) > \mathcal{K}$.*

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 $j - i + \mathit{offset} = e$, where $\mathit{offset} = x - 1$. 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 5, 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 $dist(\mathcal{Q}[x..y], T_{x+d'}[1..l+z]) \leq \mathcal{K}$, $z \in \{|d'| - d, -|d'| + d\}$, 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', $e - d \leq e' \leq e + d$, $DP_k(i,j) \leq \mathcal{K}$. 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 4.2 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 4 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.

4.2.3 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 [52, 109]. 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. These correspond to tables $\mathsf{suf_F}$ and $\mathsf{lcp_F}$ defined above. 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 [109] 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 recomputation of $\sum_{k=1}^{lcp} lcp - k + 1$ columns for $S_{\mathsf{suf}[i]}$. See an example in Figure 4.4. 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 [54]. However, to save the

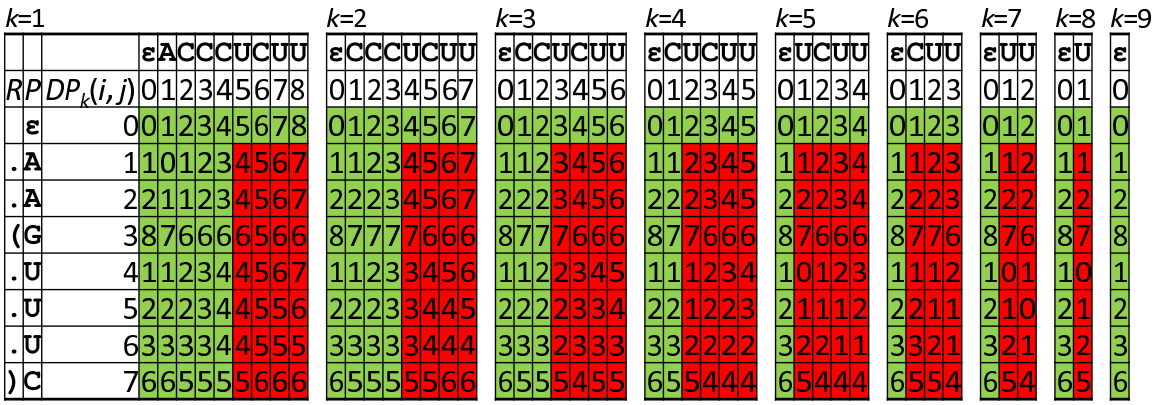


Figure 4.4: DP tables for the sequence-structure alignment computation of RSSP $\mathcal{Q} = (\text{AAGUUUC}, \texttt{..(...)})$ and substring $S_{\mathsf{suf}[i]}[1..8] = \text{ACCCUCUU}$. Given that suffix $S_{\mathsf{suf}[i]}$ shares a common prefix of length $\mathsf{lcp}[i] = 4$ with $S_{\mathsf{suf}[i-1]}$, algorithm *LESAAlign* reuses the entries in green and computes the entries in red. Used operation costs: $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = 1$, $\omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 2$, and $\omega_{\mathrm{r}} = 3$.

$4n$ 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 5. 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 4, 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', $-d \leq d' \leq \min\{d, x\}$, $dist(\mathcal{Q}[x..y], T_{x+d'}[1..l+z]) > \mathcal{K}$ with $l = y-x+1$, $z \in \{|d'|-d, -|d'|+d\}$, and $y+d' \leq m+d$

Algorithm 5: *LESAAlign*

input : Index tables suf and lcp of sequence S, RSSP $\mathcal{Q}$
output: Matching positions of $\mathcal{Q}$ in S

```
1  iSuffix := 1
2  iLcp := 0
3  lastRegion := undefined
4  while iSuffix ≤ n do
5  |  (bMatched, lastRegion) := computeDP(DP, iSuffix, iLcp, lastRegion)
6  |  iLcpCheck := lastRegion.r + d
7  |  if bMatched then
8  |  |  reportMatch(Q, S, iSuffix) //Match found at position suf[iSuffix] of S
9  |  end
10 |  iSuffix := iSuffix + 1
11 |  if iSuffix ≤ n then
12 |  |  iLcp := lcp[iSuffix]
13 |  while iSuffix ≤ n and lcp[iSuffix] ≥ iLcpCheck do
14 |  |  if lcp[iSuffix] < iLcp then
15 |  |  |  iLcp := lcp[iSuffix] //Store the smallest lcp value of the skipped interval
16 |  |  end
17 |  |  if bMatched then
18 |  |  |  reportMatch(Q, S, iSuffix) //Match found at position suf[iSuffix] of S
19 |  |  end
20 |  |  iSuffix := iSuffix + 1
21 |  end
22 end
```

Figure 4.5: Pseudocode for algorithm *LESAAlign*. For details, see main text.

(see Lemma 5). 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.

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 $S_{\mathsf{suf}[i]}$.

The pseudocode for algorithm *LESAAlign* is given in Algorithm 5 (Figure 4.5). *LESAAlign* traverses the suffix array suf of the target sequence S top down, beginning with the lexicographically smallest

suffix $S_{\mathsf{suf}[iSuffix]}$, where $iSuffix$ = 1 at this stage. During the traversal, it computes the sequence-structure edit distance $dist(\mathcal{Q}, S_{\mathsf{suf}[iSuffix]}[1..p_{iSuffix}])$ between the RSSP $\mathcal{Q}$ and the prefix of length $p_{iSuffix}$ of each suffix $S_{\mathsf{suf}[iSuffix]}$, for $1 \leq iSuffix \leq n$. This computation is done by function $computeDP$ in line 5. The input parameters of $computeDP$ are the computed DP matrices, the index $iSuffix$ of the current suffix, the length $iLcp$ of the common prefix between the last processed suffix and the current suffix, and the last computed pattern region $\mathcal{Q}[x..y]$ denoted $lastRegion$ in the code. The last two variables are used to avoid recomputation of entries of DP matrices. Function $computeDP$ returns a boolean value, stored in $bMatched$, stating whether the pattern was matched, and the last newly computed region $lastRegion$. $lastRegion.r$ is the right boundary of the last computed pattern region $\mathcal{Q}[x..y]$ and is used to compute variable $iLcpCheck$ in line 6. $iLcpCheck$, in turn, is used to check whether suffixes of the suffix array sharing a common prefix can be skipped. If $bMatched$ is true, matches are reported by function $reportMatch$ in lines 8 and 18.

4.2.4 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 $\mathcal{Q} = (\text{AAGUUUC}, \texttt{..(...)})$ and substring $S_{\mathsf{suf}[4]}[1..p_4] =$ ACCCUCUU in Figure 4.4. The enhanced suffix array of S is shown in Figure 4.6. 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.

i	suf[i]	lcp[i]	suf^{-1}[i]	$S_{\text{suf}[i]}$
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 4.6: Enhanced suffix array of sequence $S\$ = \text{CCACCCCCCACCCACCACCCUCUU\$}$ consisting of the suffix array suf, longest common prefix array lcp, and inverse suffix array suf^{-1}.

To address the first goal, we motivate our method by recalling the alignment computation example in Figure 4.2. In this example, one of the regions of $\mathcal{Q} = (\text{AAGUUUC}, \texttt{..(...)})$ is $\mathcal{Q}[3..7] = (\text{GUUUC}, \texttt{(...)})$. 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 5). In other words, $\mathcal{Q}[3..7]$ cannot align to any of the substrings $T[2..6] = \text{CCCUC}$, $T[3..6] = \text{CCUC}$, $T[3..7] = \text{CCUCU}$, $T[3..8] = \text{CCUCUU}$, or $T[4..8] = \text{CUCUU}$ 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 $T[2..8] = \text{CCCUCUU}$ 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 $|T| > 1$ [135]. 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 $link : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ as:

$$link(i, x') = \mathsf{suf}^{-1}[\mathsf{suf}[i] + x']. \tag{4.10}$$

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 4.2, determining $dist(\mathcal{Q}[4..6], T_{4+d'}[1..3+z]) > 1$, $-1 \leq d' \leq 1$, $z \in \{|d'|-1, -|d'|+1\}$, does not depend on region $\mathcal{Q}[1..2]$. Hence, region $\mathcal{Q}[1..2]$ 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 4. 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*. *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.

The pseudocode for algorithm *LGSlinkAlign* is given in Algorithm 6 (Figure 4.7). *LGSlinkAlign* traverses the suffix array in two combined strategies: top down and following suffix links. This is managed in the code with two main while-loops, where an outer loop (lines 3 to 47) performs the top down traversal and an inner loop (lines 13 to 46) performs the traversal via suffix links. To keep track of the last processed suffix via top down suffix array traversal, the index of this suffix is stored in variable $iSuffixTopDown$. To keep the code short, all alignment computations are performed only in the inner loop, distinguishing the strategy by which suffixes are traversed according to the boolean variable $bFollowedSuffixLink$. This variable is set to true (line 41) when the inner loop iterates and to false (line 4) when the iteration breaks. When $bFollowedSuffixLink$ is false, the same $computeDP$ function used by the *LESAAlign* algorithm is applied. Otherwise function $computeLastDPColumns$ is applied. This function does not use lcp information, but takes advantage of the fact that the prefix of the current suffix, determined in line 36 by following a suffix link, is equal to the previously processed suffix prefix, except by its last character. This property of the suffix prefix allows to reuse already computed entries of matrices from the previously processed suffix prefix, requiring for this only one shift of the DP matrices. This is done by function $shiftDP$ in line 42. While traversing the suffix array, processed suffixes are marked in the vtab table. This allows to avoid processing the same suffixes multiple times. In addition to these processed suffixes,

non-contiguous suffixes of the suffix array that are known not to contain matches to RSSP $\mathcal{Q}$ are also marked in this table. This is possible when pattern $\mathcal{Q}$, for the current suffix, has an unaligned prefix of length $iUnalignedPrefixLength > 0$. For determining $iUnalignedPrefixLength$ in line 31, value $lastRegion.l$ is used. This value is the left boundary of the last computed pattern region $\mathcal{Q}[x..y]$. Marking the additional suffixes in vtab is performed by function $markSuffixes$ (see Figure 4.8). This function receives as parameter a starting index $iSuffix$, $iUnalignedPrefixLength$, and the required length $iLcp$ of the common prefixes of the suffixes to be marked. The function then traverses the suffix array top down and bottom up, marking all possible suffixes in vtab.

4.2.5 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 4.6. 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 $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = \omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 1$ and $\omega_{\mathrm{r}} = 2$. 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

Algorithm 6: *LGSlinkAlign*

```
   input  : Index tables suf, lcp, suf^-1, and vtab of sequence S, RSSP Q
   output: Matching positions of Q in S
 1 iSuffixTopDown := 1
 2 lastRegion := undefined
 3 while iSuffixTopDown ≤ n do //Begin traversing suffix array top down
 4 |   bFollowedSuffixLink := false
 5 |   iLcp := lcp[iSuffixTopDown]
 6 |   while vtab[suf[iSuffixTopDown]] do //Skip already visited suffixes
 7 |   |   iSuffixTopDown := iSuffixTopDown + 1
 8 |   |   if iLcp > lcp[iSuffixTopDown] then //Store the smallest lcp value of the skipped interval
 9 |   |   |   iLcp := lcp[iSuffixTopDown]
10 |   |   end
11 |   end
12 |   iSuffix := iSuffixTopDown
13 |   while not vtab[suf[iSuffix]] do
14 |   |   if bFollowedSuffixLink then //Current suffix was obtained via a suffix link
15 |   |   |   (bMatched, lastRegion) := computeLastDPColumns(DP, iSuffix, lastRegion)
16 |   |   else//Current suffix was obtained via the top-down suffix array traversal
17 |   |   |   (bMatched, lastRegion) := computeDP(DPTopDown, iSuffix, iLcp, lastRegion)
18 |   |   end
19 |   |   iLcpCheck := lastRegion.r + d
20 |   |   repeat
21 |   |   |   vtab[suf[iSuffix]] := true
22 |   |   |   if bMatched then
23 |   |   |   |   reportMatch(Q, S, iSuffix) //Match found at position suf[iSuffix] of S
24 |   |   |   end
25 |   |   |   iSuffix := iSuffix + 1
26 |   |   |   if iSuffix > n or vtab[suf[iSuffix]] then
27 |   |   |   |   break
28 |   |   |   end
29 |   |   until lcp[iSuffix] ≥ iLcpCheck
30 |   |   iSuffix := iSuffix − 1
31 |   |   iUnanlignedPrefixLength := lastRegion.l − d − 1
32 |   |   if iUnanlignedPrefixLength > 0 then
33 |   |   |   markSuffixes(link(iSuffix, iUnanlignedPrefixLength), iUnanlignedPrefixLength,
34 |   |   |   lastRegion.r + d − iUnanlignedPrefixLength)
35 |   |   end
36 |   |   iSuffix := link(iSuffix, 1)
37 |   |   if |S_suf[iSuffix]| ≥ m − d then //If suffix is not shorter than the minimum required length
38 |   |   |   if not bFollowedSuffixLink then
39 |   |   |   |   DP := DPTopDown
40 |   |   |   end
41 |   |   |   bFollowedSuffixLink := true
42 |   |   |   shiftDP(DP)
43 |   |   else //Leave large while-loop and traverse suffix array top down
44 |   |   |   break
45 |   |   end
46 |   end
47 end
```

Figure 4.7: Pseudocode for algorithm *LGSlinkAlign*. For details, see main text.

Function $markSuffixes(iSuffix, iUnanlignedPrefixLength, iLcpCheck)$

```
1  //Mark suffixes by traversing suffix array top down
2  iSuffixDown := iSuffix + 1
3  while iSuffixDown ≤ n and lcp[iSuffixDown] ≥ iLcpCheck do
4  |   if suf[iSuffixDown] − iUnaligned PrefixLength ≥ 1 then
5  |   |   vtab[suf[iSuffixDown] − iUnaligned PrefixLength] := true
6  |   end
7  |   iSuffixDown := iSuffixDown + 1
8  end
9  //Mark suffixes by traversing suffix array bottom up
10 iSuffixUp := iSuffix − 1
11 while iSuffixUp ≥ 1 and lcp[iSuffixUp + 1] ≥ iLcpCheck do
12 |   if suf[iSuffixUp] − iUnaligned PrefixLength ≥ 1 then
13 |   |   vtab[suf[iSuffixUp] − iUnaligned PrefixLength] := true
14 |   end
15 |   iSuffixUp := iSuffixUp − 1
16 end
```

Figure 4.8: Function *markSuffixes* used by algorithm *LGSlinkAlign* to mark processed suffixes in table vtab. For details, see text above.

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.

4.3 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), which we introduced with our *Structator* method described above. We also use the same efficient local and global chaining algorithms as in *Structator*. 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.

4.4 Implementation and computational results

We implemented (1) the fast index-based algorithms *LESAAlign* and *LGSlinkAlign*, (2) the online algorithms *LScanAlign* and *ScanAlign*, both operating on the plain sequence, and (3) integrated with the search algorithms the efficient global and local chaining algorithms described in [104]. In our experiments we use *ScanAlign*, which is the scanning version of the method proposed in [70], 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 [111]. 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 4.9 describing the consensus secondary structure of the tRNA family (Acc.: RF00005). The results of this experiment are presented in Figure 4.10 and Tables 4.1, 4.2, and 4.3. *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 4.10 (2), also the number of allowed indels d influences the search time.

Influence of allowed edit costs and number of indels on search time

We describe an experiment comparing the running times of algorithms *LGSlinkAlign*, *LESAAlign*, *LScanAlign*, and *ScanAlign* to search in RFAM10.1, similar to the benchmark described above.

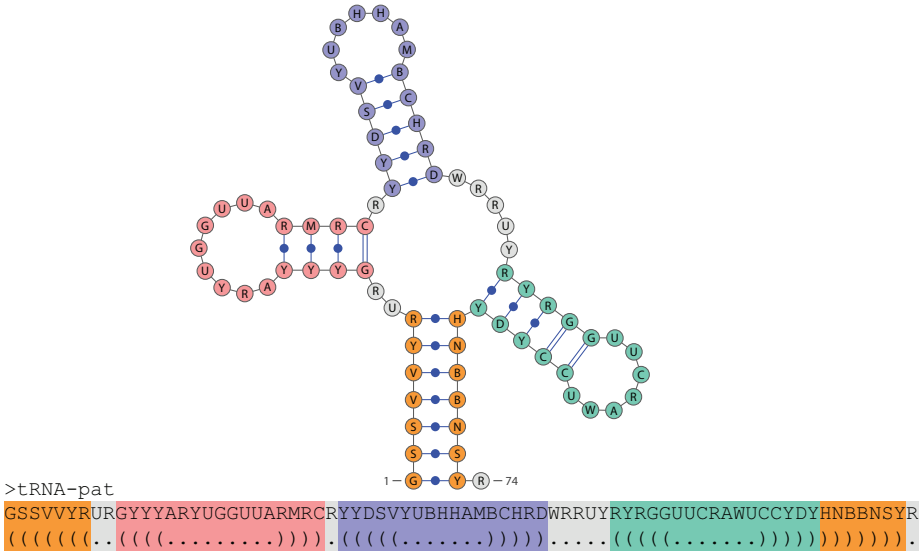


Figure 4.9: Consensus secondary structure of the tRNA family (Acc.: RF00005) as drawn by *VARNA* [128] (top) and respective sequence-structure pattern tRNA-pat (bottom).

$\mathcal{K} = d$	#matches	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
0	1	1,582.03	21.81	0.53	3.09
1	168	1,581.86	50.36	2.53	3.81
2	900	1,643.86	68.26	5.95	13.17
3	3,050	1,670.71	100.22	16.22	30.29
4	9,274	1,710.75	141.12	42.23	43.66
5	28,603	1,759.80	196.09	90.61	64.74
6	77,805	1,830.33	319.32	198.94	107.63

Table 4.1: Times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to match in RFAM10.1 the single RSSP describing the consensus secondary structure of the tRNA (Acc.: RF00005). Times are influenced by the cost threshold $\mathcal{K}$ and the number of allowed indels d.

$\mathcal{K}$	d	#matches	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
6	0	10,516	1,536.08	123.18	17.69	22.82
6	1	30,633	1,576.73	156.50	35.67	39.87
6	2	49,287	1,657.61	188.79	58.99	52.98
6	3	64,226	1,703.31	222.36	86.39	65.94
6	4	74,146	1,754.08	256.78	119.47	80.55
6	5	77,679	1,808.84	287.49	156.48	94.03
6	6	77,805	1,830.33	319.32	198.94	107.63

Table 4.2: Search times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to match in RFAM10.1 the single RSSP describing the consensus secondary structure of the tRNA (Acc.: RF00005). Here, the cost threshold $\mathcal{K}$ is constant and the number of allowed indels d increases progressively.

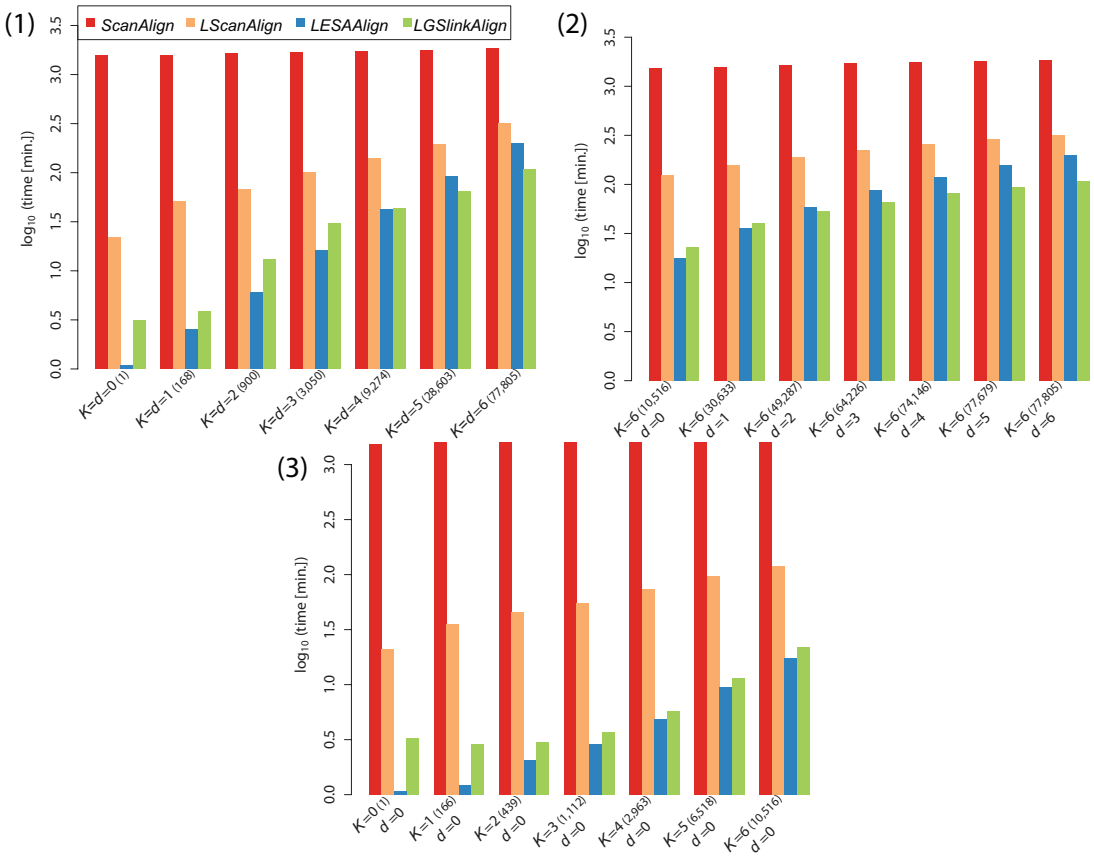


Figure 4.10: Running times (in minutes and $\log_{10}$ scale) needed by the different algorithms to search with an RSSP describing the tRNA in RFAM10.1. In (1) the cost threshold $\mathcal{K}$ and the number of allowed indels d are identical. In (2) $\mathcal{K} = 6$ is constant and d ranges from 0 to 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.

$\mathcal{K}$	d	#matches	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
0	0	1	1,582.03	21.81	0.53	3.09
1	0	166	1,601.82	35.39	1.21	2.86
2	0	439	1,601.20	45.20	2.05	3.00
3	0	1,112	1,601.90	54.83	2.87	3.72
4	0	2,963	1,606.61	74.29	4.90	5.71
5	0	6,518	1,601.01	96.93	9.53	11.57
6	0	10,516	1,601.93	118.26	17.34	21.87

Table 4.3: Search times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to match in RFAM10.1 the single RSSP describing the consensus secondary structure of the tRNA (Acc.: RF00005). Here, no indels are allowed and the cost threshold $\mathcal{K}$ increases progressively.

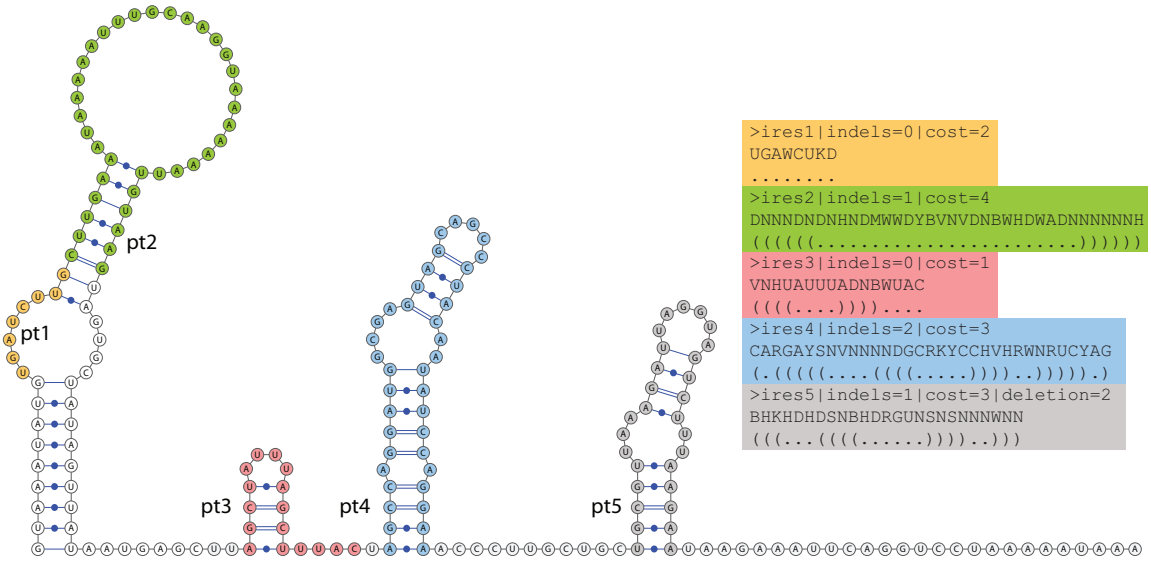


Figure 4.11: Consensus secondary structure of family Cripavirus internal ribosome entry site (Acc.: RF00458) showing its four characteristic stem-loop substructures pt2, pt3, pt4, and pt5 and the moderately conserved strand pt1 as drawn by *VARNA* [128]. The secondary structure descriptor (SSD) for this family, on the right-hand side, consists of five RSSPs ires1, ires2, ires3, ires4, and ires5 describing the strand and stem-loop substructures.

This time we set (1) $\mathcal{K} = d$ varying the values in the interval $[0, 7]$, (2) $\mathcal{K} = 7$ varying d in the interval $[0, 7]$, and (3) $d = 0$ varying $\mathcal{K}$ in the interval $[0, 7]$. We use RSSP $\mathcal{Q} =$ (CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG,(.(((((....(((((.....)))))..))))))) of length $m = 33$ describing a stem-loop substructure of Rfam family Cripavirus internal ribosome entry site (Acc.: RF00458) [136]. The secondary structure of this family and the substructure originating the pattern can be visualized in Figure 4.11, where the substructure is denoted pt4. For the results of this experiment, see Figure 4.12 and Tables 4.4, 4.5, and 4.6. *LGSlinkAlign* is the fastest algorithm with measured speedup factors over *ScanAlign* (*LScanAlign*) in the range of 160.6 for $\mathcal{K} = d = 0$ to 3.3 for $\mathcal{K} = d = 7$ (17.8 for $\mathcal{K} = d = 1$ to 3.3 for $\mathcal{K} = d = 7$). In a comparison between the two online algorithms, *LScanAlign* is faster than *ScanAlign* up to a cost threshold of $\mathcal{K} = 6$ and for any value of $\mathcal{K}$ in case no indels are allowed, i.e. $d = 0$. *LESAAlign* is only faster than the online algorithms for up to $(\mathcal{K} = d) \leq 5$ and $\mathcal{K} = 7$ and $d \leq 3$. For higher cost thresholds and allowed indels, its performance decreases significantly. We explain this behavior with the increased reading depth in the suffix array implicated by $\mathcal{K}$ and d and the reduced number of suffixes sharing a common prefix that can be skipped.

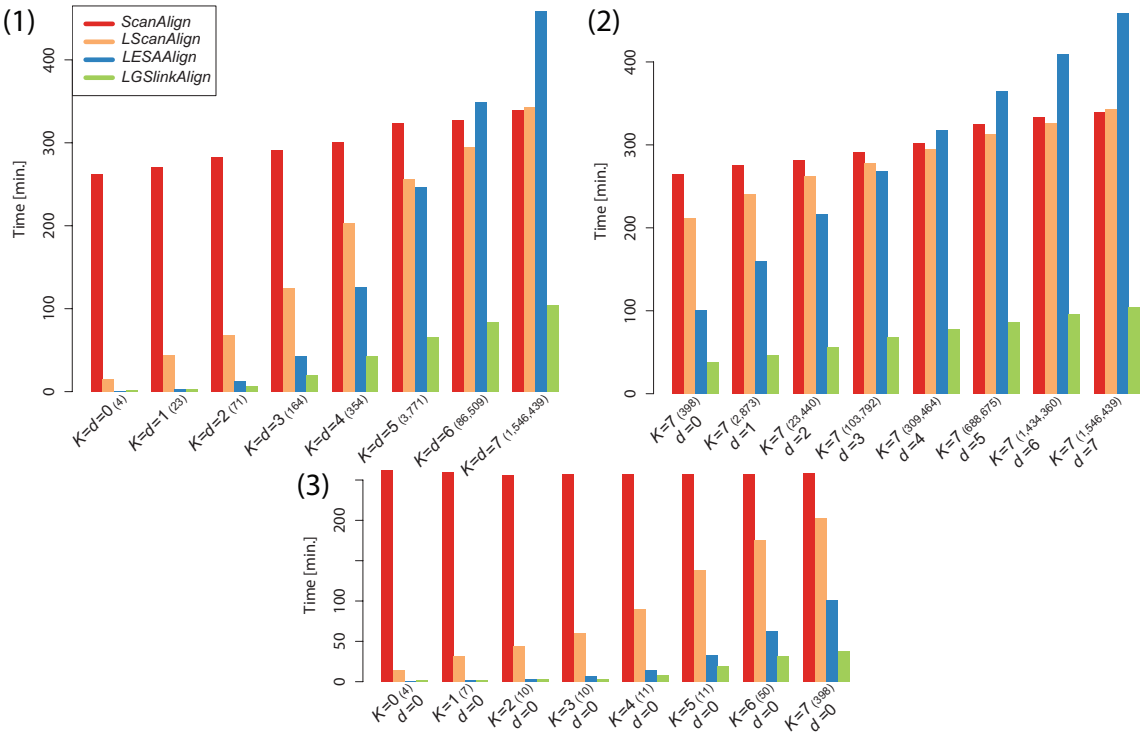


Figure 4.12: Running times needed by the different algorithms to search with a stem-loop pattern of length 33 in RFAM10.1. In (1) the cost threshold $\mathcal{K}$ and the number of allowed indels d increase equally. In (2) $\mathcal{K} = 7$ is constant and d increases from 0 to 7. In (3) $d = 0$ is constant and $\mathcal{K}$ increases from 0 to 7. The numbers of resulting matches are given on the x-axes in brackets.

$\mathcal{K} = d$	#matches	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
0	4	261.80	14.60	0.56	1.63
1	23	270.43	43.83	3.62	2.46
2	71	282.99	67.87	12.39	6.06
3	164	291.66	124.34	43.19	19.90
4	354	300.93	203.04	125.74	43.01
5	3,771	323.66	256.16	246.60	66.25
6	86,509	326.85	294.44	348.69	83.96
7	1,546,439	339.55	342.66	459.26	104.08

Table 4.4: Times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to search with a stem-loop pattern of length 33 in RFAM10.1. Times are influenced by the cost threshold $\mathcal{K}$ and the number of allowed indels d. For a graphical representation of the measurements, see Figure 4.12(1).

$\mathcal{K}$	d	#matches	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
7	0	398	264.68	211.03	101.11	37.93
7	1	2,873	275.95	240.55	159.24	45.92
7	2	23,440	281.80	262.52	216.26	56.46
7	3	103,792	290.80	278.29	268.33	68.04
7	4	309,464	302.53	295.05	317.23	78.01
7	5	688,675	325.10	313.40	364.49	86.72
7	6	1,434,360	333.96	325.80	409.53	95.51
7	7	1,546,439	339.55	342.66	459.26	104.08

Table 4.5: Times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to search with a stem-loop pattern of length 33 in RFAM10.1. Here, the cost threshold $\mathcal{K}$ is constant and the number of allowed indels d increases progressively.

$\mathcal{K}$	d	#matches	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
0	0	4	261.80	14.60	0.56	1.63
1	0	7	259.17	31.45	1.78	1.86
2	0	10	256.27	44.37	3.10	2.52
3	0	10	257.19	60.41	6.43	3.57
4	0	11	257.01	90.22	14.52	8.07
5	0	11	257.61	138.30	33.11	18.90
6	0	50	257.50	176.02	63.11	31.45
7	0	398	258.00	202.68	100.82	37.45

Table 4.6: Times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to search with a stem-loop pattern of length 33 in RFAM10.1. Here, indels are not allowed and the cost threshold $\mathcal{K}$ increases progressively.

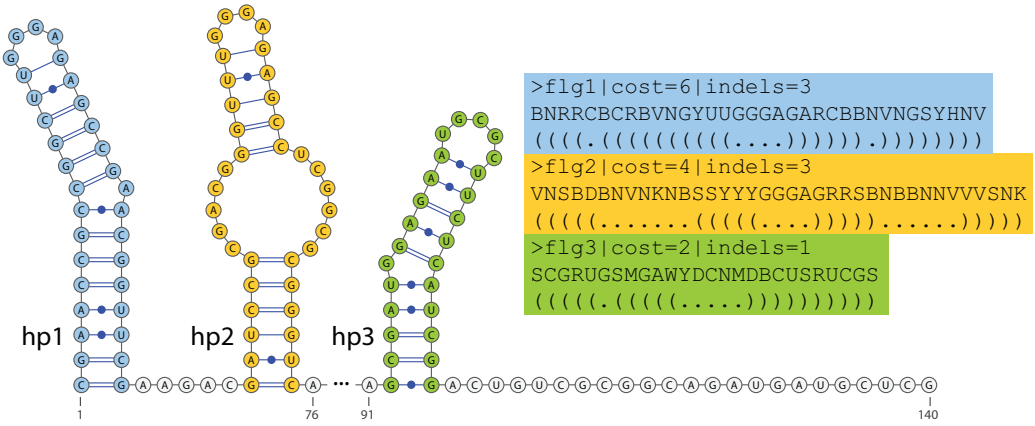


Figure 4.13: Consensus secondary structure of family flg-Rhizobiales RNA motif (Acc.: RF01736) showing its three stem-loop substructures hp1, hp2, and hp3 as drawn by *VARNA* [128]. The secondary structure descriptor (SSD) for this family, on the right-hand side, consists of three RSSPs flg1, flg2, and flg3 derived from the stem-loop substructures.

Scaling behavior of the online and index-based algorithms

In a third 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) [137] fold into. These patterns differ in length, cost threshold $\mathcal{K}$ and number of allowed indels d; see Figure 4.13 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 4.14 and Table 4.7. *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.

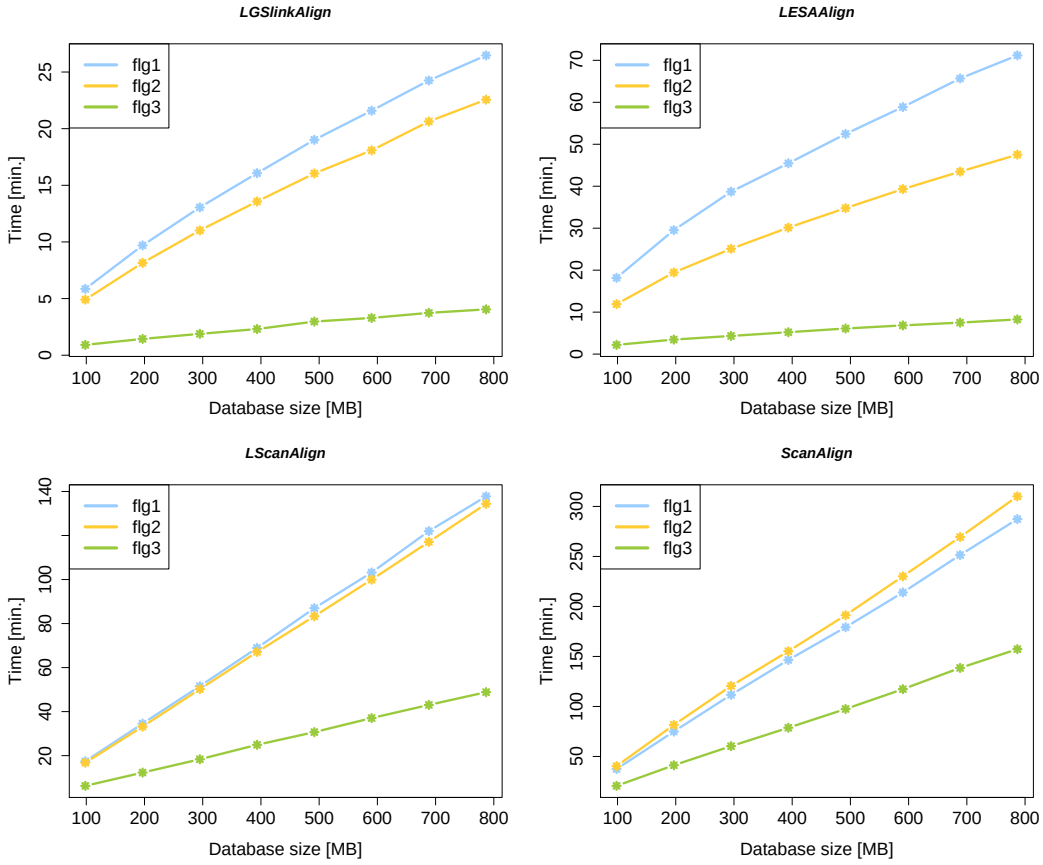


Figure 4.14: Scaling behavior of algorithms *LGSlinkAlign*, *LESAAlign*, *LScanAlign*, and *ScanAlign* when searching with RSSPs flg1, flg2, and flg3 in subsets of RFAM10.1 of different length. For details, see main text.

RFAM10.1 subset	*ScanAlign*			*LScanAlign*			*LESAAlign*			*LGSlinkAlign*		
size (MB)	flg1	flg2	flg3	flg1	flg2	flg3	flg1	flg2	flg3	flg1	flg2	flg3
98.3	37.42	40.32	20.61	17.42	16.78	6.27	18.17	11.90	2.20	5.86	4.91	0.91
196.7	74.91	81.51	41.21	34.55	33.18	12.29	29.53	19.45	3.46	9.69	8.15	1.44
295.0	111.63	120.53	60.29	51.54	50.20	18.35	38.70	25.09	4.33	13.05	11.01	1.89
393.4	146.50	155.24	78.69	68.93	67.10	24.92	45.46	30.13	5.21	16.07	13.57	2.31
491.7	179.22	191.24	97.46	87.00	83.32	30.68	52.46	34.78	6.10	19.01	16.05	2.97
590.1	213.99	230.11	117.29	103.18	99.96	37.06	58.87	39.32	6.84	21.59	18.08	3.29
688.4	251.42	269.40	138.52	121.99	117.08	43.04	65.68	43.48	7.50	24.26	20.64	3.74
786.8	287.32	310.06	157.28	137.78	134.40	48.83	71.18	47.52	8.27	26.47	22.56	4.05

Table 4.7: Search times in minutes used to investigate the scaling behavior of algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* on random subsets of RFAM10.1 of increasing size. See the definition of the searched RSSPs flg1, flg2, and flg3 in Figure 4.13 and further details of this experiment on the main text.

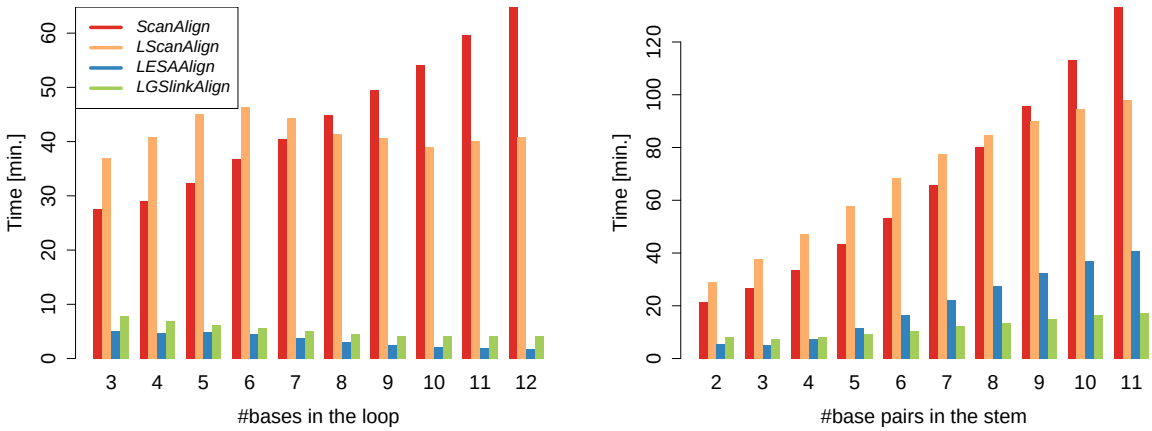


Figure 4.15: Search times for different number of bases in the loop (left-hand side) and base pairs in the stem (right-hand side) for given RSSPs.

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 $\mathcal{Q} = (\text{NNNACANNN}, (((...)))$ from 3 to 12 by using As and Cs. Additionally, we vary the number of base pairs in the stem of pattern $\mathcal{Q} = (\text{NNACANN}, ((...)))$ 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 (4.7) and (4.8), respectively. The cost threshold and the number of allowed indels 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 4.15. 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.

Comparisons between *RaligNAtor* and *RNAMotif* in terms of sensitivity and specificity

RNAMotif [42] is one of the most popular tools for approximate matching of RSSPs supporting the operations replacement and mispairing (which corresponds to the arc breaking operation defined

above). A number of allowed replacements and mispairings, which we here simply denote *errors*, can be specified for each part of the structure along with an overall number constraining the entire structure. However, the arc altering and arc removing operations are not supported. Also, insertions and deletions are only supported by using regular expression quantifiers. This means that the user has to know in advance for which positions of the pattern such operations can occur.

In this experiment we first analyze the results of *RaligNAtor* when searching RFAM10.1 with the tRNA (Acc.: RF00005) RSSP shown in Figure 4.9. In particular, we show the importance of secondary structure information incorporated in the search for homologous sequences by varying the cost of edit operations on base pairs. Secondly, we compare the results obtained by *RaligNAtor* with the results of *RNAMotif* version 3.07 when searching with an equivalent *RNAMotif* pattern. For the used *RNAMotif* descriptor, see Figure 4.16.

For the searches with *RaligNAtor*, we vary the cost threshold $\mathcal{K}$ and the number of allowed indels d between 0 and 25 in steps of 5. We use operation costs $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = \omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 1$ and $\omega_{\mathrm{r}} = 2$. Then we increase the costs of the operations arc breaking, arc altering, and arc removing. More precisely, we set $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = 1$, $\omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 2$, and $\omega_{\mathrm{r}} = 3$. The results are shown in Table 4.8. Unsurprisingly, we observe that *RaligNAtor*'s sensitivity increases with increasing values of $\mathcal{K}$ and d. However, for low costs of the operations on base pairs, its specificity decreases considerably when $\mathcal{K}$ and d are increased from 20 to 25. For high costs of these operations, *RaligNAtor* is sensitive while maintaining a high specificity.

To search with *RNAMotif*, we vary the number of allowed errors per substructure between 0 and 25 in steps of 5, constraining the total number of errors to this same number. This means that no indels are allowed, since this requires many different patterns specifying possible indels only for specific pattern positions. The results are shown in Table 4.8. *RNAMotif* is highly specific for the complete range of allowed indels, but it is not as sensitive as *RaligNAtor*. Notably, unlike in the search with *RaligNAtor*, its sensitivity only marginally increases when the number of allowed errors varies from 20 to 25, with some decrease of its specificity. Similar results can be obtained with *RaligNAtor* by setting $d = 0$.

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) [137]. 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 4.13, 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

RaligNAtor, edit operation costs: $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = \omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 1$, $\omega_{\mathrm{r}} = 2$

$\mathcal{K} = d$	#TP	#FP	#FN	Sensitivity	Specificity	Accuracy	Precision
0	1	0	1,101,832	0.000	1.000	0.600	1.000
5	10,726	0	1,091,107	0.010	1.000	0.606	1.000
10	146,124	3	955,709	0.133	1.000	0.671	1.000
15	517,984	65	583,849	0.470	1.000	0.822	1.000
20	959,243	164,708	142,590	0.871	0.941	0.921	0.853
25	1,097,783	1,168,140	4,050	0.996	0.702	0.767	0.484

RaligNAtor, edit operation costs: $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = 1$, $\omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 2$, $\omega_{\mathrm{r}} = 3$

$\mathcal{K} = d$	#TP	#FP	#FN	Sensitivity	Specificity	Accuracy	Precision
0	1	0	1,101,832	0.000	1.000	0.600	1.000
5	10,427	0	1,091,406	0.009	1.000	0.606	1.000
10	127,865	2	973,968	0.116	1.000	0.662	1.000
15	263,277	8	838,556	0.239	1.000	0.722	1.000
20	669,252	262	432,581	0.607	1.000	0.874	1.000
25	1,034,028	122,285	67,805	0.938	0.956	0.951	0.894

RNAMotif

#Errors	#TP	#FP	#FN	Sensitivity	Specificity	Accuracy	Precision
0	1	0	1,101,832	0.000	1.000	0.600	1.000
5	7,289	0	1,094,544	0.007	1.000	0.604	1.000
10	40,669	0	1,061,164	0.037	1.000	0.621	1.000
15	66,451	1	1,035,382	0.060	1.000	0.633	1.000
20	68,236	1	1,033,597	0.062	1.000	0.634	1.000
25	68,492	139	1,033,341	0.062	1.000	0.634	0.998

Table 4.8: Results of the searches in RFAM10.1 for the tRNA (Acc.: RF00005). For the two series of searches with *RaligNAtor* using the operation costs above, the sequence-structure pattern shown in Figure 4.9 is used. For the searches with *RNAMotif* varying the number of allowed errors (#Errors), the descriptor shown in Figure 4.16 is used. These errors comprehend replacements and mispairings. #TP, #FP, and #FN stand for number of true positives, false positives, and false negatives, respectively. Sensitivity is computed as $\frac{\#TP}{\#TP+\#FN}$, specificity as $\frac{\#TN}{\#TN+\#FP}$, accuracy as $\frac{\#TP+\#TN}{\#TP+\#FP+\#FN+\#TN}$, and precision as $\frac{\#TP}{\#TP+\#FP}$. For additional details, see text above.

```
parms
    wc += gu;

descr
    h5(seq="^GSSVVYR$")
      ss(seq="^UR$")
      h5(seq="^GYYY$")
        ss(seq="^ARYUGGUUA$")
      h3(seq="^RMRC$")
      ss(seq="^R$")
      h5(seq="^YYDSV$")
        ss(seq="^YUBHHAM$")
      h3(seq="^BCHRD$")
      ss(seq="^WRRUY$")
      h5(seq="^RYRGG$")
        ss(seq="^UUCRAWU$")
      h3(seq="^CCYDY$")
    h3(seq="^HNBBNSY$")
    ss(seq="^R$")
```

Figure 4.16: *RNAMotif* descriptor without errors for the tRNA.

ires5 in Figure 4.11, where the last four RSSPs describe the stem-loop substructures the members of this family fold into. ires1 describes a moderately conserved strand occurring in these members. Observe also in Figures 4.13 and 4.11 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 4.9 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 4.10.

RSSP	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
ires1	13.13	12.85	1.02	2.68
ires2	203.67	356.78	135.03	60.12
ires3	51.21	8.54	0.37	1.61
ires4	281.44	103.52	28.11	14.53
ires5	138.86	103.90	22.35	13.31

Table 4.9: Times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to match the RSSPs that build the SSD for family Cripavirus internal ribosome entry site (Acc.: RF00458) in RFAM10.1.

RSSP	*ScanAlign*	*LScanAlign*	*LESAAlign*	*LGSlinkAlign*
flg1	288.21	143.23	74.90	27.03
flg2	310.68	141.73	50.01	22.00
flg3	156.60	51.74	8.67	3.83

Table 4.10: Times in minutes required by algorithms *ScanAlign*, *LScanAlign*, *LESAAlign*, and *LGSlinkAlign* to match the RSSPs that build the SSD for family flg-Rhizobiales RNA motif (Acc.: RF01736) in RFAM10.1.

Comparison with *RSEARCH* and *ERPIN* when searching a bacterial genome

We compare *RaligNAtor*'s performance when searching a bacterial genome using local chaining with the performance of the well-known tools *RSEARCH* [74] and *ERPIN* [73, 41]. Also refer to the section in Chapter 2 where *RSEARCH* and *ERPIN* are explained. We search the complete forward strand of the 5.2 MB genome of Rhodopseudomonas palustris HaA2 (EMBL Acc.: CP000250) for an occurrence of family RF01736. For the search with *RaligNAtor*, we reuse the SSD shown in Figure 4.13 of the previous experiment. To indicate the position of the RNA substructure modeled by each RSSP within the molecule, we set option *startpos* of RSSPs flg1, flg2, and flg3 to 1, 39, and 92, respectively. For more details about this option, which is used to score local chains, see Chapter 3 and *RaligNAtor* manual in Appendix B.3. For the search with *RSEARCH*, we compute the consensus sequence from the family's seed alignment and use the consensus secondary structure given by Rfam. For the scoring of the computed alignments, we arbitrarily choose the RIBOSUM85 scoring matrix provided with *RSEARCH*, where 85 is the percent identity of the sequences used for the computation of this matrix [74]. For the search with *ERPIN*, we use the structure-annotated seed alignment of the family preprocessed with program *parent2epn.pl*. This program, which is provided with *ERPIN*, transforms paired positions aligned with gaps into unpaired positions, since the secondary structure profile built by *ERPIN* only supports gaps at unpaired positions (see Chapter 2). Searching with *RaligNAtor* results in 94 matches for flg1, 763 for flg2, and 5 for flg3. After searching with each RSSP individually, *RaligNAtor* reports a single local chain of score 112 at the correct location of the RNA in the genome. Using algorithm *LGSlinkAlign*, the total search time is 30 seconds. *RSEARCH*, which uses a different scoring system, reports hundreds of locations for the

given structure, with the correct location (with score 13.01) being on rank 217, i.e. there are 216 locations achieving a better score than the correct location. Besides the low specificity compared to *RaligNAtor*, it is also orders of magnitudes slower, requiring 32.3 (user time) or 146.8 (real time) hours of running time. The cause of the disparity between the user and real times is unclear to us. *ERPIN* requires 23.8 hours of running time and reports the single correct location of the RNA in the genome. We note that *ERPIN* allows to specify which positions of the query alignment are matched first, so that other positions are only matched if previously matched positions achieve a certain score cutoff (see *ERPIN* manual [138]). By specifying the positions corresponding to substructures hp1, hp2, and hp3 shown in Figure 4.13 as initial positions, the running time of *ERPIN* reduces to 10.5 minutes, which is still 21 times slower than *RaligNAtor*.

RNA family classification using *Structator*

Structator (see previous chapter) is an ultra fast tool for RSSP matching. It is the first tool to integrate algorithms for global and local chaining of RNA pattern matches. However, it has limited support to approximate matching, lacking support of the sequence-structure edit operations allowed by *RaligNAtor*.

Here, we report the number of sequence members obtained by *Structator* when searching RFAM10.1 with the SSDs of families RF00458 and RF01736. The SSDs are shown in Figures 4.11 and 4.13. Despite sharing the same pattern syntax with *RaligNAtor*, we observe the following differences and adaptations.

- *Structator* cannot search for stem-loop patterns with dangling ends. Therefore, we remove the dangling end of the RSSP ires3 belonging to the SSD of family RF00458.
- As *Structator* does not allow for edit operations, parameters *cost* and *indels* have no effect in the search. However, a number of allowed mispairings for each pattern can be specified by the user. We allow for each pattern a number of mispairings equal to the value of parameter *cost*.
- *Structator* has lower sensitivity compared to *RaligNAtor* when the latter searches with allowed costs greater than zero. For this reason, we chain the matches to the single RSSPs varying the minimum required chain length between 2 and the total number of RSSPs of each SSD.

The results are shown in Table 4.11. We observe that, in the search with the SSD of family RF00458, *Structator* cannot find all its true sequence members without increasing considerably the number of false positives. In the search with the SSD of family RF01736, only up to 4 true sequence members can be found. *RaligNAtor*, in contrast, finds all sequence members of both families and no false positives as described in our experiment above performing chaining of matches.

RF00458			
Min. chain length	#TP	#FP	#FN
2	16	5807	0
3	16	14	0
4	14	0	2
5	3	0	13

RF01736			
Min. chain length	#TP	#FP	#FN
2	4	0	11
3	1	0	14

Table 4.11: Results obtained with *Structator* [104] when searching with the secondary descriptors of families RF00458 and RF01736 in RFAM10.1. The first column for each family indicates the minimum required length of a chain to be considered a matching chain. #TP, #FP, and #FN stand for number of true positives, false positives, and false negatives, respectively. For additional details, see text above.

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 4.12 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* [35] 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 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 $\omega_{\mathrm{d}} = \omega_{\mathrm{m}} = 1$, $\omega_{\mathrm{b}} = \omega_{\mathrm{a}} = 2$, and $\omega_{\mathrm{r}} = 3$. 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 4.12 and Figure 4.17 for the results of this experiment. A ROC curve with values averaged over all families is shown in Figure 4.17(1). In addition, we show in Figures 4.17(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 4.18 and 4.19. Clearly, by using base pairing information, *RaligNA-*

tor 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 4.12). 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 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 4.17 (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 4.12). 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.

4.5 *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 4.1, 4.9, 4.11, and 4.13. 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 transforma-

			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)
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)

Table 4.12: Results of *RaligNAtor* and *blastn* database searches for members of RNA families of different degrees of sequence identity in RFAM10.1. 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 4.17, 4.18, and 4.19. Column pAUC gives the partial area under the curve up to a false positive rate of 20%. For additional details, see main text.

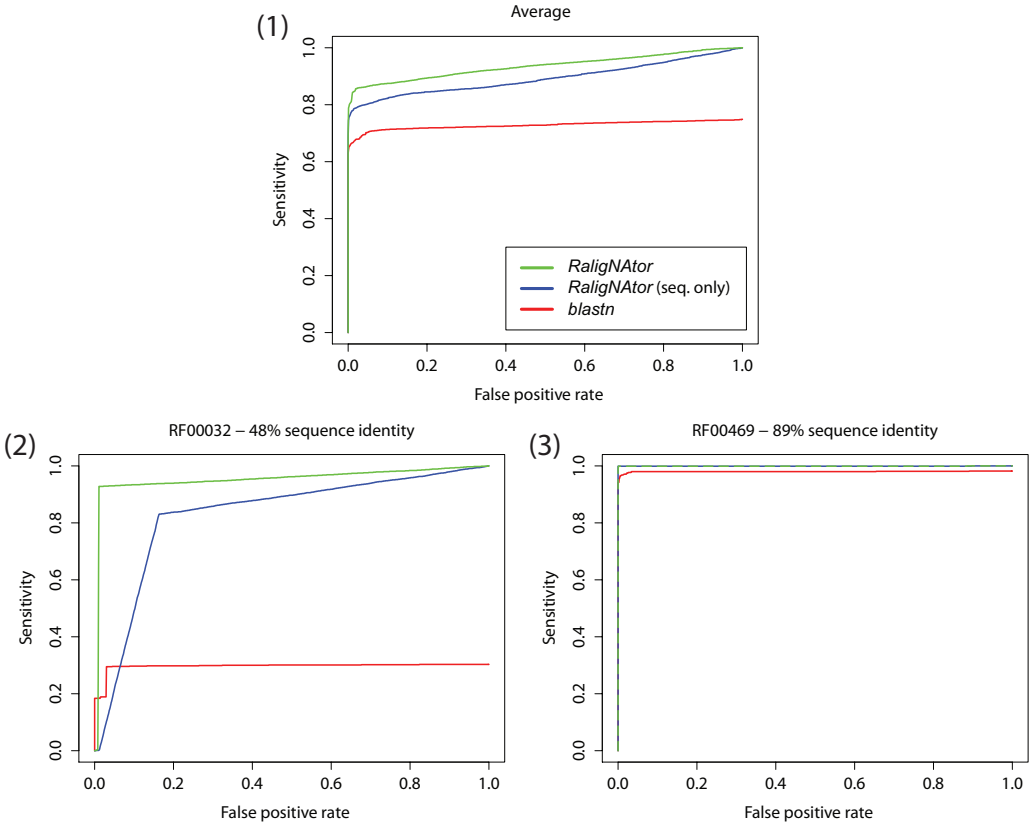


Figure 4.17: Results of ROC analyses using *RaligNAtor* with and without base pairing information and *blastn* for the 35 selected Rfam families shown in Table 4.12. ROC curves showing *RaligNAtor*'s classification performance using (ignoring) base pairing information are shown in green (blue). Blast 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.

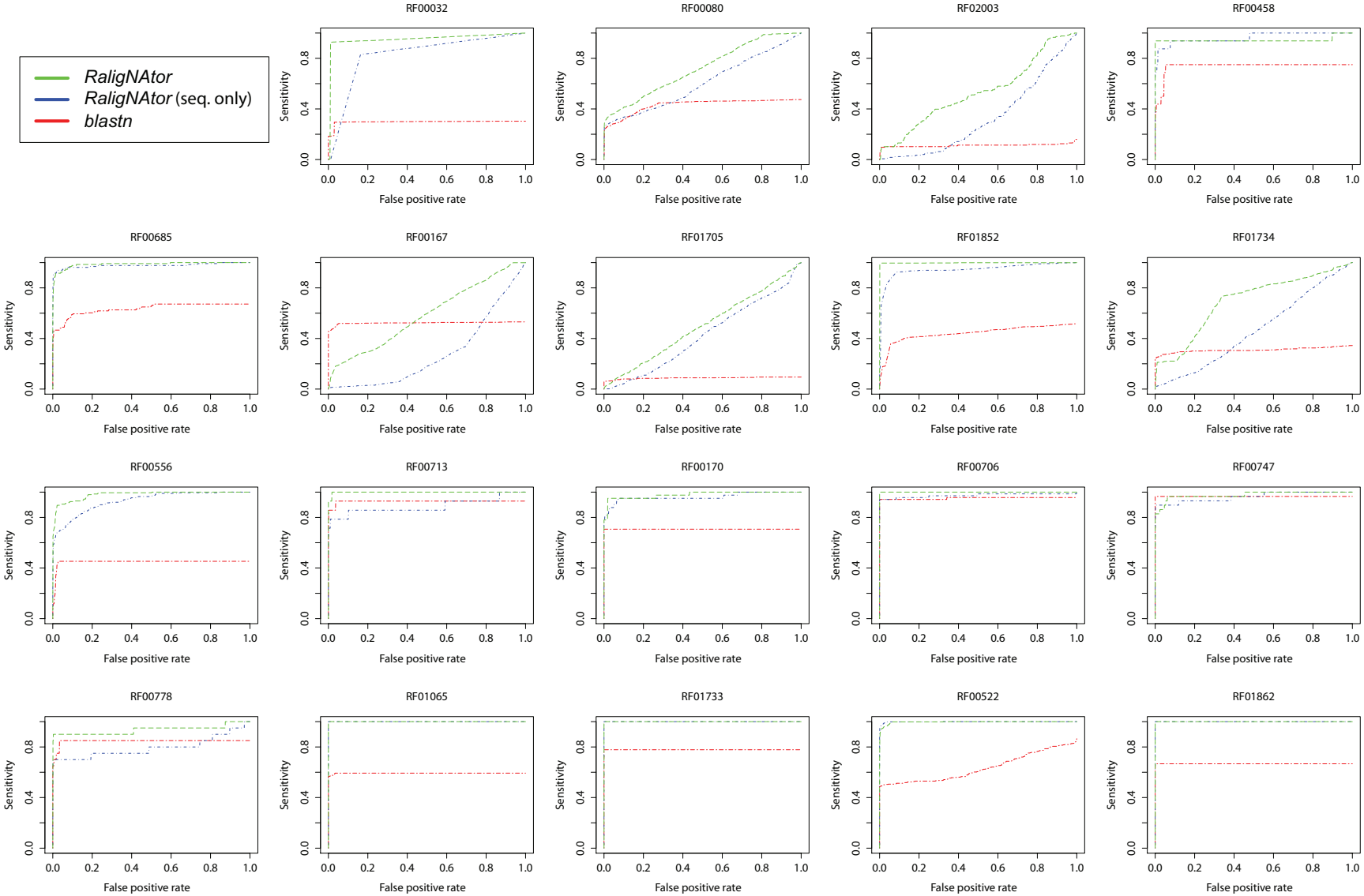


Figure 4.18: Results of ROC analyses using *RaligNAtor* with and without base pairing information and *blastn* [35] for the Rfam families shown in Table 4.12. ROC curves showing *RaligNAtor*'s classification performance using (ignoring) base pairing information are shown in green (blue). Blast performance results are shown in red. The ROC curves are sorted by increasing level of sequence identity of the respective family, i.e. in the same order each family is listed in Table 4.12. Additional ROC curves are shown in Figure 4.19. For details of this experiment, see corresponding text.

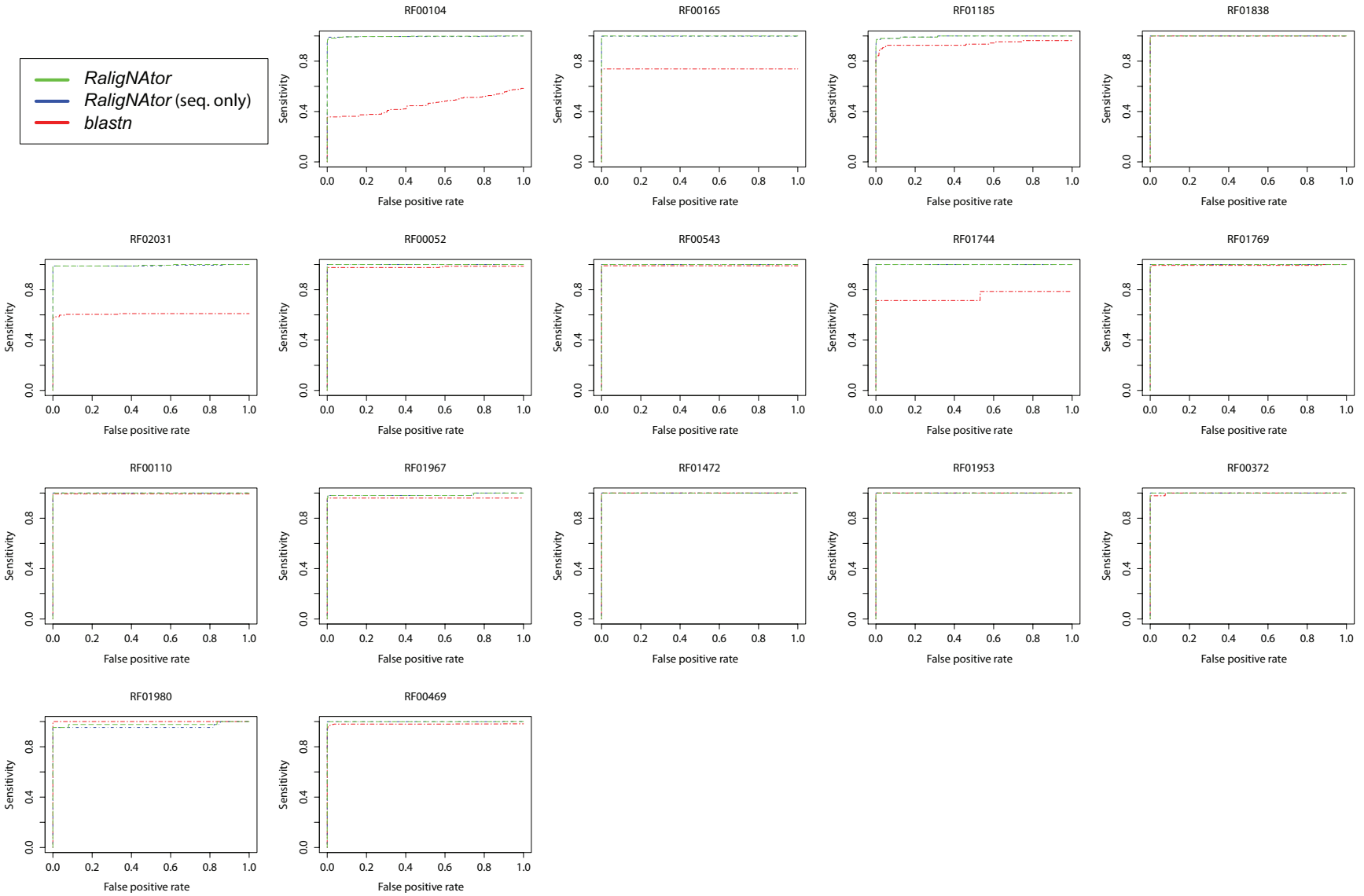


Figure 4.19: Additional ROC curves. See description of Figure 4.18 for details.

tion 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 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. Lastly, we remark that our software also provides an implementation of the original algorithm of Jiang *et al.* for global sequence-structure alignment [70], easily applicable by the user.

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.

4.6 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 4.13 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 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 4.10). 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

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		✓	✓	$9.125n$	✓	✓	✓	✓

Table 4.13: Overview of the presented algorithms. 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).

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 [104], 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 [74] or energy based scoring [139], 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 work.

4.7 Further techniques integrated in the *RaligNAtor* software for search acceleration

To further accelerate the algorithms in the *RaligNAtor* software, we apply to the algorithms two general techniques which can be enabled and disabled by the user. An evaluation of these techniques is given following their description.

4.7.1 Sequence-based filter acceleration

Our search algorithms computing the sequence-structure edit distance $dist(\mathcal{Q}, T)$ of an RSSP $\mathcal{Q}$ and a substring T of a target sequence S with Equations 4.7, 4.8, and 4.9 require $O(mm'^3)$ time and use $m'+1$ DP matrices, recalling that m and m' are the length of $\mathcal{Q}$ and T, respectively. Searching S of length n considering up to n substrings takes $O(nmm'^3)$ time. In practice, our experiments show that our index-based algorithms *LGSlinkAlign* and *LESAAlign* are very fast by exploiting repetitions of substrings and using techniques like the early-stop computation described above. Nevertheless, we observe that we can further accelerate the search by efficiently preprocessing the search space to eliminate substrings not matching $\mathcal{Q}$. For this, in a first step we search S using only sequence information of the RSSP $\mathcal{Q}$. After this sequence-based filtering step, we search S using both sequence and secondary structure information of $\mathcal{Q}$. We emphasize that, under a proper selection of the edit operation costs, applying the sequence-based filter does not affect the sensitivity of the search algorithms.

More precisely, we implement our sequence-based filter by using a cost threshold $\mathcal{K}$ as usual, but considering all positions of the given RSSP $\mathcal{Q}$ as unpaired. This allows to search S using only Equation 4.7 and computing for each aligned substring T a single DP matrix of size of $m \times m'$. This means that we compute sequence edit distances, hence ignoring structural edit operations. Consequently, this search takes only $O(nmm')$ time instead of $O(nmm'^3)$ if structure information is considered. After a substring T of S is aligned to $\mathcal{Q}$, only if the obtained sequence edit distance is below $\mathcal{K}$ then, in a second step, it is realigned to $\mathcal{Q}$ to obtain the sequence-structure edit distance $dist(\mathcal{Q}, T)$. Observe that the edit operations arc altering and arc removing involve one and two indel operations, respectively. Therefore, to prevent the elimination from the search space of possible matches to the RSSP with structure information, we must set the cost ω_{d} of an indel as at most the cost ω_{a} of an arc altering and as at most two times the cost ω_{r} of an arc removing. That is, $\omega_{\mathrm{d}} \leq \omega_{\mathrm{a}}$ and $2{*}\omega_{\mathrm{d}} \leq \omega_{\mathrm{r}}$ must hold. This guarantees that the sequence edit distance of each aligned substring T to $\mathcal{Q}$ will not exceed $dist(\mathcal{Q}, T)$, since the constraints imposed by the base pairs can only lead to edit operations increasing the edit distance. To also accelerate the filtering step in the index-based algorithms, we avoid the computation of matrix entries by using information of the lcp table and the early-stop computation technique. These techniques are applied as described above to the single used DP matrix. As of the writing of this work, the sequence-based filter is only integrated in algorithm *LGSlinkAlign*, but it can easily be integrated in all our online and index-based algorithms.

4.7.2 Multithreaded searching

To take advantage of computer systems with multiple CPU cores, all search algorithms implemented in the *RaligNAtor* software support multithreading (POSIX threads). There are two modes of parallelism. At first, different patterns are matched using each one thread. Additionally, the

search space (i.e. the sequence for the online algorithms and the index structure for the index-based methods) is partitioned, being each part processed by a different thread.

The first mode of parallelism is particularly useful for searching with SSDs composed of multiple RSSPs, such as the SSDs shown in Figures 4.11 and 4.13. The number of simultaneously executed threads, which corresponds to the number of simultaneously searched patterns, can be specified by the user or be automatically defined as the number of available CPU cores. Thus, specifying a number of threads at least as large as the number of given RSSPs means that all RSSPs will be simultaneously searched for. In the case of a smaller number of threads, the patterns are put in a queue and each is searched for as soon as a thread slot becomes available. Ideally, the number of specified threads should not exceed the number of available CPU cores, since this causes the threads to compete for CPU time.

In this first mode of parallelism, we observe that a number of threads larger than the number of patterns does not further accelerate the search. In addition, searching with a single RSSP cannot be accelerated at all. This is overcome by splitting the search space into k parts and searching each part with a different thread, where k can be defined by the user. With this approach, even searches with a single RSSP can be accelerated by searching different parts of the search space with the same RSSP. In the online algorithms *ScanAlign* and *LScanAlign*, splitting a sequence S of length n into k parts is trivially done by dividing n by k. Therefore, each part j, $1 \leq j \leq k$, has length $n_{\text{part}} = \lfloor n/k \rfloor$ and begins at position $start_j = (j-1) * n_{\text{part}} + 1$ of S. It ends at position $end_j = j * n_{\text{part}} - 1$ if $j < k$. Let $m_{\text{min}} = m - d$ be the minimum length of a possible match, recalling that m is the pattern length and d is the number of allowed indels. For the case that $j = k$, we set $end_k = n - m_{\text{min}}$ to ensure that the last positions of the target sequence S are also searched. We note that a pattern can match a substring overlapping different partitions, e.g. a substring beginning at a partition j of S and ending at partition $j+1$. In the index-based algorithms *LGSlinkAlign* and *LESAAlign*, $start_j$ and end_j refer to positions in the suffix array `suf` instead of direct positions in S. Each thread in algorithm *LESAAlign* traverses the suffix array top down in the interval between given index positions $start_j$ and end_j. In *LGSlinkAlign*, each thread is also assigned an index interval to be processed. However, a thread can also process suffixes whose index in table `suf` belongs to an interval assigned to a different thread due to the use of suffix links. For this reason, as in the single-threaded algorithm, every processed suffix is marked in table `vtab` so that it is processed only once, but in addition the algorithm must synchronize `vtab` among all threads. This is done efficiently by reading and writing to `vtab` as atomic operations.

Both modes of parallelism can naturally be combined to optimize CPU usage. Given p patterns to be searched for in k sequences or index partitions, *RaligNAtor* conveniently queues $p*k$ jobs, where each job is characterized by a pattern and a partition, i.e. a range from $start_j$ to end_j, $1 \leq j \leq k$. Given, in addition, a number t of threads, *RaligNAtor* executes up to t jobs in parallel, starting new jobs from the queue as threads terminate their search. With this combined approach, searching with different patterns in parallel reduces the total practical running time of the algorithms and splitting

the search space better distributes the computation among the CPU cores, further speeding up the search. In particular, splitting the search space reduces the chance that some CPU cores remain idle after being used only for a relatively short time by threads searching with more specific or shorter patterns, while other threads can take longer to terminate by searching with more sensitive or longer patterns in the entire search space (see influence of pattern length, number of allowed indels, and cost threshold on the search time in e.g. Figures 4.12 and 4.15). This situation is avoided by the use of several threads to search with the same pattern in different parts of the search space.

Once all threads terminate, they are joined in the main program thread and found matches can then be chained. We note that, during the search, the matches can immediately be printed out and/or be stored in a temporary container for use in chaining, depending on the user choice. When a match is immediately printed out, the standard output channel (stdout) is blocked to prevent another thread from also printing a match and making the results unreadable. As a consequence, printing a large number of matches can slow down the search. Using a container is in general more sensible, since each thread has its own container which does not have to be synchronized with other threads during the search.

4.7.3 Benchmark experiments

We evaluate our two techniques for additional search acceleration described above on a computer system with four Xeon E5-4640 CPUs running at 2.40 GHz and with 768 GB of main memory. Each CPU has 8 cores, hence there are 32 cores in total (with disabled CPU threading). In our first experiment we use only one CPU core and, posteriorly, exploit up to all cores.

Comparison of times to search with and without sequence-based filtering

In an experiment, we compare the time required by algorithm *LGSlinkAlign* to search RFAM10.1 with and without our sequence-based filter. We use the eight RSSPs shown in Figures 4.11 and 4.13 describing families Cripavirus internal ribosome entry site (Acc.: RF00458) and flg-Rhizobiales RNA (Acc.: RF01736). For these patterns, except ires5, $\omega_{\mathrm{d}} = 1$, $\omega_{\mathrm{a}} = 2$, and $\omega_{\mathrm{r}} = 3$. Therefore, since $\omega_{\mathrm{d}} \leq \omega_{\mathrm{a}}$ and $2 * \omega_{\mathrm{d}} \leq \omega_{\mathrm{r}}$ holds, applying the filter does not eliminate from the search space potential matches to these patterns. Only for ires5, $\omega_{\mathrm{d}} = 2$. The results of this experiment are shown in Figure 4.20. For all but one pattern, namely ires2, the filter considerably accelerates the search. To explain this behavior, note that RSSPs can lose specificity when lacking secondary structure information, as we show in an experiment above assessing the importance of secondary structural constraints for RNA family classification. However, even when the filtering step ignores the secondary structure information of the patterns, most patterns still present some specificity which allows to remove from the search space a large number of substrings whose sequence edit distance to the patterns exceed the respective cost threshold. Pattern ires2, in contrast, is mostly composed of wildcards N, D, and H, which can match any of the bases in $\varphi(\mathrm{N}) = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{U}\}$,

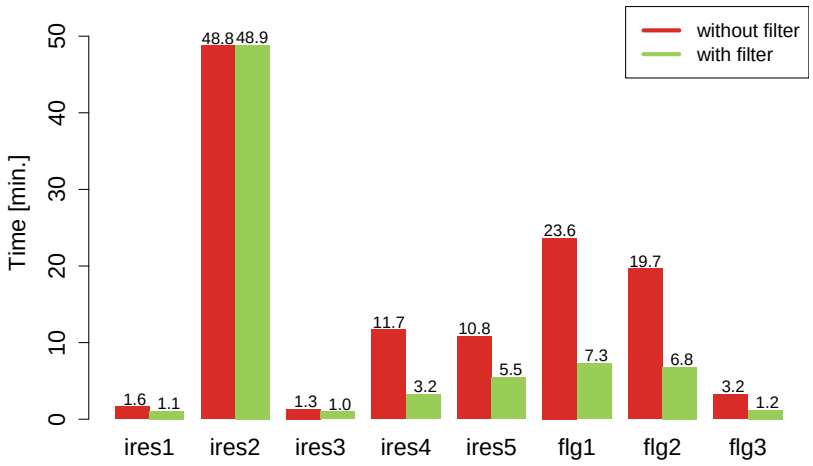


Figure 4.20: Running times of algorithm *LGSlinkAlign* to search RFAM10.1 with and without a sequence-based filter. The used RSSPs, whose names are shown on the x-axis, can be seen in Figures 4.11 and 4.13.

$\varphi(\mathrm{D}) = \{\mathrm{A}, \mathrm{G}, \mathrm{U}\}$, or $\varphi(\mathrm{H}) = \{\mathrm{A}, \mathrm{C}, \mathrm{U}\}$. In addition, it is searched for using a relatively high cost threshold of 4. Consequently, this pattern without structure information is too unspecific and can be poorly exploited the sequence-based filter. While the filter does not accelerate the search for this pattern, we note that it only minimally slows it down. For all other patterns, the speedup factor provided by the filter ranges from 1.3 for pattern ires3 to 3.6 for pattern ires4.

Comparison of running times of multithreaded searches

In another experiment, we compare the running times of *RaligNAtor* to search with an SSD using different numbers of threads and partitions of the target database RFAM10.1. The used SSD, composed of five RSSPs, is shown in Figure 4.11. Although multithreaded searching is possible with all our algorithms, we only report results for *LScanAlign* and *LGSlinkAlign*, which are the fastest online and index-based algorithms in the majority of our experiments described above. For *LGSlinkAlign*, we enable sequence-based filtering.

First, we analyze the speedup obtained by (1) searching with one thread per pattern, thus using 5 threads, and (2) maintaining the number of simultaneously allowed threads, but in addition splitting the search space into two parts. For comparison, we also search using a single thread. The results are shown in Figure 4.21. Using *LGSlinkAlign* (*LScanAlign*), searching with one thread per pattern reduced the search time from 59.6 (455.0) minutes to 49.8 (279.3) minutes compared to single-threaded searching, whereas in addition splitting the search space into two parts reduced the search time to 27.4 (147.4) minutes. That is, *LGSlinkAlign* (*LScanAlign*) searching with 5 simultaneous threads and no search space partitioning required 83% (61%) of the time to search with a single thread, whereas by partitioning the search space it took only 45% (32%) of the time. These results clearly show a benefit of splitting up the search space. Observe in Figure 4.21 that the green area

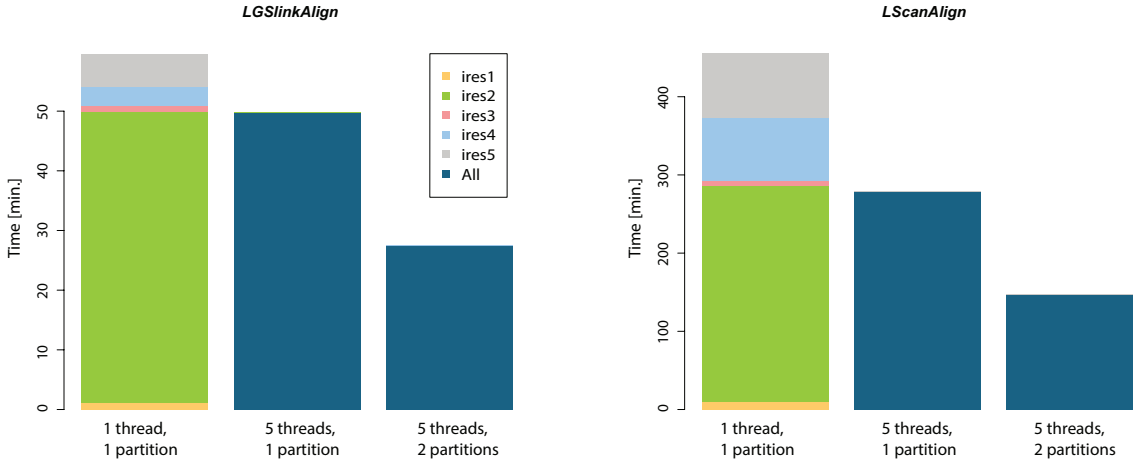


Figure 4.21: Running times of algorithms *LGSlinkAlign* (left-hand side) and *LScanAlign* (right-hand side) to search RFAM10.1 with the RSSPs ires1 to ires5 shown in Figure 4.11. The fist bar in each graph represents the time to sequentially search with each RSSP using a single thread. The other two bars indicate the total time to (1) simultaneously search with one thread per pattern and to (2) search by, in addition, splitting the search space into two parts.

denoting the time to search for pattern ires2 closely corresponds to the area denoting the time to search for all patterns using 5 threads but no partitioning. This indicates that, when the search space is not split, the total search time largely depends on the time to search for pattern ires2. This can occur because ires2 is searched for by a single thread in the entire search space, even though some CPU cores may no longer be used by other threads. In contrast, splitting the search space into two parts allows to search for ires2 with two threads, leading to a larger speedup.

In the second part of our experiment we analyze the speedup obtained by equally increasing the maximum number of simultaneously allowed threads and the number of search space partitions. Beginning with single-threaded searching, we increase the number of threads and partitions to 2 and then increase these in steps of 4 up to 32, which is the total number of CPU cores of the used computer system. The results are shown in Figure 4.22. Both *LGSlinkAlign* and *LScanAlign* profit from the increasing number of threads and partitions. *LGSlinkAlign* (*LScanAlign*) reduces its single-threaded running time of 61.0 (457.5) minutes to 5.5 (21.3) minutes by using 32 threads and partitions. We observe that *LGSlinkAlign* searching with 32 threads and partitions is 11 times faster compared to searching with a single thread, whereas *LScanAlign* is 21 times faster compared to its single-threaded searching time. This can be explained by some overhead computation occurring in *LGSlinkAlign*, where the same thread can process suffixes belonging to different partitions. Nevertheless, *LGSlinkAlign* is about 4 times faster than *LScanAlign* when searching with 32 threads and partitions.

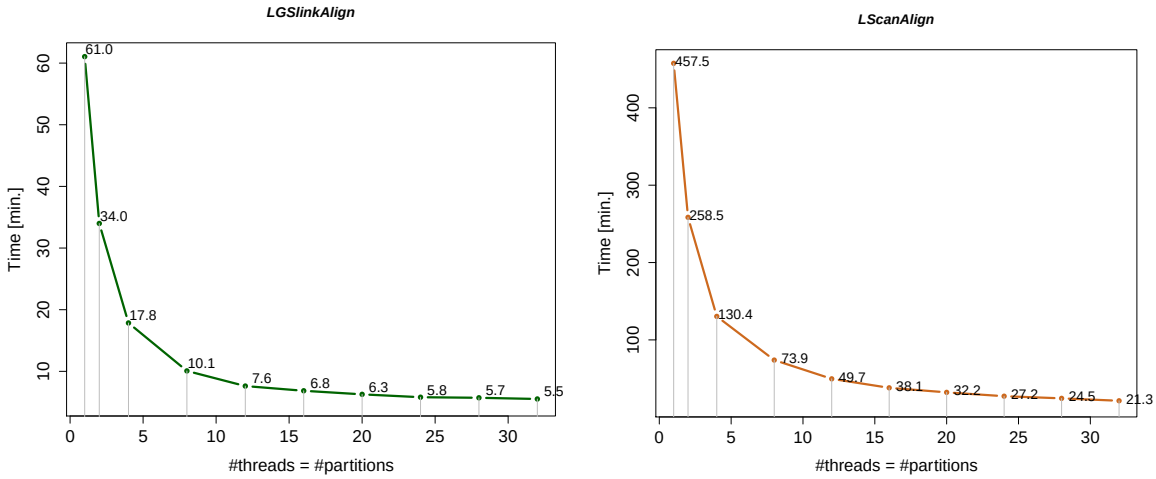


Figure 4.22: Running times of algorithms *LGSlinkAlign* (left-hand side) and *LScanAlign* (right-hand side) to search RFAM10.1 with the SSD shown in Figure 4.11. The number of simultaneously allowed threads and the number of search space partitions used by both algorithms equally vary between 1 and 32.

5 Conclusions and future work

We have presented novel methods for fast online and index-based matching of RNA sequence-structure patterns (RSSPs) in databases, all implemented in the well-documented and readily applicable *Structator* and *RaligNAtor* software packages.

With *Structator*, we have presented the first publicly available tool for bidirectional pattern search using the affix array index data structure. Employing our search algorithm based on affix arrays, called *BIDsearch*, *Structator* was in our experiments much faster than online algorithms and its running time scaled sublinearly in the length of the searched sequences. Compared to the widely known tools *RNAMotif* [42] and *RNABOB* [98], it was faster by up to two orders of magnitude. In addition, compared to the program of [130] working on compressed index data structures, *BIDsearch* yielded a speedup of factor 2. Although the speed of *BIDsearch* can decrease when searching with RSSPs with long unconserved loop regions, it profits from even only a few bases specified in the loop, which are expected to occur in the majority of biological patterns. Also, our detailed complexity analysis shows that bidirectional search using affix arrays does not improve the worst-case time complexity compared to online search. However, the unrealistic pattern for which the worst case occurs consists only of wildcards and no base pairs which reduce the search space. In terms of space consumption, our implementation of the affix array data structure requires $18n$ bytes for a sequence of length n. This is a significant reduction compared to the $\sim 45n$ bytes needed for the affix tree. We note that, with a contribution from [131], *Structator* can also perform bidirectional search using only $10n$ bytes. However, this requires a lazy construction of the affix links needed for switching the search direction. An additional option in *Structator* is to search online with our *ONLsearch* algorithm running in linear time.

To search for RNAs with branching secondary structures, *Structator* uses our new concept of secondary structure descriptors (SSDs) and integrates efficient global and local chaining algorithms. With this concept, we allow an effective description of RNAs containing branching structures like multi-loops by their decomposition into sequences of non-branching substructures that can be described with RSSPs. Combined with the matching of single RSSPs using the affix array, building chains of matches of the RSSPs defined in a SSD eliminates spurious matches and constitutes a very efficient method for RNA homology search.

RaligNAtor is the first tool for RSSP matching that supports a full set of edit operations on both the primary and secondary structure levels. It includes online and index-based algorithms, all which integrate a new computing scheme to optimally reuse the entries of the required dynamic program-

ming matrices. By further integrating in our online algorithm *LScanAlign* a technique to avoid the alignment computation of non-matching substrings, it achieved in our experiments a speedup of factor 45 compared to our most basic online algorithm *ScanAlign*. Our index-based algorithms *LESAAlign* and *LGSlinkAlign*, which operate on enhanced suffix arrays and scale sublinearly in the length of the target sequence, were up to 1,323 and 560 times faster than *ScanAlign*, respectively. Although not as fast as *LESAAlign* searching with tight cost threshold, *LGSlinkAlign* was up to 4 times faster than *LESAAlign* when a larger number of insertions and deletions was allowed. Also, *LGSlinkAlign* was the overall best performing algorithm for a variety of patterns. Concerning the space requirements, *LESAAlign* and *LGSlinkAlign* use $5n$ and $9.125n$ bytes for the enhanced suffix array, respectively.

As *Structator*, *RaligNAtor* allows to use our concept of SSDs by integrating the same efficient global and local chaining algorithms. Even though *RaligNAtor* can search with RSSPs describing branching structures, searching with a SSD composed of a sequence of RSSPs allows to better balance sensitivity, specificity, and running time. Compared to *RNAMotif* in homology search, *RaligNAtor* showed in our experiments higher sensitivity while having similar specificity. Compared to *Structator*, *RaligNAtor* was also more sensitive, although not as fast. Our experiments also showed that *RaligNAtor* is much more sensitive and specific if it uses information of the primary and secondary structure of the sought RNA family compared to using only information of the primary structure, in particular of families with low degree of sequence conservation.

As a further contribution, both our software packages include tools for the efficient construction and persistent storage of enhanced suffix and affix arrays.

5.1 Future work

Several extensions to *Structator* and *RaligNAtor* are possible. We begin observing that both tools can rank chains of matches by e.g. chain length to facilitate the identification of matches that more closely correspond to the sought RNA. However, *Structator* cannot rank matches of single patterns, whereas *RaligNAtor* can only use a relatively simple scoring scheme for ranking, which is sequence-structure edit distance. Therefore, a more fine grained scoring scheme would be desirable. This could be achieved with the incorporation of the concept of secondary structure profiles, for example based on log-odds scores as used in the *ERPIN* tool [73, 41]. Alternatively, one could use position independent scoring matrices such as RIBOSUM [74] or energy based scoring [139].

Another extension to *Structator* would be allowing a sequence edit distance between unpaired positions of the patterns and matched substrings with the goal of increasing the search sensitivity. In this extension, matching the loop region of stem-loop patterns could be performed with standard dynamic programming. Then, pairing positions would be efficiently matched via bidirectional search using the affix array data structure. We note that *ERPIN* also uses standard dynamic programming for matching unpaired positions. However, it does not profit from affix arrays for search accelera-

tion. We also note that, although we would consider this extension an improvement to *Structator*, it still would not have the same flexibility as *RaligNAtor*, since, like *ERPIN*, it does not support edit operations on the secondary structure level.

Finally, to further increase the space efficiency of *Structator* and *RaligNAtor*, compressed index data structures based e.g. on the FM-index [53] could be applied to both tools. However, such structures can slow down the search for structural RNA patterns in databases, as we observed in a comparison between *Structator* and the implementation of bidirectional wavelet index of [130] which can only be used to search with a small set of hard-coded patterns. Unfortunately, the lack of other suitable implementations of compressed structures hampers further investigation of the effect on the running time of using such structures in the analysis of biological sequences. For instance, a general and well-designed implementation of the FM-index is given in [140], but it is not optimized to handle nucleotide sequences with a small alphabet. Also, while the BWA program [141] uses an FM-index to process nucleotide sequences, its ad-hoc implementation of this structure cannot be easily used as a stand-alone software library.

A *Structator* user's manual

A.1 Introduction

Structator is a software package for time efficient matching of RNA sequence-structure patterns in sequence databases. Its main features are:

- Persistent construction of an index data structure of the target database. The index, called affix array, only needs to be constructed once and is stored on disk.
- Flexible alphabet handling, including predefined DNA, RNA, and protein alphabets and the possibility to use personalized ones.
- Matching on forward and reverse complement strands. Matching in plain FASTA files is also possible.
- Support of a variety of patterns with ambiguous IUPAC symbols.
- Standard and user-defined base pairing rules.
- Integrated fast global and local chaining algorithms.
- Output of results in different formats, including BED for visualization in the UCSC Genome Browser.

Structator consists of two command line programs: *afconstruct* and *afsearch*. *afconstruct* allows the construction of tables that constitute the affix array index data structure of the target database. *afsearch* allows fast sequence-structure pattern matching in a precomputed affix array or in the plain database.

This software is available as open source under the GNU General Public License Version 3.

A.2 Index construction with *afconstruct*

afconstruct constructs the affix array index data structure of a given database. In summary, the process of construction includes reading the database in FASTA format, mapping the sequences of the database to an alphabet, selecting the desired tables of the index to be constructed, and saving the index to files on disk. This process is performed smoothly by *afconstruct*, where the user only

<file>	Load FASTA file
-alph <file>	Use alphabet defined in file
-dna	Use 4-letter DNA alphabet (default)
-rna	Use 4-letter RNA alphabet
-protein	Use 20-letter protein alphabet
-a	Construct all tables
-suf	Construct suf table
-lcp	Construct lcp table
-skp	Construct skp table
-aflk	Construct aflk table
-sufr	Construct sufr table
-lcpr	Construct lcpr table
-skpr	Construct skpr table
-aflkr	Construct aflkr table
-s <index>	Save constructed structures to given index name
-x	Do not save alphabetically transformed sequence
-c	Output constructed structures to screen
-t <file>	Output constructed structures to text file
-time	Display elapsed times

Table A.1: Overview of options of program *afconstruct*.

needs to set a few options. An overview of all possible options is given in Table A.1 and their detailed description is given below.

Index construction options

- <file>
  <file> is the path and name of the FASTA file for which the is index is to be constructed. The file may contain one or more sequences and all are selected for index construction. Note that index-based search in the forward and reverse complement sequences only requires the construction of a single index.
- -alph <file>
  -alph takes as parameter the path and name of the text file specifying an alphabet. The sequences' characters are mapped to this alphabet and the sequences are then said to be alphabetically transformed. The index is constructed for the alphabetically transformed sequences. This option also allows alphabet reduction. Each line in the file specifies a class of characters, which means that all characters of a class are not distinguished between each other. Below is an example of an alphabet file.

  ```
  Aa A
  Cc
  Gg
  ```

```
TtUu U
*BbDdHhNnYyRrSsVvWwKkMmXx
```

Lines beginning with *, like the last one, imply a class of wildcards (i.e. ambiguous characters). Wildcards in the database indicate unknown or unsequenced regions, hence such regions cannot be matched against any pattern. Furthermore, characters must be given without spaces in each line. A space and a character imply that the first character after the space is a so-called *class representative*. The class representative is shown in place of the original character when outputting transformed sequences to file or screen. If no representative is explicitly specified, the first character of the line is chosen as the representative. In summary, in the example above we have 5 character classes, whose representatives are A, C, G, U, and *.

As a remark, although ambiguous IUPAC character such as N, R, Y, etc. indicate unknown regions in the database, they can be used for defining patterns. It is noted here that the user does not have to create character classes for such characters since they are already recognized by *Structator*. More about this is discussed in the section about program *afsearch*.

- `-dna`, `-rna`, `-protein`
  These options allow transforming the input sequences to predefined alphabets. The alphabet for DNA, RNA, and protein sequences has size 4, 4, and 20, respectively. More precisely, the characters of each alphabet option are the following:
  `-dna`: A, C, G, T
  `-rna`: A, C, G, U
  `-protein`: A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y
  Uppercase and lowercase characters are not distinguished. If the sequences contain characters other than the ones above, one can create a new alphabet in a text file and use it with the option `-alph`.

- `-a`
  `-a` selects all eight tables of the affix array for construction. The tables are listed next.

- `-suf`, `-lcp`, `-skp`, `-aflk`,
  `-sufr`, `-lcpr`, `-skpr`, `-aflkr`
  These options allow the user to individually select the desired tables of the index to be constructed. Each option corresponds, as expected, to the table of the same name, that is:

  `-suf`: suffix array
  `-lcp`: longest common prefix
  `-skp`: skip
  `-aflk`: affix link
  `-sufr`: reverse prefix (i.e. suffix array of the reverse sequences)
  `-lcpr`: longest common prefix of the reverse sequences

`-skpr`: skip of the reverse sequences
`-aflkr`: affix link of the reverse sequences

Note that, because certain tables depend on another for their construction, a table may be constructed even if it is not selected. For example, table suf_F will automatically be constructed if the user only selects table lcp_F. For constructing table aflk_F (aflk_R) there are two possibilities. By selecting aflk_F (aflk_R) only, tables suf_F, lcp_F (lcp_R), and suf_R are automatically selected as well, and binary search method is used in the construction of aflk_F (aflk_R). If the user additionally selects skp_F (skp_R), the construction of aflk_F (aflk_R) is sped up by the additional use of this table. The skip tables skp_F and skp_R can be deleted by the user after the construction of the affix link tables aflk_F and aflk_R because they are not required for pattern matching.

- `-s <index>`
  By using the option `-s` along with an index name, each table that is constructed is stored on disk in its own file. The name of each file is `[index name].[table name]`. Additional files are also stored. One file with extension `.alph` stores the alphabet, one with extension `.base` stores basic information about the sequences such as their length, and one with extension `.des` stores the description of each sequence. The sequences and alphabetically transformed sequences are stored in a file with extension `.seq` and `.tseq`, respectively. Note that all the generated files are binary.
- `-x`
  This option prevents *afconstruct* from saving alphabetically transformed sequences to file. This is useful for saving disk space, but note that the sequences of the index will be transformed each time program *afsearch* (see next section) is executed.
- `-c`
  `-c` outputs the constructed tables and the corresponding suffixes (or reverse prefixes) to screen. For ease of readability, the strings of the reverse prefixes are printed in reverse order. This option is only recommended for small databases, say, with sequence length up to 100.
- `-t <file>`
  `-t` works like the option `-c`, but it directs the output to the specified file.
- `-time`
  With this option the time required to construct each selected table is displayed.

Be aware that the generated files may overwrite existing ones without warning!

Using *afconstruct*

We show an example for constructing all tables of the affix array, including also tables skp and skpr, for the Rfam database release 9.1. The database is stored in the file `Rfam.fas`. Because the sequences contain characters different from the 4-character RNA alphabet, we use with option `-alph` the same alphabet file that is exemplarily described above with 5 character classes. This file is here called `myrna.alph`. Below is the program call and its screen output.

```
$ ./afconstruct /path/to/fasta_file/Rfam.fas -alph
/path/to/alphabet_file/myrna.alph -a -s /path/to/save/index/Rfam
Fasta file:          Rfam.fas
Number of sequences: 1149685
Total length:        179030400
Computing suf... done
Computing lcp... done
Computing skp... done
Computing sufr... done
Computing lcpr... done
Computing skpr... done
Computing aflk with skpr... done
Computing aflkr with skp... done
```

The program execution produces these files:

```
$ ls -goh
total 5.0G
-rw-r--r-- 1 688M 2010-01-04 16:28 Rfam.aflk
-rw-r--r-- 1 688M 2010-01-04 16:39 Rfam.aflkr
-rw-r--r-- 1 68 2010-01-04 16:13 Rfam.alph
-rw-r--r-- 1 4.4M 2010-01-04 16:13 Rfam.base
-rw-r--r-- 1 29M 2010-01-04 16:13 Rfam.des
-rw-r--r-- 1 172M 2010-01-04 16:15 Rfam.lcp
-rw-r--r-- 1 116M 2010-01-04 16:15 Rfam.lcpe
-rw-r--r-- 1 116M 2010-01-04 16:17 Rfam.lcper
-rw-r--r-- 1 172M 2010-01-04 16:17 Rfam.lcpr
-rw-r--r-- 1 172M 2010-01-04 16:13 Rfam.seq
```

```
-rw-r--r-- 1 688M 2010-01-04 16:15 Rfam.skp
-rw-r--r-- 1 688M 2010-01-04 16:17 Rfam.skpr
-rw-r--r-- 1 688M 2010-01-04 16:15 Rfam.suf
-rw-r--r-- 1 688M 2010-01-04 16:17 Rfam.sufr
-rw-r--r-- 1 172M 2010-01-04 16:13 Rfam.tseq
```

A.3 Searching with *afsearch*

afsearch is the program for matching RNA sequence-structure patterns in a precomputed index or directly in a plain FASTA file. In case an index is used, matching patterns containing no base pairs and no ambiguous IUPAC characters requires only the precomputation of the suf_F table, otherwise tables suf_F and suf_R, lcp_F and lcp_R, and aflk_F and aflk_R are required. An overview of the options of *afsearch* is given in Table A.2 and are explained in more detail below.

Pattern search options

- `<data>`
  `<data>` is the path and target FASTA file or the path and prefix name of the files (i.e. file name without extension) storing an index. Remember to map the desired tables in case the target is an index.
- `-alph`
  `-alph` takes as parameter the path and name of the text file specifying an alphabet. See the full description of alphabet files above on the section about *afconstruct*. Note that this option is only effective if the target data is a FASTA file. Otherwise, if it is an index, the alphabet used is obtained from the index.

  If the FASTA file or the pattern file (see below option `-pat`) contains ambiguous IUPAC characters, e.g. N, R, S, Y, etc., these must be specified in the alphabet file. However, while in the FASTA file they indicate unknown or unsequenced regions and hence cannot be searched, these characters can be used to define patterns. It is noted that the use of ambiguous characters in patterns does not require the user to create the corresponding character classes, e.g. N denoting A, C, G, or U or R denoting A or G, since all standard IUPAC classes are already recognized by *Structator*.
- `-dna, -rna, -protein`
  Alphabet option for the respective kind of sequence data. For more details on the predefined DNA, RNA, and protein alphabets see the section about *afconstruct*. As with the `-alph` option, these are only effective if the target data is a FASTA file.

`<data>`	Index name or FASTA file
`-alph <file>`	Use alphabet defined by file (option applies only to FASTA file)
`-dna`	Use 4-letter DNA alphabet (default) (option applies only to FASTA file)
`-rna`	Use 4-letter RNA alphabet (option applies only to FASTA file)
`-protein`	Use 20-letter protein alphabet (option applies only to FASTA file)
`-pat <file>`	Search for (structural) patterns
`-for`	Search in the forward sequence (default)
`-rev`	Search in the reverse complement sequence. For searching in the forward sequence as well, combine it with -for
`-comp <file>`	Load base-pair complementarity rules from file
`-a`	Map all index tables
`-suf`	Map suf table
`-lcp`	Map lcp table
`-aflk`	Map aflk table
`-sufr`	Map sufr table
`-lcpr`	Map lcpr table
`-aflkr`	Map aflkr table
`-bed`	Output matches in BED format
`-allm`	Report all matches of variable length patterns, i.e. not only the longest ones
`-match <k>`	Report only sequences matching at least k different patterns
`-t <file>`	Write matches to text file instead of to screen
`-seqdesc`	Include sequence description in the results, otherwise tag each pattern match with the sequence id
`-time`	Display elapsed times
`-silent1`	Do not output matches
`-silent2`	Do not output anything
Chaining options:	
`-global`	Perform global chaining
`-local`	Perform local chaining
`-wf <wf>`	Apply weight factor $>$ 0.0 to fragments
`-maxgap <width>`	Allow chain gaps with up to the specified width
`-minscore <score>`	Report only chains with at least the specified score
`-minlen <length>`	Report only chains with number of fragments $>=$ length
`-top <#>`	Report only top # scoring chains of each sequence
`-chainrep <file>`	Write chaining report to text file instead of to screen
`-show`	Show chains in the report

Table A.2: Overview of options of program *afsearch*.

- `-pat <file>`

  `-pat` takes as parameter a text file containing one or multiple sequence-structure patterns. Each pattern is specified in three consecutive lines. The first line begins with the symbol $>$ followed by the description of the pattern. Optionally, the description may be followed by pipe symbols `|` separating these supplemental options:

  `weight`: a weight that is assigned to a chain fragment corresponding to a match of the respective pattern. If no weight is provided, value 1 is assumed by default.

  `startpos`: this option, used for computing the score of local chains, denotes the starting position of the pattern within the modeled RNA molecule. Alternatively, it can also be used to denote the expected starting match position of the pattern in the searched sequences, since this can reflect the distance of the pattern to other patterns modeling other substructures of the same RNA. Note that this option must be specified for all or none of the patterns. If not specified, the starting position of the patterns are automatically computed in a stacked way, i.e., `startpos` of the first pattern in a file is 1 and for other patterns it is the sum of the length of all patterns defined before it $+1$.

  `instance`: the instance is the number that defines the allowed order of occurrence of a chain fragment in a chain of matches. Patterns of equal instance are equivalent w.r.t. the chaining position. This option must be specified for none or all patterns. If not specified, the order of occurrence of chain fragments respects the top-bottom order in which the respective matching pattern is defined in the patterns file. For instance, a chain fragment of a pattern defined in the beginning of the file must occur at a position prior to a chain fragment of a pattern defined in the end of the file.

  `maxstemlength`: maximum length (i.e. number of base pairs) of the stem region of the pattern. The minimum length is derived from the dot-bracket sequence structure. For example, if the pattern has structure `((((...))))`, the minimum stem length is 4 and `maxstemlength` must be at least 4. The pattern characters for base pairs occurring in number above the minimum stem length are assumed to be ambiguous characters N.

  `maxrightloopextent` (alternatively `mrlex`): number of positions by which to extend the beginning (from left to right) of the loop region. The extended pattern positions are assumed to be characters N. See the example below for the usage of this option.

  `maxleftloopextent` (alternatively `mllex`): number of positions by which to extend the end (from left to right) of the loop region. The extended pattern positions are assumed to be characters N. See the example below for the usage of this option.

  `maxmispair`: maximum number of base pairs that may not obey the chosen complementarity rules, say, the Watson-Crick (A, U), (U, A), (C, G), (G, C).

  Supplemental options must be provided between two pipe symbols and its keyword, say, *weight*, is followed by the equal sign (=) and a value.

  The second line of the pattern definition contains the sequence information, i.e., a sequence of bases possibly containing ambiguous IUPAC characters. It is noted that *Structator* auto-

matically recognizes ambiguous characters and tries to match the corresponding base, e.g. A or G in place of an R. The third line contains the structure information in dot-bracket notation. In this notation, unpaired bases are represented by dots `.` and paired bases are represented by `(` and `)`. Note that positions specified by dots are not strictly unpaired, i.e., they may form a base pair with another position although this is not required. Supported structures are hairpins with bulges and/or internal loops and also single strands. Observe that for specifying a single stranded pattern it is necessary to provide a sequence of dots.

As an example, a patterns file may contain the following text.

```
>p0|maxleftloopextent=1|maxrightloopextent=1|maxstemlength=6
RNSNGKUNGCNHNSCY
(.((((....)))).)
```

The pattern above represents a set of patterns, namely:

```
>p0
RNSNGKUNGCNHNSCY
(.((((....)))).)
>p1
RNSNGKNUNGCNHNSCY
(.((((.....)))).)
>p2
RNSNGKUNGCNNHNSCY
(.((((.....)))).)
>p3
RNSNGKNUNGCNNHNSCY
(.((((......)))).)
>p4
NRNSNGKUNGCNHNSCYN
((.((((....)))).))
>p5
NRNSNGKUNGCNNHNSCYN
((.((((.....)))).))
>p6
NRNSNGKNUNGCNHNSCYN
((.((((.....)))).))
>p7
NRNSNGKNUNGCNNHNSCYN
((.((((......)))).))
```

- `-for`
  Option for searching in the forward sequences. This option is selected by default.
- `-rev`
  Option for searching in the reverse complement sequences. If used in combination with the option `-for`, search is performed in both the forward and reverse complement sequences, otherwise search is only performed in the reverse complement sequences. Observe that searching in reverse complement sequences of a database does not require computing an index for the reverse complement sequences. *afsearch* handles this by automatically computing the reverse complement of the patterns and by using these patterns for search.
- `-comp <file>`
  The parameter of the option `-comp` is a file specifying complementary bases. A line with two bases, given without any spaces or punctuation, implies that matches to the patterns can contain such a base pair. It is not necessary to specify the pairing rule twice. For example, for pairs (C, G) and (G, C) it suffices to have a line `CG`. Below is a sample file.

  ```
  AU
  CG
  GA
  GU
  ```

  According to this file, these base pairs are possible: (A, U), (U, A), (C, G), (G, C), (A, G), (G, A), (U, G), (G, U). Note that if the option `-comp` is not used, Watson-Crick base pairs are allowed by default.
- `-a`
  `-a` maps all six tables of the index (see the next options) to memory. Mapping means that they are made available to *afsearch*, but are not immediately loaded into memory. Blocks of data are only effectively loaded into memory as parts of the tables are read during pattern matching operations.
- `-suf`, `-lcp`, `-aflk`
  `-sufr`, `-lcpr`, `-aflkr`
  These options allow the individual selection of the tables that are mapped to memory. Matching single-stranded patterns containing no ambiguous characters requires only table suf_F. Otherwise, it is additionally mandatory the selection of tables suf_R, lcp_F, lcp_R aflk_F, and aflk_R.
- `-bed`
  Option for printing out the matches in BED format. Otherwise, if not used, the matches are printed out in a format similar to BED, but including the matched substring and its secondary structure.

- `-allm`
  This option is only effective when matching patterns of variable length. By using it, all matches of all possible different pattern lengths are reported. Otherwise, if not used and there are matches embedded in other matches of the same pattern, embedded matches are ignored. For example, consider a pattern with minimum length 6 and maximum length 10 and an arbitrary sequence. If the pattern matches with length 6 at sequence position 5 and with length 10 it matches at position 2, then the match at position 5 is ignored because it is embedded in the match at position 2.
- `-match <k>`
  `-match` with parameter `k` neglects sequences and pattern matches occurring in them if the matches are of not of at least `k` different patterns.
- `-t <file>`
  `-t` writes the matches to the specified file instead of to screen. The matches are sorted by sequence and, within a sequence, by ascending matching position.
- `-seqdesc`
  Option `-seqdesc` includes the sequences' description in the list of pattern matches. If this option is not used, the sequence is identified by a number that corresponds to its order of definition in the database, beginning from 0.
- `-time`
  Option to display the time needed to search for each pattern.
- `-silent1`
  `-silent1` avoids the output of matches and chains. Note that also the output to text file by the use of option `-t` is neglected.
- `-silent2`
  Option for not outputting anything.

Chaining options

- `-global`
  Option to perform global chaining of matches. It is the default option.
- `-local`
  Option to perform local chaining of matches.
- `-wf <wf>`
  `-wf` takes as parameter a positive weight factor that is applied to all chain fragments. For instance, if a chain fragment of a pattern has weight 2, a weight factor of 10 implies that the chain fragment will have weight 20.

- `-maxgap <width>`
  `-maxgap` takes as parameter the maximum distance (i.e. number of bases) allowed between chain fragments.
- `-minscore <score>`
  Report only chains with at least the specified score.
- `-minlen <len>`
  Report only chains with at least the specified number of chain fragments.
- `-top <#>`
  Report only top # scoring chains. If this option is not used, all chains are reported.
- `-chainrep <file>`
  `-chainrep` writes to the specified file the chaining report, otherwise the chains are written to screen. Chains are reported in descending order of their chain score.
- `-show`
  Show chain fragments and their coordinates (i.e. start and end matching position and weight) in the chaining report.

Using *afsearch*

We use *afsearch* in this example to search with three patterns derived from the consensus structure of the Rfam family *OxyS RNAs* (Acc.: RF00035). The patterns, shown below, are assigned a weight of 1 for computing global chains of matches. The patterns are stored in a file called `oxyS.pat`. We search in the index of Rfam release 10, here called `Rfam10`, which was preconstructed with *afconstruct*. The allowed base pairs are (A, U), (U, A), (C, G), (G, C), (G, U), and (U, G), which are specified in a text file and used with the option `-comp`. We also set *afsearch* to report global chains of matches with at least score 2 by using the option `-minscore`. The pattern matches and the chains are written to files `matches.txt` and `chains.txt`, respectively. The patterns file is as follows.

```
>HP1|maxrightloopextent=1|maxleftloopextent=1|maxmispair=6|weight=1
NNNNNNNNNNNNNNNNNNNNNACCCNUNANNNNNNNNNNNNNNNN
(((((((.((((.((.((..........)).))..)))))))))))
>HP2|maxrightloopextent=5|weight=1
GNNNNNCUCACNN
((((.....))))
>HP3|maxmispair=2|maxrightloopextent=2|weight=1
NNGGANCUNNNNNNNNNNN
```

```
(((((((.....)))))))
```

The command to call *afsearch* and the screen output are:

```
$ ./afsearch /path/to/index/Rfam10 -pat /path/to/patterns_file/oxyS.pat
-comp /path/to/comp_file/wcgu.comp -a -t matches.txt -minscore 2 -show
-chainrep chains.txt

Number of sequences: 1149685
Total length:        179030400

!Searching for pattern HP1 in the forward sequence(s)... done
!#Matches: 8619

!Searching for pattern HP2 in the forward sequence(s)... done
!#Matches: 1699

!Searching for pattern HP3 in the forward sequence(s)... done
!#Matches: 142219

!#Total matches: 152537
```

The first 10 lines of the matches file are:

```
$ head -n 15 matches.txt
![matched substring/structure] [seq. id] [matching pos.] [pattern id]
[weight] [strand]
ACGGAUCUCUUGGUUCUGG 119 11 2 1 f
(((((((.....)))))))
ACGGAUCUCUUGGUUCUGG 122 11 2 1 f
(((((((.....)))))))
ACGGAUCUCUUGGUUCUGG 124 11 2 1 f
(((((((.....)))))))
ACGGAUCUCUUGGUUCUGG 125 11 2 1 f
(((((((.....)))))))
ACGGAUCUCUUGGUUCUGG 126 11 2 1 f
(((((((.....)))))))
ACGGAUCUCUUGGUUCCGG 132 11 2 1 f
(((((((.....)))))))
ACGGAUCUCUUGGUUCUGG 136 11 2 1 f
(((((((.....)))))))
```

Observe that the matches are sorted by ascending sequence id. The id corresponds to the order of occurrence of the sequence in the database. Below are the first 26 lines of the chaining report showing 5 chains. There are in total 316 chains with at least score 2.

```
$ head -n 26 chains.txt
head -n 26 chains.txt
![sequence] [chain score] [chain length] [strand]
>CP000468.1+4477379-4477488 3 3 f
0 47 0 46 1
48 65 49 62 1
66 86 90 108 1
GAAACGGAGCGGCACCUCUUUUAACCCUUGAAGUCACUGCCCGUUUC GAGUUUCUCAACUC GCGGAUCUCCAGGAUCCGC
>CP000034.1+3532296-3532405 3 3 f
0 47 0 46 1
48 65 49 62 1
66 86 90 108 1
GAAACGGAGCGGCACCUCUUUUAACCCUUGAAGUCACUGCCCGUUUC GAGUUUCUCAACUC GCGGAUCUCCAGGAUCCGC
>AAJW02000005.1+188036-188145 3 3 f
0 47 0 46 1
48 65 49 62 1
66 86 90 108 1
GAAACGGAGCGGCACCUCUUUUAACCCUUGAAGUCACUGCCCGUUUC GAGUUUCUCAACUC GCGGAUCUCCAGGAUCCGC
>ABHW01000012.1+10515-10624 3 3 f
0 47 0 46 1
48 65 49 62 1
66 86 90 108 1
GAAACGGAGCGGCACCUCUUUUAACCCUUGAAGUCACUGCCCGUUUC GAGUUUCUCAACUC GCGGAUCUCCAGGAUCCGC
>AE014073.1+3594803-3594912 3 3 f
0 47 0 46 1
48 65 49 62 1
66 86 90 108 1
GAAACGGAGCGGCACCUCUUUUAACCCUUGAAGUCACUGCCCGUUUC GAGUUUCUCAACUC GCGGAUCUCCAGGAUCCGC
```

The chains are sorted by descending chain score. In this example, 3 is the maximum score possible. Each chain contains the description of the sequence where it occurs, the fragments' coordinates (i.e. expected or "stacked" start and end matching positions of the fragment, actual start and end matching positions of the fragment, and fragment weight), and the matching substring of the fragments.

B *RaligNAtor* user's manual

B.1 Introduction

RaligNAtor is a software package for fast approximate matching of RNA sequence-structure patterns. It searches sequence databases for occurrences of user-given patterns annotated with secondary structure. Its main features are:

- Implementations of new efficient user-selectable online and index-based matching algorithms.
- Matching computation based on a sequence-structure edit distance with a full set of edit operations on single bases and base pairs.
- Patterns can describe any (branching, non-crossing) RNA secondary structures. Sequence information can contain ambiguous IUPAC symbols.
- Search in DNA and RNA sequences possible due to flexible alphabet handling.
- Matching on forward and reverse complement strands.
- Customizable base pairing rules.
- Integrated fast algorithms for global and local chaining of matches.
- Output of results including matching positions, sequence-structure alignments, scores, etc.

For index-based matching, *RaligNAtor* uses a data structure based on the suffix array precomputed from the target sequence database. This precomputation is performed by the *sufconstruct* tool distributed with *RaligNAtor*, which is described next. *RaligNAtor*'s description follows subsequently.

This software is available as open source under the GNU General Public License Version 3.

B.2 Database preprocessing with *sufconstruct*

sufconstruct preprocesses a sequence database generating an index to be searched with *RaligNAtor* using algorithm *LESAAlign* or *LGSlinkAlign*. In summary, this procedure consists of reading the target database in FASTA format, mapping the sequences of the database to an alphabet consisting e.g. of characters A, C, G, and U, computing the required index structures according to the desired search algorithm, and saving the structures to files on disk. All this is performed smoothly, where

<file>	Load FASTA file
-alph <file>	Use alphabet defined in file
-dna	Use DNA alphabet {A, C, G, T} and IUPAC wildcards (default)
-rna	Use RNA alphabet {A, C, G, U} and IUPAC wildcards
-lesa	Construct index for LESAAlign (tables suf and lcp)
-lgslink	Construct index for LGSlinkAlign and LESAAlign (tables suf, lcp, and suf ^-1)
-s <index>	Save constructed structures to given index name
-x	Do not save alphabetically transformed sequence
-c	Output constructed structures to screen
-t <file>	Output constructed structures to text file
-time	Display elapsed times

Table B.1: Overview of options of program *sufconstruct*.

the user only needs to set a few options. An overview of all possible options is given in Table B.1 and their detailed description is given below.

Preprocessing options

- <file>
  <file> is the path and name of the FASTA file for which the is index is to be constructed. The file may contain one or more sequences and all are selected for index construction. Note that index-based search in the forward and reverse complement sequences only requires the construction of a single index.
- -alph <file>
  -alph takes as parameter the path and name of the text file specifying an alphabet. The sequences' characters are mapped to this alphabet and the sequences are then said to be alphabetically transformed. The index is constructed for the alphabetically transformed sequences. This option also allows for alphabet reduction (see below). Note that the used alphabet will also be used to map pattern characters when the constructed index is searched with *RaligNAtor*.

  Each line in the file specifies a class of characters of the alphabet. These must be ASCII printable characters, i.e. they must have character code between 32 and 127. A class of characters can be of three types:

  – Non-matching characters of the target sequence: specifies characters that can occur in the target sequence but *cannot* match any pattern character. This is useful for cases in which stretches of the target sequence are unknown, commonly represented by sequences of Ns. There can be only one such character class, specified in one line begin-

ning with symbol `!`. We emphasize that this class does not do any transformation of pattern characters. E.g.

```
!BbNnRrYySsWwKkMmDdHhVv
```

All characters used in this example that occur in the target sequence cause mismatches to any pattern character. However, these characters can be used with a different behavior in the pattern; see the following characters classes.

– Matching characters: a set of characters, whose members are not distinguished between each other, mapping pattern characters to match the same set of characters in the target sequence. In other words, characters (of both the pattern and the target sequence) belonging to one such class are transformed to a single symbol. Hence, this character class can be used for alphabet reduction. Such a character class is specified in one line with a simple list of the member characters. E.g.

```
Aa
```

The class above indicates that `A` and `a` are not distinguished between each other. Another didactic example is

```
AaM
```

This class allows `M` to be used in the pattern, even if it belongs to *non-matching characters of the target sequence*. `M` will be able to match `A`s and `a`s of the target sequence, but it will not match `M`s (if in the target sequence `M` is a non-matching character). We observe that, in the alignments reported by *RaligNAtor*, an alignment column of two matching characters of the same class is marked with symbol `|`, e.g. an alignment of `A` with `a`.

– Wildcards of the patterns: a class of this type specifies a special pattern symbol that can be used to match characters belonging to different *matching character* classes. A typical application is to specify a character e.g. `R` to match `A`s and `G`s in the target sequence, where `A` and `G` belong to two different *matching character* classes. Such a class is specified in one line beginning with a `*`. E.g.

```
*RAG
```

This class defines a wildcard symbol `R`, i.e. the first symbol after `*`, to match `A`s and `G`s in the target sequence. In addition, it will match every character belonging to the classes to which `A` and `G` belong, for instance `a`s and `g`s. Attention: make sure that all characters belonging to this class, except `R`, also belong to a *matching character* class. Otherwise, this wildcard class will not be accepted. We observe that a wildcard character aligned to a *matching character* of its class is annotated with a `+` in the *RaligNAtor* output, as in the following example.

```
Pattern  ...((-..))..(((...)))
         CCCAA-CCUUAAUCCAUARGA
         | ||| |||| ||||| |+||
Target   CGCAACCCUU-AUC-AAAGGA
         ...((...))-.((.....))
```

Naturally, alignments found with *RaligNAtor* show, for each non-gapped position, a single character of the corresponding character class. Each such character is called a *class representative*. By default, the first character different from `!` and `*` of each line is the representative of the class. Another more explicit way to specify the class representative is to end the class definition with a whitespace followed by the desired representative character. As an example, observe that the representative of the class of *non-matching characters of the target sequence* above is `B`. To set it to `N`, define it instead as

```
!BbNnRrYySsWwKkMmDdHhVv N
```

Below is an example of a complete alphabet file.

```
Aa A
Cc C
Gg G
UuTt U
*AG R
*CTU Y
*CA M
*UTG K
*UTA W
*CG S
*CGUT B
*AGUT D
*ACUT H
*ACG V
*ACGUT N
!NnRrYySsWwKkMmBbDdHhVv N
```

This alphabet file defines four *matching character* classes, whose representatives are `A`, `C`, `G`, and `U`. The class with representative `U`, for example, allows for the use in the pattern of both uppercase and lowercase `U`s and `T`s, such that any of these characters will match both uppercase and lowercase `U`s and `T`s in the target sequence. Because `U` is the class representative, alignments found with *RaligNAtor* will show `U` wherever these characters occur. The file also defines several wildcards that can be used in the pattern, e.g. `R`, to match uppercase and lowercase `A`s and `G`s in the target sequence. Finally, it defines a class of *non-matching characters of the target sequence*. This can contain characters of the previous two classes, e.g. `R`.

However, Rs occurring in the target sequence will cause mismatches, whereas R used in the pattern will match uppercase and lowercase As and Gs in the target sequence. Remember that

  – all characters used to define patterns must belong to a *matching character* and/or *wildcard* class and

  – all characters occurring in the target sequence must belong to a *matching character* or *non-matching character* class.

- `-dna`, `-rna`
  These options allow transforming the input sequences to predefined DNA or RNA alphabets. The alphabets are equal to the alphabet file shown above. The DNA alphabet only differs from the RNA alphabet by having `T` as class representative instead of `U`. If the target sequences contain other characters, one can create a new alphabet in a text file and use it with the option `-alph`.

- `-lesa`
  `-lesa` selects for construction the structures needed for searching the target database with algorithm *LESAAlign*. The structures consist of the suffix array suf and the longest common prefix table lcp. Note: suf and lcp are also constructed via option `-lgslink`. Hence, it is not necessary to select option `-lesa` if the database was already processed for search with the *LGSlinkAlign* algorithm.

- `-lgslink`
  `-lgslink` selects for construction the structures needed for searching the target database with algorithms *LGSlinkAlign* and *LESAAlign*. The structures consist of the suffix array suf, the longest common prefix table lcp, and the inverse suffix array suf^{-1}.

- `-s <index>`
  By using option `-s` along with an index name, each table that is constructed is stored on disk in its own file. The name of each file is `[index name].[table name]`. Additional files are also stored. One file with extension `.alph` stores the alphabet, one with extension `.base` stores basic information about the sequences such as their length, and one with extension `.des` stores the description of each sequence. The sequences and alphabetically transformed sequences are stored in a file with extension `.seq` and `.tseq`, respectively. Note that all the generated files are binary.

- `-x`
  This option prevents *sufconstruct* from saving alphabetically transformed sequences to file. This is useful for saving disk space, but it will require *RaligNAtor* to convert the sequences of the index for each search run.

- `-c`
  `-c` outputs the constructed tables and the corresponding suffixes to screen. This option is only recommended for small databases, say, with sequence length up to 100.

- `-t <file>`
  `-t` works like the option `-c`, but it directs the output to the specified file.
- `-time`
  With this option the elapsed construction time of each table is displayed.

Be aware that the generated files may overwrite existing ones without warning!

Using *sufconstruct*

We show an example for preprocessing a database for search with algorithm *LGSlinkAlign*. The database, stored in file `Rfam.fas`, consists of sequences obtained from the full alignments of Rfam release 10.1. Below is the program call and its screen output.

```
$ ./sufconstruct /path/to/fasta_file/Rfam.fas -rna -lgslink
-s /path/to/save/index/Rfam
Fasta file:          Rfam.fas
Number of sequences: 2756313
Total length:        824991406
Computing suf... done
Computing lcp... done
Computing suf... done
```

The program execution produces these files:

```
$ ls -goh
total 11.0G
-rw-r--r-- 1 68 2012-02-24 16:02 Rfam.alph
-rw-r--r-- 1 11M 2012-02-24 16:02 Rfam.base
-rw-r--r-- 1 67M 2012-02-24 16:02 Rfam.des
-rw-r--r-- 1 790M 2012-02-24 16:08 Rfam.lcp
-rw-r--r-- 1 2.1G 2012-02-24 16:08 Rfam.lcpe
-rw-r--r-- 1 790M 2012-02-24 16:02 Rfam.seq
-rw-r--r-- 1 3.1G 2012-02-24 16:08 Rfam.suf
-rw-r--r-- 1 3.1G 2012-02-24 16:08 Rfam.sufinv
-rw-r--r-- 1 790M 2012-02-24 16:02 Rfam.tseq
```

B.3 Searching with *RaligNAtor*

RaligNAtor can search for given sequence-structure patterns in (1) a precomputed index using algorithm *LESAAlign* or *LGSlinkAlign* or (2) directly in a plain FASTA file using algorithm *ScanAlign* or *LScanAlign*. For computing an index, please refer to program *sufconstruct* above. All algorithms deliver the same results, differing for the user only in their running times. For faster index-based and online searches, we recommend using algorithms *LGSlinkAlign* and *LScanAlign*, respectively. An overview of the options of *RaligNAtor* is given in Table B.2 and are explained in more detail below.

Search options

- `<data>`
  `<data>` is the path and target FASTA file or the path and prefix name of the files (i.e. file name without extension) storing an index. *RaligNAtor* requires `<data>` to point to a FASTA file in case the user wants to perform an online search with algorithm *ScanAlign* or *LScanAlign* (see options `-scan` and `-lscan` below). For index-based searches with algorithm *LESAAlign* or *LGSlinkAlign, RaligNAtor* requires `<data>` to point to an index (see options `-lesa` and `-lgslink` below).
- `-alph`
  `-alph` takes as parameter the path and name of the text file specifying an alphabet. See the full description of alphabet files above in the section about *sufconstruct*.
- `-dna, -rna`
  Alphabet option for the respective kind of sequence. See section about *sufconstruct* for details.
- `-pat <file>`
  `-pat` takes as parameter a text file containing one or multiple sequence-structure patterns describing any (branching, non-crossing) RNA secondary structures. Each pattern is specified in three consecutive lines. The first line begins with the symbol $>$ followed by the description of the pattern. Optionally, the description may be followed by pipe symbols `|` separating these supplemental options:
  `replacement`, `deletion`, `arc-breaking`, `arc-altering`, `arc-removing`: cost of the respective edit operation, being the same whether the operation occurs in the target sequence or the pattern. The default cost for `arc-removing` is 2 and for all others it is 1.
  `cost`: cost (i.e. sequence-structure edit distance) threshold for matches. Its default value is 0.
  `indels`: number of allowed indels. Its default value is the cost threshold divided by the cost of an indel, i.e. `cost/deletion`. Note that since `cost` bounds the number of indels that

<data>	Index name or FASTA file
-alph <file>	Use alphabet defined by file (option applies only to FASTA file)
-dna	Use DNA alphabet {A, C, G, T} and IUPAC wildcards (default)
-rna	Use RNA alphabet {A, C, G, U} and IUPAC wildcards
-pat <file>	Structural pattern(s) to search for
-for	Search in the forward sequence (default)
-rev	Search in the reverse complement sequence. For searching in the forward sequence as well, combine it with -for
-comp <file>	Load base pair complementarity rules from file
-byseq	Sort matches by sequence and matching position
-byscore	Sort matches of the same pattern by descending score
-byscorea	Sort matches of the same pattern by ascending score
-table	Print matches in table format
-no-overlaps	Filter out low-scoring overlapping matches of the same pattern
-silent	Do not output matches
-progress	Show progress message for each $\sim 5\%$ processed data
Operation costs and thresholds. These do not override parameters set in the patterns file	
-replacement <cost>	Cost of a base mismatch (default = 1)
-deletion <cost>	Cost of base deletion/insertion (default = 1)
-arc-breaking <cost>	Cost of an arc-breaking (default = 1)
-arc-altering <cost>	Cost of an arc-altering (default = 1)
-arc-removing <cost>	Cost of an arc-removing (default = 2)
-cost <x>	Allow edit distance $<=$ x (default = 0)
-indels <x>	Allow number of indels $<=$ x (default = cost / cost of one indel)
Index-based algorithmic variants*	
-lgslink	Uses early-stop acceleration, enhanced suffix array, and generalized suffix links
-lgslink_nof	Variant lgslink with disabled sequence-based filter
-lesa	Uses early-stop acceleration and enhanced suffix array
*lgslink requires tables suf, lcp, and sufinv. lesa requires only suf and lcp.	
Online algorithmic variants	
-scan	Slides a window over the target sequence reusing matrix entries
-lscan	Scanning variant with early-stop acceleration
-alignngl	Aligns globally reporting the best alignment (no pattern matching)
Chaining options	
-global	Perform global chaining
-local	Perform local chaining
-wf <wf>	Apply weight factor > 0.0 to fragments
-maxgap <width>	Allow chain gaps with up to the specified width
-minscore <score>	Report only chains with at least the specified score
-minlen <length>	Report only chains with number of fragments $>=$ length
-top <#>	Report only top # scoring chains of each sequence
-allglobal	Report for each sequence all global chains satisfying above criteria
-show	Show chains in the report
-show2	Print complete sequences and omit all other matching information

Table B.2: Overview of options of *RaligNAtor*.

can actually occur in a match, if `indels*deletion>cost` *RaligNAtor* will also automatically set `indels=cost/deletion`.
`weight`: a weight that is assigned to a chain fragment corresponding to a match of the respective pattern. Its default value is the score associated to a match; see match score definition in *RaligNAtor*'s publication.
`startpos`: this option, used for computing the score of local chains, denotes the starting position of the pattern within the modeled RNA molecule. Alternatively, it can also be used to denote the expected starting match position of the pattern in the searched sequences, since this can reflect the distance of the pattern to other patterns modeling other substructures of the same RNA. Note that this option must be specified for all or none of the patterns. If not specified, the starting position of the patterns are automatically computed in a stacked way, i.e., `startpos` of the first pattern in a file is 1 and for other patterns it is the sum of the length of all patterns defined before it $+1$.

Supplemental options must be provided between two pipe symbols and its keyword, e.g. *weight*, is followed by the equal sign (=) and a value. We observe that these options can also be provided in the command line call to *RaligNAtor*, overriding the respective option value given in the patterns file.

The second line of the pattern definition contains the sequence information, i.e., a sequence of bases possibly containing ambiguous IUPAC characters. *RaligNAtor* automatically recognizes ambiguous characters and tries to match the corresponding base, e.g. A or G in place of an R. The third line contains the structure information in dot-bracket notation. In this notation, unpaired bases are represented by dots `.` and paired bases are represented by `(` and `)`. Observe that for specifying a completely single stranded pattern it is necessary to provide a sequence of dots.

As an example, a patterns file may contain the following text.

```
>tRNA-pat|replacement=2|deletion=3|arc-removing=5
GSSVVYRURGYYYARYUGGUUARMRCRYYDSVYUBHHAMBCHRDWRRUYRYRGGUUCRAWUCCYDYHNBBNSYR
(((((((..((((.........)))).(((((.......))))).....(((((.......)))))))))))).
```

Another example is a file containing multiple patterns as follows.

```
>ires1|cost=2|indels=0
UGAWCUKD
........
>ires2|indels=1|cost=4
DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH
((((((.........................))))))
```

```
>ires3|indels=0|cost=1
VNHUAUUUADNBWUAC
((((....)))).....
>ires4|indels=2|cost=3
CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG
(.((((((....((((.....))))..))))))..)
>ires5|indels=1|cost=3|deletion=2
BHKHDHDSNBHDRGUNSNSNNNWNN
(((...((((......))))..)))
```

- `-for`
  Option for searching in the forward sequences. This option is selected by default.
- `-rev`
  Option for searching in the reverse complement sequences. If used in combination with the option `-for`, search is performed in both the forward and reverse complement sequences, otherwise search is only performed in the reverse complement sequences. Observe that searching in reverse complement sequences of a database does not require computing an index for the reverse complement sequences. *RaligNAtor* handles this by automatically computing the reverse complement of the patterns and by using these patterns for search. The patterns will contain complement characters according to the IUPAC table. This holds for alphabets specified with option `-dna`, `-rna`, or `-alph`. Characters not belonging to the IUPAC table cannot be complemented and remain unchanged. Base pairing rules are also automatically complemented. This means that, given Watson-Crick and wobble pairs, Watson-Crick pairs remain unchanged but accepted pairs derived from wobble (U, G) and (G, U) pairs automatically become (A, C) and (C, A). Note that (A, C) and (C, A) pairs must not be defined using option `-comp` (see below), since these pairs are then allowed when searching the forward sequences.
- `-comp <file>`
  The parameter of option `-comp` is a file specifying complementary bases. A line with two bases, given without any whitespaces or punctuation, implies that matches to the patterns can contain such a base pair. It is not necessary to specify the pairing rule twice. For example, for pairs (C, G) and (G, C) it suffices to provide a line `CG`. Below is a sample file.

  ```
  AU
  CG
  GA
  GU
  ```

According to this file, these base pairs are possible: (A, U), (U, A), (C, G), (G, C), (A, G), (G, A), (U, G), (G, U). Note that if the option `-comp` is not used, Watson-Crick base pairs are allowed by default.

- `-byseq`
  With this option matches are reported by sequence and matching position, such that matches at the beginning of a sequence are reported first. Note that with this option matches are not reported during search as they are found, but only once the search in the entire database is completed.
- `-byscore`, `-byscorea`
  With `-byscore` or `-byscorea` matches are sorted in descending or ascending order of their score, respectively. The match score is inversely proportional to the cost associated to a match; see exact score definition in *RaligNAtor*'s publication. Note that since the score for different patterns is not normalized, matches of the same pattern are reported consecutively.
- `-table`
  Option for reporting the matches in a table format, with one match per row.
- `-no-overlaps`
  `-no-overlaps` filters out low-scoring overlapping matches of the same pattern. More precisely, if the starting and ending positions of a matched substring overlap with the starting and ending positions of another matched substring of the same pattern, only the matched substring with a higher score is reported. In the case of a tie, one of the matches is arbitrarily filtered out. *RaligNAtor* checks several times during search for overlapping matches, hence avoiding a memory overflow in the case of highly sensitive patterns. Note that this option used with the different online and index-based search algorithms does not guarantee an identical output of matches. This can occur due to the different order by which matches are found and filtered out.
- `-silent`
  `-silent` disables the output of matches.
- `-progress`
  `-progress` shows a progress message for each $\sim 5\%$ processed data.
- `-replacement`, `-deletion`, `-arc-breaking`, `-arc-altering`, `-arc-removing`
  Options taking each a value that specifies the cost of the respective edit operation, with meaning and default value as detailed above for option `-pat`. A used option holds for all patterns in a patterns file and overrides the respective value specified in that file. To specify different operation costs for each searched pattern, see option `-pat`.
- `-cost`, `-indels`
  Cost threshold and number of allowed indels for matches. As with the edit operation costs provided in the command line, the value given via these options holds for all patterns of a

patterns file and override the respective value specified in that file. To specify different cost thresholds and number of allowed indels for each searched pattern, see option `-pat` above.

- `-lgslink`, `-lesa`
  Selects one of the index-based algorithms *LGSlinkAlign* or *LESAAlign*. These algorithms require an index of the target database, which can be generated with the *sufconstruct* tool above.

  Note: since version 1.1 of *RaligNAtor*, *LGSlinkAlign* performs in a first step sequence-based filtering with standard dynamic programming considering only edit operations on single bases, i.e. insertions, deletions, and replacements. In a second step, it considers also edit operations on base pairs. This filtering can considerably speed up search and affects neither sensitivity nor specificity, but the following condition must be fulfilled. If the cost of an insertion operation is set to e.g. 2, then the cost of an arc altering (option `-arc-altering`) and arc removing (option `-arc-removing`) must be set to at least 2 and 4, respectively, since these imply one and two deletions. The user is responsible for this consistency.

- `-lgslink_nof`
  Selects algorithm *LGSlinkAlign* but does not perform sequence-based filtering.

- `-scan`, `-lscan`
  Selects one of the online algorithms *ScanAlign* or *LScanAlign*. These algorithms operate directly on the database provided as FASTA file.

- `-aligngl`
  Aligns globally each sequence-structure pattern and each sequence of the database reporting the best alignment and the respective sequence-structure edit distance.

We remark that matches are reported on the standard output channel (stdout), whereas additional information such as set costs and thresholds is redirected to the standard error channel (stderr).

Chaining options

The following options allow to chain matches of the different patterns specified in one patterns file. A chain of matches is a sequence of non-overlapping matches (where each match is then called a chain *fragment*) such that the order of the matches in the chain resembles the order of the respective patterns in the patterns file.

- `-global`
  Option to perform global chaining of matches.

- `-local`
  Option to perform local chaining of matches.

- `-wf <wf>`
  `-wf` takes as parameter a positive weight factor that is applied to all chain fragments. For instance, if a chain fragment of a pattern has score 2, a weight factor of 10 implies that the chain fragment will have score 20.
- `-maxgap <width>`
  `-maxgap` takes as parameter the maximum distance (i.e. number of bases) allowed between chain fragments.
- `-minscore <score>`
  Report only chains with at least the specified score.
- `-minlen <len>`
  Report only chains with at least the specified number of chain fragments.
- `-top <#>`
  Report only top # scoring chains. If this option is not used, all chains are reported.
- `-allglobal`
  Guarantees that all global chains are reported without discarding any chains with the same score.
- `-show`
  Show chain fragments and their coordinates (i.e. start and end matching position and score) in the chaining report.
- `-show2`
  Print complete sequences for which at least one chain was found and omit all other matching information. A sequence is only printed once. Sequences are printed in their order of occurrence in the database.

We note that chains are reported in descending order of their chain score.

Using *RaligNAtor*

As an example, we used *RaligNAtor* to search for five patterns derived from the consensus structure of the Rfam family *Cripavirus internal ribosome entry site* (Acc.: RF00458). The patterns, called ires1, ires2, ires3, ires4, and ires5, are shown above in the description of option `-pat`. Here, we stored these patterns in a file called `ires.pat`. The searched database contained sequences obtained from the full alignments of Rfam 10.1. To search using algorithm *LGSlinkAlign*, we preprocessed this database with *sufconstruct* generating an index called `Rfam`. The allowed base pairs were (A, U), (U, A), (C, G), (G, C), (G, U), and (U, G), which were specified in a text file and used with the option `-comp`. We also set *RaligNAtor* to report global chains of matches with minimum length 5 by using the option `-minlen`. Due to the large number of expected matches for single pat-

terns, we used option `-silent` to prevent matches from being printed out but used option `-show` to print out the resulting chains.

The command call to *RaligNAtor* and the screen output are as follows.

```
$ ./RaligNAtor/path/to/index/Rfam10 -pat /path/to/patterns_file/ires.pat
-comp /path/to/comp_file/rna.comp -lgslink -silent -global -minlen 5 -show

!Number of sequences: 2756313
!Total length:        824991406


!Searching for pattern ires1 in the forward sequence(s)...
Cost threshold (edist) = 2
Max. allowed indels = 0
Min./Max. match length = 8 / 8
Max. match score = 8
Costs: Replacement = 1
Deletion = 1
Arc-breaking = 1
Arc-altering = 1
Arc-removing = 2


Time: 160822.0290 ms
Number of matches: 16033351


!Searching for pattern ires2 in the forward sequence(s)...
Cost threshold (edist) = 4
Max. allowed indels = 1
Min./Max. match length = 35 / 37
Max. match score = 48
Costs: Replacement = 1
Deletion = 1
Arc-breaking = 1
Arc-altering = 1
Arc-removing = 2


Time: 3607395.4620 ms
Number of matches: 8950417


!Searching for pattern ires3 in the forward sequence(s)...
Cost threshold (edist) = 1
Max. allowed indels = 0
Min./Max. match length = 16 / 16
Max. match score = 24
Costs: Replacement = 1
Deletion = 1
Arc-breaking = 1
Arc-altering = 1
Arc-removing = 2


Time: 96774.9180 ms
Number of matches: 1052


!Searching for pattern ires4 in the forward sequence(s)...
Cost threshold (edist) = 3
Max. allowed indels = 2
Min./Max. match length = 31 / 35
Max. match score = 53
Costs: Replacement = 1
Deletion = 1
Arc-breaking = 1
Arc-altering = 1
Arc-removing = 2


Time: 871779.0860 ms
Number of matches: 112
```

```
!Searching for pattern ires5 in the forward sequence(s)...
Cost threshold (edist) = 3
Max. allowed indels = 1
Min./Max. match length = 24 / 26
Max. match score = 39
Costs: Replacement = 1
Deletion = 2
Arc-breaking = 1
Arc-altering = 1
Arc-removing = 2


Time: 798519.5760 ms
Number of matches: 1222639


Total number of matches: 26207571


!Chaining matches... done
Time: 13660.1450 ms


![sequence] [chain score] [chain length] [strand]
>AB183472.1/62866484 171 5 f
0 7 10 18 8
8 43 19 54 47
44 59 79 95 24
60 92 99 132 53
93 117 147 172 39
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++++++++ ++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++++++++++++||++++++++++
UGAUCUGA UAGAAGUAAGAAAAUUCCUAGUUAUAA-UAUUUUUA AGUUAUUUAGCUUUAC CAGGAUGGGGUGCAGCGUUCCUGCAAUAUCCAG CCUUGUAGUUUUAGUGGACUUUAGG
........ ((((((......................-..)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
>AB017037.1/62866484 171 5 +
0 7 10 18 8
8 43 19 54 47
44 59 79 95 24
60 92 99 132 53
93 117 147 172 39
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++++++++ ++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++++++++++++||++++++++++
UGAUCUGA UAGAAGUAAGAAAAUUCCUAGUUAUAA-UAUUUUUA AGUUAUUUAGCUUUAC CAGGAUGGGGUGCAGCGUUCCUGCAAUAUCCAG CCUUGUAGUUUUAGUGGACUUUAGG
........ ((((((......................-..)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
>AF218039.1/60286228 171 5 +
0 7 10 18 8
8 43 19 55 48
44 59 80 96 24
60 92 100 133 53
93 117 149 173 38
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++++++++|++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++++++++++++||++++ +++++
UGAUCUUG UUGUAAAUACAAUUUUGAGAGGUUAAUAAAUUACAA AGCUAUUUAGCUUUAC CAGGAUGCCUAGUGGCAGCCCCACAAUAUCCAG UUUUUCAGAUUAGGUAGUC-GAAAA
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((....(((......)))-..)))
>AF014388.1/60786278 170 5 +
0 7 10 18 8
8 43 19 55 48
44 59 80 96 24
60 92 100 133 52
93 117 150 174 38
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++++++++|++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++ ++++++++++||++++++++++
UGAUCUUG UUCCUUAUACAAUUUUGAGAGGUUAAUAAGAAGGAA AACUAUUUAGUUUUAC CAGGAUGCCUAUUGGCAGCCCCAUAAUAUCCAG UU-AUAUGAUUAGGUUGUCAUUUAG
........ ((((((.........................)))))) ((((....)))).... (.((((((.....(((.....)))...)))))).) ((-...((((......))))...))
>AF014388.1/60786278 170 5 +
0 7 10 18 8
8 43 19 55 48
44 59 80 96 24
60 92 100 133 52
93 117 149 174 38
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++++++++|++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++++++++++++||++++++++++
UGAUCUUG UUCCUUAUACAAUUUUGAGAGGUUAAUAAGAAGGAA AACUAUUUAGUUUUAC CAGGAUGCCUAUUGGCAGCCCCAUAAUAUCCAG CUUAUAUGAUUAGGUUGUCAUUUAG
```

```
........ ((((((.........................)))))) ((((....)))).... (.((((((.....(((.....)))..)))))).) ((....((((......))))...))
>AB006531.1/60036204 170 5 +
0 7 10 18 8
8 43 20 56 47
44 59 82 98 24
60 92 102 135 53
93 117 150 175 38
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++ ++++++++++++++++++|++++++++ +++|||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++++++++++++||++++++++++
UGAUCUUA AAAAUUAGGUUAAAUUUCGAGGUUAAAAAUAGUUUU GUAUAUUUAUACUUAC CAAGAUGGACCGGAGCAGCCCUCCAAUAUCUAG GCUCAAACAUUAAGUGGUGUUGUGC
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) ((....((((......))))...))
>EU680971.1/184383 169 5 +
0 7 10 18 8
8 43 19 54 47
44 59 80 96 24
60 92 100 133 51
93 117 147 172 39
........ ((((((.........................)))))) ((((....)))).... (.((((((....((-((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNN-DGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++ ++++|++++++++ +++|||||||++++||| ||+||+ ++++++ +||+++||+++++++||+|| ++++++++++++||++++++++++
UGAUCUUU AUCGGGACAUGCAAAUGCAAGG-ACAAAACUCCGAU GGAUAUUUAUCCUUAC CAGGAU-CAGCUCAGGCAGCCCCGAAAAAUCCAG CUUCGAAGAGAAGGUGCUCUAGAAG
........ ((((((.................-.......)))))) ((((....)))).... (.((((-....((.((.....))))..)))))).) (((...((((......))))..)))
>AF183905.1/56475848 168 5 +
0 7 10 18 8
8 43 20 55 47
44 59 81 97 24
60 92 101 136 50
93 117 151 176 39
........ ((((((.........................)))))) ((((....)))).... (.((((((....(-(((.....)))-)..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNN-NDGCRKYCCHV-HRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++++++++++++|+ ++++++ +++|||||||++++||| ||+||+++++++ ++||+++||++ +++++||+|| ++++++++++++||++++++++++
UGAUCUUG UGCGGAGGCAAAAUUUGCACAGUAUAAAA-UCUGCA ACCUAUUUAGGUUUAC CAAGAUCGGUGGAUAGCAGCCCUAUCAAUAUCUAG UUUAGAAGAUUAGGUAGUCUCUAAA
........ ((((((.........................-)))))) ((((....)))).... (.((((.....(.(((.....))).)...)))).) (((...((((......))))..)))
>EF517515.1/55125714 168 5 +
0 7 10 18 8
8 43 20 56 47
44 59 82 98 24
60 92 102 137 50
93 117 152 177 39
........ ((((((.........................)))))) ((((....)))).... (.((((((....(-(((.....)))-)..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNN-NDGCRKYCCHV-HRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++ ++++++|++++++++ +++|||||||++++||| ||+||+++++++ ++||+++||++ +++++||+|| ++++++++++++||++++++++++
UGAUCUUG UGUGGAGGCAAAAAUUUGCACAGUAUAAAAUCUGCA ACCUAUUUAGGUUUAC CAAGAUCGGUGGAUAGCAGCCCUAUCAAUAUCUAG UUUAGAAGAUUAGGUAGUCUCUAAA
........ ((((((.........................)))))) ((((....)))).... (.((((.....(.(((.....))).)...)))).) (((...((((......))))..)))
>DQ288865.1/58026001 168 5 +
0 7 10 18 8
8 43 20 56 48
44 59 81 97 24
60 92 101 134 52
93 117 149 173 36
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))).) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ ++++++++++++++++++++++++++|++++++++ +++|||||||++++||| ||+||+++++++++||+++||+++++++||+|| + ++++++++++||++++++++++
UGAACUUG UCUCUCAACAAAAAGCCACCGACAUUAAGAGAGAGA CCCUAUUUAGGGUUAC CAGGAUCUGCAACAGCAUUCCUGUAUCAUCCAG GG-UGAGGAUUGAGUUGACCUCAUC
........ ((((((.........................)))))) ((((....)))).... (.((((.....((((.....))))...)))).) ((-...((.(......).))...))
>EF517520.1/55135715 167 5 +
0 7 10 18 8
8 43 19 56 46
44 59 82 98 24
60 92 102 137 50
93 117 152 177 39
........ (-((((((.........................)))))) ((((....)))).... (.((((((....(-(((.....)))-)..)))))).) (((...((((......))))..)))
UGAWCUKD D-NNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNN-NDGCRKYCCHV-HRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ + ++++++++++++++++++ ++++++|++++++++ +++|||||||++++||| ||+||+++++++ ++||+++||++ +++++||+|| ++++++++++++||++++++++++
UGAUCUUG UCGCAGAGGCAAAAAUUUGCACAGUAUAAAAUCUGCA ACCUAUUUAGGUUUAC CAAGAUCGGUGGAUAGCAGCCCUAUCAAUAUCUAG UUUAGAAGAUUAGGUAGUCUCUAAA
........ (.((((((.........................)))))) ((((....)))).... (.((((.....(.(((.....))).)...)))).) (((...((((......))))..)))
>EF517519.1/55125714 167 5 +
0 7 10 18 8
8 43 19 56 46
44 59 82 98 24
60 92 102 137 50
93 117 152 177 39
........ (-((((((.........................)))))) ((((....)))).... (.((((((....(-(((.....)))-)..)))))).) (((...((((......))))..)))
UGAWCUKD D-NNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNN-NDGCRKYCCHV-HRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ + ++++++++++++++++++ ++++++|++++++++ +++|||||||++++||| ||+||+++++++ ++||+++||++ +++++||+|| ++++++++++++||++++++++++
```

```
UGAUCUUG UCGCAGAGGCAAAAAUUUGCACAGUAUAAAAUCUGCA ACCUAUUUAGGUUUAC CAAGAUCGGUGGAUAGCAGCCCUAUCAAUAUCUAG UUUAGAAGAUUAGGUAGUCUCUAAA
........ (.(((((.........................)))))) ((((....)))).... (.((((.....(.(((.....))).).)...)))).) (((...((((......))))..)))
>EF517521.1/55135715 167 5 +
0 7 10 18 8
8 43 19 56 46
44 59 82 98 24
60 92 102 137 50
93 117 152 177 39
........ (-(((((.........................)))))) ((((....)))).... (.(((((....(-(((.....)))-)..)))))) (((...((((......))))..)))
UGAWCUKD D-NNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNN-NDGCRKYCCHV-HRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+||++ + ++++++++++++++++++ ++++++|++++++++ +++|||||+++|||  ||+||+++++++ ++||+++||++ +++++||+|| ++++++++++++||++++++++++
UGAUCUUG UCGCAGAGGCAAAAAUUUGCACAGUAUAAAAUCUGCA ACCUAUUUAGGUUUAC CAAGAUCGGUGGAUAGCAGCCCUAUCAAUAUCUAG UUUAGAAGAUUAGGUAGUCUCUAAA
........ (.(((((.........................)))))) ((((....)))).... (.(((((.....(.(((.....))).).)...)))).) (((...((((......))))..)))
>AF178440.1/59256123 166 5 +
0 7 10 18 8
8 43 31 66 45
44 59 79 95 24
60 92 99 132 52
93 117 148 172 37
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|||+|||++ ++++++++++++++ ++++++++++ ++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++ ++++++++++||+++++++ ++
UGAUCUUG AUUCUGUACAUAAAAGUCGAAAGUAUU-GCUAUAGU GCCUAUUUAGGCAUAC CAGGAUGGCGCGUUGCAGUCCAACAAGAUCCAG UCCUAUACCUCGAGUCGGGUUU-GG
........ (((.((.....................-..)).))) ((((....)))).... (.((((.....((((.....))))...)))).) ((....((((......))))..-))
>AF536531.1/66416834 165 5 +
0 7 5 13 6
8 43 15 50 46
44 59 75 91 24
60 92 95 128 51
93 117 143 168 38
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
| |+ |++ + ++++++++++++++++++++++|++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++ ++++++++++||++++++++++
UAAUUUGA U-UUAGGUUAUAAUGUUAGGACUAUAAAAAUUAGCU AGUUAUUUAACUUUAC CAAGAUGGCCGUUGGCAGCCCCACGAAAUCUAG CUAUUUUGAUUAGGUGGUCAGAUAG
........ .-((((.........................)))).. ((((....)))).... (.((((.......(((.....)))....)))).) (((...((((......))))..)))
>AF022937.1/69357121 162 5 +
0 7 3 11 6
8 43 15 50 44
44 59 68 84 24
60 92 88 121 51
93 117 134 158 37
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|| + |++ + ++ + ++++++++++++++++++++ ++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++++++++++++||+++++++ ++
UGUUGUGU U-UGCGCGAUAAAUGCUGACGUGAAAACGUUGCGUA AGCUAUUUAGCUUUAC CAAGACGCCGUCGUGCAGCCCACAAAAGUCUAG GAGCAUACGCUAGGUCGCGUUG-AC
........ (-((((.........................)))).) ((((....)))).... (.((((.......(((.....)))....)))).) (.....((((......))))..-.)
>EU282007.1/69357121 162 5 +
0 7 3 11 6
8 43 15 50 44
44 59 68 84 24
60 92 88 121 51
93 117 134 158 37
........ ((((((.........................)))))) ((((....)))).... (.((((((....((((.....))))..)))))) (((...((((......))))..)))
UGAWCUKD DNNNDNDNHNDMWWDYBVNVDNBWHDWADNNNNNNH VNHUAUUUADNBWUAC CARGAYSNVNNNNDGCRKYCCHVHRWNRUCYAG BHKHDHDSNBHDRGUNSNSNNNWNN
|| + |++ + ++ + ++++++++++++++++++++ ++++++++ +++||||||++++||| ||+||+++++++++||+++||+++++++||+|| ++++++++++++||+++++++ ++
UGUUGUGU U-UGCGCGAUAAAUGCUGACGUGAAAACGUUGCGUA AGCUAUUUAGCUUUAC CAAGACGCCAUCGUGCAGCCCACAAAAGUCUAG GAGCAUACGCUAGGUCGCGUUG-AC
........ (-((((.........................)))).) ((((....)))).... (.((((.......(((.....)))....)))).) (.....((((......))))..-.)


Total number of chains: 17
```

Each chain contains the description of the sequence where the chain occurs followed by the chain score, chain length, and matched strand direction (+ for forward or − for reverse). In addition, it contains the fragments' coordinates (i.e. expected or "stacked" start and end matching positions of the fragment, actual start and end matching positions of the fragment, and fragment score) and the matching substring of the fragments along with their sequence-structure alignment to the corresponding patterns.

Bibliography

[1] F. Crick. On protein synthesis. In *Symposium of the Society for Experimental Biology*, volume 12, pages 138–163, 1958.

[2] F. Crick. Central dogma of molecular biology. *Nature*, 227(5258):561–563, 1970.

[3] S. Ohno. So much "junk" in our genome. *Evolution of genetic systems. Brookhaven Symp Biol.*, 23:366–370, 1972.

[4] L.E. Orgel and F.H.C. Crick. Selfish DNA: the ultimate parasite. *Nature*, 284(5757):604–607, 1980.

[5] S. M. Berget, C. Moore, and P. A. Sharp. Spliced segments at the 5' terminus of adenovirus 2 late mRNA. *Proceedings of the National Academy of Sciences USA*, 74(8):3171–3175, 1977.

[6] L.T. Chow, J.M. Roberts, J.B. Lewis, and T.R. Broker. A map of cytoplasmic RNA transcripts from lytic adenovirus type 2, determined by electron microscopy of RNA: DNA hybrids. *Cell*, 11(4):819–836, 1977.

[7] The International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[8] B. E. Bernstein, E. Birney, I. Dunham, E. D. Green, C. Gunter, and M. Snyder. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57–74, 2012.

[9] M. J. Hangauer, I. W. Vaughn, and M. T. McManus. Pervasive Transcription of the Human Genome Produces Thousands of Previously Unidentified Long Intergenic Noncoding RNAs. *PLoS Genet.*, 9(6):e1003569+, 2013.

[10] K. Kruger, P. J. Grabowski, A. J. Zaug, J. Sands, D. E. Gottschling, and T. R. Cech. Self-splicing RNA: autoexcision and autocyclization of the ribosomal RNA intervening sequence of Tetrahymena. *Cell*, 31(1):147–157, 1982.

[11] C. Guerrier-Takada, K. Gardiner, T. Marsh, N. Pace, and S. Altman. The RNA moiety of ribonuclease P is the catalytic subunit of the enzyme. *Cell*, 35(3):849–857, 1983.

[12] J. A. Doudna and T. R. Cech. The chemical repertoire of natural ribozymes. *Nature*, 418(6894):222–228, 2002.

[13] W. Gilbert. The RNA world. *Nature*, 319(6055):618, 1986.

[14] D. M. Simon and S. Zimmerly. A diversity of uncharacterized reverse transcriptases in bacteria. *Nucl. Acids Res.*, 36(22):7219–7229, 2008.

[15] H. M. Temin and S. Mizutani. RNA-dependent DNA Polymerase in Virions of Rous Sarcoma Virus. *Nature*, 226(5252):1211–1213, 1970.

[16] D. Baltimore. Viral RNA-dependent DNA Polymerase: RNA-dependent DNA Polymerase in Virions of RNA Tumour Viruses. *Nature*, 226(5252):1209–1211, 1970.

[17] W. Li, P. Zhang, J. P. Fellers, B. Friebe, and B. S. Gill. Sequence composition, organization, and evolution of the core Triticeae genome. *The Plant Journal*, 40(4):500–511, 2004.

[18] R. C. Lee, R. L. Feinbaum, and V. Ambros. The C. elegans heterochronic gene lin-4 encodes small RNAs with antisense complementarity to lin-14. *Cell*, 75(5):843–854, 1993.

[19] A. Fire, S. Xu, M. K. Montgomery, S. A. Kostas, S. E. Driver, and C. C. Mello. Potent and specific genetic interference by double-stranded RNA in Caenorhabditis elegans. *Nature*, 391(6669):806–811, 1998.

[20] J. Couzin. Breakthrough of the year. Small RNAs make big splash. *Science*, 298(5602):2296–2297, 2002.

[21] F. Calore, F. Lovat, and M. Garofalo. Non-Coding RNAs and Cancer. *Int. J. Mol. Sci.*, 14(8):17085–17110, 2013.

[22] Y. Barash, J. A. Calarco, W. Gao, Q. Pan, X. Wang, O. Shai, B. J. Blencowe, and B. J. Frey. Deciphering the splicing code. *Nature*, 465(7294):53–59, 2012.

[23] J. S. Mattick. The hidden genetic program of complex organisms. *Sci. Am.*, 291(4):60–67, 2004.

[24] J. H. Bergmann and D. L. Spector. Long non-coding RNAs: modulators of nuclear structure and function. *Curr. Opin. Cell Biol.*, 26:10–18, 2014.

[25] B.J. Tucker and R.R. Breaker. Riboswitches as versatile gene control elements. *Curr. Opin. Struct. Biol.*, 15(3):342–348, 2005.

[26] M. Hlevnjak, A. A. Polyansky, and B. Zagrovic. Sequence signatures of direct complementarity between mRNAs and cognate proteins on multiple levels. *Nucl. Acids Res.*, 40(18):8874–8882, 2012.

[27] S. Carpenter, D. Aiello, M. K. Atianand, E. P. Ricci, P. Gandhi, L. L. Hall, M. Byron, B. Monks, M. Henry-Bezy, J. B. Lawrence, L. A. J. ONeill, M. J. Moore, D. R. Caffrey, and K. A. Fitzgerald. A Long Noncoding RNA Mediates Both Activation and Repression of Immune Response Genes. *Science*, 341(6147):789–792, 2013.

[28] E. Loh, E. Kugelberg, A. Tracy, Q. Zhang, B. Gollan, H. Ewles, R. Chalmers, V. Pelicic, and C. M. Tang. Temperature triggers immune evasion by Neisseria meningitidis. *Nature*, 0(0):8874–8882, 2013.

[29] E. Westhof and P. Auffinger. RNA Tertiary Structure. *Encyclopedia of Analytical Chemistry*, pages 5222–5232, 2006.

[30] B. Lewin, J. E. Krebs, E. S. Goldstein, and S. T. Kilpatrick. *Genes X*. Jones & Bartlett Learning, 2011.

[31] R. T. Mitsuyasu, T. C. Merigan, A. Carr, J. A. Zack, M. A. Winters, C. Workman, M. Bloch, J. Lalezari, S. Becker, L. Thornton, B. Akil, H. Khanlou, R. Finlayson, R. McFarlane, D. E. Smith, R. Garsia, D. Ma, M. Law, J. M. Murray, C. von Kalle, J. A. Ely, S. M. Patino, A. E. Knop, P. Wong, A. V. Todd, M. Haughton, C. Fuery, J. L. Macpherson, G. P. Symonds, L. A. Evans, S. M. Pond, and D. A. Cooper. Phase 2 gene therapy trial of an anti-HIV ribozyme in autologous CD34+ cells. *Nature Medicine*, 15(3):285–292, 2009.

[32] A. Hüttenhofer and P. Schattner. The principles of guiding by RNA: chimeric RNA-protein enzymes. *Nature Reviews Genetics*, 7(6):475–482, 2006.

[33] Y. Wan, M. Kertesz, R. C. Spitale, E. Segal, and H. Y. Chang. Understanding the transcriptome through RNA structure. *Nature Reviews Genetics*, 12(9):641–655, 2011.

[34] S. W. Burge, J. Daub, R. Eberhardt, J. Tate, L. Barquist, E. P. Nawrocki, S. R. Eddy, P. P. Gardner, and A. Bateman. Rfam 11.0: 10 years of RNA families. *Nucl. Acids Res.*, 2012.

[35] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucl. Acids Res.*, 25(17):3389–3402, 1997.

[36] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988.

[37] E. K. Freyhult, J. P. Bollback, and P. P. Gardner. Exploring genomic dark matter: A critical assessment of the performance of homology search methods on noncoding RNA. *Genome Research*, 17(1):117–125, 2007.

[38] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, 1981.

[39] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, May 1998.

[40] E. P. Nawrocki and S. R. Eddy. Infernal 1.1: 100-fold faster RNA homology searches. *Bioinformatics*, 29(22):2933–2935, 2013.

[41] A. Lambert, M. Legendre, J.F. Fontaine, and D. Gautheret. Computing expectation values for RNA motifs using discrete convolutions. *BMC Bioinformatics*, 6:118, 2005.

[42] T. Macke, D. Ecker, R. Gutell, D. Gautheret, D.A. Case, and R. Sampath. RNAMotif – A new RNA secondary structure definition and discovery algorithm. *Nucl. Acids Res.*, 29(22):4724–

4735, 2001.

[43] J. H. Havgaard, E. Torarinsson, and J. Gorodkin. Fast Pairwise Structural RNA Alignments by Pruning of the Dynamical Programming Matrix. *PLoS Comput. Biol.*, 3(10):e193+, 2007.

[44] S. Will, K. Reiche, I. L. Hofacker, P. F. Stadler, and R. Backofen. Inferring noncoding RNA families and classes by means of genome-scale structure-based clustering. *PLoS Comput. Biol.*, 3(4):e65+, 2007.

[45] The International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, 2004.

[46] C. S. Ku and D. H. Roukos. From next-generation sequencing to nanopore sequencing technology: paving the way to personalized genomic medicine. *Expert Rev Med Devices*, 10(1):1–6, 2013.

[47] The 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012.

[48] U. Nagalakshmi, Z. Wang, K. Waern, C. Shou, D. Raha, M. Gerstein, and M. Snyder. The transcriptional landscape of the yeast genome defined by RNA sequencing. *Science*, 320(5881):1344–1349, 2008.

[49] F. Ozsolak, A. R. Platt, D. R. Jones, J. G. Reifenberger, L. E. Sass, P. McInerney, J. F. Thompson, J. Bowers, M. Jarosz, and P. M. Milos. Direct RNA sequencing. *Nature*, 461(7265):814–818, 2009.

[50] K. A. Wetterstrand. DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program. `http://www.genome.gov/sequencingcosts/`, 2013. Accessed October, 2013.

[51] D. Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, January 1997.

[52] U. Manber and E.W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[53] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

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

[55] A. Cornish-Bowden. Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984. *Nucl. Acids Res.*, 13(9):3021–3030, 1985.

[56] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for Loop Matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.

[57] T. Xia, J. Santalucia, M. E. Burkard, R. Kierzek, S. J. Schroeder, X. Jiao, C. Cox, and D. H. Turner. Thermodynamic Parameters for an Expanded Nearest-Neighbor Model for Formation of RNA Duplexes with Watson-Crick Base Pair. *Biochemistry*, 37(42):14719–14735, 1998.

[58] D.H. Mathews, J. Sabina, M. Zuker, and D.H. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *J. Mol. Biol.*, 288:911–940, 1999.

[59] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucl. Acids Res.*, 9(1):133–148, 1981.

[60] M. Zuker, D.H. Mathews, and D.H. Turner. Algorithms and Thermodynamics for RNA Secondary Structure Prediction: A Practical Guide. *RNA Biochemistry and Biotechnology*, 1999.

[61] J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29(6-7):1105–1119, 1990.

[62] R. Lorenz, S. H. Bernhart, C. Höner Zu Siederdissen, H. Tafer, C. Flamm, P. F. Stadler, and I. L. Hofacker. ViennaRNA Package 2.0. *Algorithms Mol. Biol.*, 6(1):26+, 2011.

[63] P. Gardner and R. Giegerich. A comprehensive comparison of comparative RNA structure prediction approaches. *BMC Bioinformatics*, 5(140), 2004.

[64] D. Sankoff. Simultaneous solution of the RNA folding, alignment and protosequence problem. *SIAM Journal on Applied Mathematics*, 45(5):810–825, 1985.

[65] J. Gorodkin, L. J. Heyer, and G. D. Stormo. Finding the most significant common sequence and structure motifs in a set of RNA sequences. *Nucl. Acids Res.*, 25(18):3724–3732, 1997.

[66] D. H. Mathews. Predicting a set of minimal free energy RNA secondary structures common to two sequences. *Bioinformatics*, 21(10):2246–2253, 2005.

[67] I. L. Hofacker, S. H. Bernhart, and P. F. Stadler. Alignment of RNA base pairing probability matrices. *Bioinformatics*, 20(14):2222–2227, 2004.

[68] S. Will, M. Siebauer, S. Heyne, J. Engelhardt, P.F. Stadler, K. Reiche, and R. Backofen. LocARNAscan: incorporating thermodynamic stability in sequence and structure-based RNA homology search. *Algorithms Mol. Biol.*, 8:14, 2013.

[69] S. Siebert and R. Backofen. MARNA: multiple alignment and consensus structure prediction of RNAs based on sequence structure comparisons. *Bioinformatics*, 21(16):3352–3359, 2005.

[70] T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *J. Comput. Biol.*, 9(2):371–388, 2002.

[71] C. Notredame, D.G. Higgins, and J. Heringa. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *J. Mol. Biol.*, 302(1):205–217, 2000.

[72] S. Schirmer and R. Giegerich. Forest alignment with affine gaps and anchors, applied in RNA structure comparison. *Theor. Comput. Sci.*, 483:51–67, 2013.

[73] D. Gautheret and A. Lambert. Direct RNA motif definition and identification from multiple sequence alignments using secondary structure profiles. *J. Mol. Biol.*, 313:1003–11, 2001.

[74] R.J. Klein and S.R. Eddy. RSEARCH: finding homologs of single structured RNA sequences. *BMC Bioinformatics*, 4(1):44, 2003.

[75] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998.

[76] K. Karplus, C. Barrett, and R. Hughey. Hidden Markov models for detecting remote protein homologies. *Bioinformatics*, 14(10):846–856, 1998.

[77] M. Madera and J. Gough. A comparison of profile hidden Markov model procedures for remote homology detection. *Nucl. Acids Res.*, 30(19):4321–4328, 2002.

[78] S. R. Eddy and R. Durbin. RNA sequence analysis using covariance models. *Nucl. Acids Res.*, 22(11):2079–2088, 1994.

[79] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjölander, R. C. Underwood, and D. Haussler. The application of stochastic context-free grammars to folding, aligning and modeling homologous RNA sequences. *unpublished*, 1994.

[80] Y. Sakakibara, M. Brown, R. C. Underwood, I. S. Mian, and D. Haussler. Stochastic context-free grammars for modeling RNA. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 5, pages 284–293. IEEE Computer Society Press, 1994.

[81] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[82] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[83] S. R. Eddy. A memory-efficient dynamic programming algorithm for optimal alignment of a sequence to an RNA secondary structure. *BMC Bioinformatics*, 3(1):18+, 2002.

[84] K. Sjölander, K. Karplus, M. Brown, R. Hughey, A. Krogh, Mian, and D. Haussler. Dirichlet mixtures: a method for improved detection of weak but significant protein sequence homology. *Comput. Appl. Biosci.*, 12(4):327–345, 1996.

[85] E. P. Nawrocki and S. R. Eddy. Query-Dependent Banding (QDB) for Faster RNA Similarity Searches. *PLoS Comput. Biol.*, 3(3):e56+, 2007.

[86] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2000.

[87] D. Younger. Recognition and parsing of context-free languages in time n3*. *Information and Control*, 10(2):189–208, 1967.

[88] T. Kasami. An efficient recognition and syntax algorithm for context-free algorithms. *Technical Report AFCRL-65-758*, 1965.

[89] E. P. Nawrocki, D. L. Kolbe, and S. R. Eddy. Infernal 1.0: inference of RNA alignments. *Bioinformatics*, 25(10):1335–1337, 2009.

[90] Z. Weinberg and W. L. Ruzzo. Sequence-based heuristics for faster annotation of non-coding RNA families. *Bioinformatics*, 22(1):35–39, 2006.

[91] D. L. Kolbe and S. R. Eddy. Fast filtering for RNA homology search. *Bioinformatics*, 27(22):3102–3109, 2011.

[92] J. Mistry, R. D. Finn, S. R. Eddy, A. Bateman, and M. Punta. Challenges in homology search: HMMER3 and convergent evolution of coiled-coil regions. *Nucl. Acids Res.*, 2013.

[93] Infernal User's Guide. `http://infernal.janelia.org/`, 2013.

[94] E. P. Nawrocki. Structural RNA Homology Search and Alignment Using Covariance Models. *PhD Thesis: Washington University School of Medicine*, 2009.

[95] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89(22):10915–10919, 1992.

[96] RNAMotif Users' Manual. `http://casegroup.rutgers.edu/casegr-sh-2.5.html`, 2001.

[97] D. Gautheret, F. Major, and R. Cedergren. Pattern searching/alignment with RNA primary and secondary structures: an effective descriptor for tRNA. *Comput. Appl. Biosci.*, 6(4):325–31, 1990.

[98] RNABOB: a program to search for RNA secondary structure motifs in sequence databases. `http://selab.janelia.org/software.html`.

[99] M. Dsouza, N. Larsen, and R. Overbeek. Searching for patterns in genomic data. *Trends Genet.*, 13(12):497–8, December 1997.

[100] B. Billoud, M. Kontic, and A. Viari. Palingol: a declarative programming language to describe nucleic acids' secondary structures and to scan sequence database. *Nucl. Acids Res.*, 24(8):1395–403, April 1996.

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

[102] G. Mauri and G. Pavesi. Algorithms for pattern matching and discovery in RNA secondary structure. *Theor. Comput. Sci.*, 335(1):29–51, 2005.

[103] Moritz G. Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, 2003.

[104] F. Meyer, S. Kurtz, R. Backofen, S. Will, and M. Beckstette. Structator: fast index-based search for RNA sequence-structure patterns. *BMC Bioinformatics*, 12(1):214, 2011.

[105] G. Mauri and G. Pavesi. Pattern discovery in RNA secondary structures using affix trees. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 278–294. Springer, 2003.

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

[107] S. J. Puglisi, W.F. Smyth, and A. Turpin. The performance of linear time suffix sorting algorithms. In *DCC '05: Proceedings of the Data Compression Conference*, pages 358–367, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

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

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

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

[113] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, volume 2476, pages 31–43. Springer, 2002.

[114] N. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758764, 1946.

[115] I.L. Hofacker, M. Fekete, and P.F. Stadler. Secondary structure prediction for aligned RNA sequences. *J. Mol. Biol.*, 319(5):1059–66, 2002.

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

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

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

[119] J.H. Havgaard, R.B. Lyngso, G.D. Stormo, and J. Gorodkin. Pairwise local structural alignment of RNA sequences with sequence similarity less than 40%. *Bioinformatics*, 21:1815–1824, 2005.

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

[121] A.O. Harmanci, G. Sharma, and D.H. Mathews. Efficient pairwise RNA structure prediction using probabilistic alignment constraints. *BMC Bioinformatics*, 8(130), 2007.

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

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

[124] P.P. Gardner, J. Daub, J. Tate, B.L. Moore, I.H. Osuch, S. Griffiths-Jones, R.D. Finn, E.P. Nawrocki, D.L. Kolbe, S.R. Eddy, and A. Bateman. Rfam: Wikipedia, clans and the "decimal" release. *Nucl. Acids Res.*, 2010.

[125] P.P. Gardner, J. Daub, J.G. Tate, E.P. Nawrocji, D.L. Kolbe, S. Lindgreen, A.C. Wilkinson, R.D. Finn, S. Griffith-Jones, S.R. Eddy, and A. Bateman. Rfam: updates to the RNA families database. *Nucl. Acids Res.*, 37:D136–D140, 2008.

[126] M.I. Abouelhoda and E. Ohlebusch. Chaining algorithms for multiple genome comparison. *Journal of Discrete Algorithms*, 3(2-4):321–341, 2005.

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

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

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

[130] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Inf. Comput.*, 213:13–22, 2012.

[131] B. Albrecht and V. Heun. Space Efficient Modifications to Structator - A Fast Index-Based Search Tool for RNA Sequence-Structure Patterns. In *Experimental Algorithms*, volume 7276 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2012.

[132] N. El-Mabrouk, M. Raffinot, J. E. Duchesne, M. Lajoie, and N. Luc. Approximate matching of structured motifs in DNA sequences. *J. Bioinform. Comput. Biol.*, 3(2):317–342, 2005.

[133] F. Meyer, S. Kurtz, and M. Beckstette. Fast online and index-based algorithms for approximate search of RNA sequence-structure patterns. *BMC Bioinformatics*, 14(1):226, 2013.

[134] Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118, March 1985.

[135] E. Ukkonen. Online construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[136] Y. Kanamori and N. Nakashima. A tertiary structure model of the internal ribosome entry site (IRES) for methionine-independent initiation of translation. *RNA*, 7(2):266–274, 2001.

[137] Z. Weinberg, J.X. Wang, J. Bogue, J. Yang, K. Corbino, R.H. Moy, and R.R. Breaker. Comparative genomics reveals 104 candidate structured RNAs from bacteria, archaea, and their metagenomes. *Genome Biology*, 11(3):R31, 2010.

[138] ERPIN Documentation - Manual. `http://tagc.univ-mrs.fr/erpin/`, 2006.

[139] David H. Mathews and Douglas H. Turner. Prediction of RNA secondary structure by free energy minimization. *Current Opinion in Structural Biology*, 16(3):270–278, 2006.

[140] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software Practice and Experience*, 44(11):1287–1314, 2014.

[141] H. Li and R. Durbin. Fast and accurate short read alignment with BurrowsWheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Doktorarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Hamburg, den Juni 2014

Unterschrift