Dissertation submitted for the degree of "Doctor rerum naturalium" (Dr. rer. nat.) to the Department of Technology, Bielefeld University

Methods for the Identification of Common RNA Motifs

by Benedikt Löwes

April 2017

Dissertation submitted for the degree of "Doctor rerum naturalium" (Dr. rer. nat.) to the Department of Technology, Bielefeld University

Methods for the Identification of Common RNA Motifs

by Benedikt Löwes

April 2017

Referees:

- Prof. em. Dr. Robert Giegerich
- Prof. Dr. Peter Unrau
- Prof. Dr. Markus Nebel

Gedruckt auf alterungsbeständigem Papier °° ISO 9706.
Printed on non-aging paper °° ISO 9706.

ABSTRACT

For a long time, non-coding RNAs were given less attention than messenger RNAs, even though their existence was proposed at a similar time in 1971, because the research focus was mostly on protein coding genes. With the discovery of catalytically active RNA molecules and micro RNAs, which are involved in the post-transcriptional regulation of gene expression, non-coding RNAs have gained widespread attention. It was revealed early on that non-coding RNAs are often more conserved in structure than in sequence. Since determining the function of non-coding RNAs includes costly and time consuming laboratory experiments, computational methods can help identifying further homologs of experimentally validated RNA families. But a question remains: can we identify potential RNAs with novel functions solely by using *in silico* methods?

In this thesis, we perform an evaluation of 4,667 viral reference genomes in order to identify common RNA motifs shared by multiple taxonomically distant viruses. One potential mechanism that might explain similar motifs in taxonomically distant viruses that infect common hosts by interacting with their cellular components is convergent evolution. Convergent evolution is used to describe the phenomenon that two different species that are originated from two ancestors share related or similar traits. By looking for long stretches of exact RNA structure matches with low sequence conservation, we want to maximize the chance that the common motifs are the result of structural convergence due to similar selection criteria in common host organisms. Viruses are an excellent fit when it comes to the discovery of shared RNA motifs without the involvement of conserved sequence regions because of their high mutation rates. We were able to identify 69 RNA motifs, which could not be assigned to any of the existing RNA families, with a length of at least 50 nucleotides that are shared among at least three taxonomically distant viruses.

The secondary structure of an RNA molecule can be represented as a string. Finding maximal repeats in strings can be done using well-known string matching techniques based on suffix trees and arrays. In contrast to normal RNA sequences, secondary structure strings represent base pairing interactions within a single molecule. Thus, not every substring of the secondary structure defines a well-formed RNA structure. Therefore, we describe a new data structure, the viable suffix tree, that takes the constraints on the RNA secondary structure into account

and only returns maximal repeats that are well-formed structures. But this data structure is not limited to RNA structures but can also be used for any other problem domain for which a set of allowed words can be defined, e.g. by using a grammar. However, the overall complexity of constructing the viable suffix tree cannot be lower than the complexity of the word problem for the language of such a grammar.

The limitation of exact structure matching is the need for long common stretches of secondary structures that are not allowed to have a mismatch at any position. But we need to allow small mismatches to find more potential targets, but current state of the art techniques use computationally too expensive methods for sequence and structure comparisons and exhibit high false positive rates around 50%. We present a new approach that uses smaller RNA sequence and structure seed motifs that do not require long stretches of the secondary structure to be identical. The sequence and structure motifs can be hashed into integer values, which can be compared much faster. An evaluation using the three well understood hammerhead ribozyme families showed that our approach is able to detect 70% to 80% of the hammerhead motifs with a similar false positive rate as the other approaches.

Whenever the performance of new and existing tools should be compared, there is a need for a benchmark data set with an underlying gold standard. BRaliBase is a widely used benchmark for assessing the accuracy of RNA secondary structure alignment methods. In most case studies based on the BRaliBase benchmark, one can observe a puzzling drop in accuracy in the 40% to 60% sequence identity range, the so-called “BRaliBase dent”. We show that this dent is due to a bias in the composition of the BRaliBase benchmark, namely the inclusion of a disproportionate number of tRNAs, which exhibit a very conserved secondary structure. Furthermore, we show that a simple sampling approach that restricts the presence of the most abundant RNA families can prevent such artifacts during the performance evaluation.

ACKNOWLEDGMENTS

I want to thank Robert Giegerich for his supervision of my thesis and the years leading up to that. I am particularly grateful for the freedom given to me to pursue the topics in the direction that interests me, and the fun time during the teaching exchange in Vancouver and at SFU. I also want to thank Peter Unrau for hosting me multiple times at SFU during the exchange program and the discussions about the results of my tools. Thank you also to Markus Nebel for agreeing to review this thesis.

In the beginning of the Ph.D. program, I was very lucky to share an office with Michael Beckstette, who was fun to work and talk with in so many ways and who has given me valuable comments on the topic of string matching. I also want to thank all other people of the former GI group and all people from the DiDy program for a fun time on M-3 and U/V-10.

Last but not least, I would like to thank my family and friends for all your support, but I do not think this is the right place to use extravagant words to tell you how much your support means to me. I just hope that I tell you that often enough in person.

CONTENTS

CHAPTER

1

MOTIVATION & OUTLINE

In order to understand life in the current forms, ranging from simple cells to complex multicellular organisms, we need to understand how they rose in evolution. One of the fundamental processes in modern cells is the expression of heredity information using a complex machinery that forms (enzymatic) proteins from a deoxyribonucleic acid (DNA) template through an ribonucleic acid (RNA) intermediate. Following the current majority opinion among biologists, cells cannot form from inanimate chemicals in one step. Thus, we have to ask: what are the intermediate forms of life, the precellular life? In 1986, Walter Gilbert was the first to use the term *RNA world* [39] that places the RNA molecule in the center of the thoughts, because it can do both: store information in its sequence and catalyze primitive chemical reactions. It shows a way around the "chicken-and-egg" dilemma: DNA needs protein and protein needs DNA. Only later, DNA took over as the primary genetic material while proteins became the prevalent catalysts for the metabolism of a cell. But the transition from the RNA world to our modern-day DNA and protein world was never complete, because there are still a plethora of RNA molecules that fulfill various catalytic functions in present-day cells. On this account, there are several objections to the RNA world hypothesis: RNA is still too complex to arise directly from inanimate chemicals; the catalytic activity is a property of relatively long RNA molecules only; and the catalytic actions of the molecule are fairly limited compared to enzymes [12]. However, regardless of the question whether the RNA world theory is correct, it remains that RNA still plays a central role in all life forms and we have just begun to understand its importance.

Apart from the role in the central dogma as messenger RNA (mRNA), which forms the intermediary from DNA to proteins and carries information in its primary sequence, there are ribosomal, spliceosomal, transfer RNAs (tRNAs), and many more whose structures are essential for the molecule to function properly. Traditionally, the study of cellular and metabolic regulation has focused mostly on proteins, but within the past 20 years several discoveries of novel regulatory RNAs have been made. Micro RNAs (miRNAs), which are involved in the RNA interference mechanism [31], as well as studies in human showing that only 1% of the genome encodes for proteins and two thirds are transcribed [26] have proven that non-coding

RNAs (ncRNAs) constitute the majority of the transcriptional output, at least in eukaryotic genomes. And even in prokaryotes, the domain of regulatory RNAs is expanding [97]. With these discoveries, the interest in RNA structures that determine both, the function of the molecule and the mechanism behind that function, has risen drastically.

Bioinformaticians have contributed their fair share to the progress by developing methods that can be used to answer the question: given an RNA sequence, what are the biologically relevant structures that this sequence can form? Modern-day computers are up to the task of predicting the most likely or a set of most likely secondary structures of a molecule using the free energy based model. The 3D structure of an RNA molecule, on the other hand, that is formed by stacking helices allows for inter-helix interaction or more complex interactions with other molecules. It can be determined in the laboratory with high material costs and time. The computational prediction of these tertiary structures, on the contrary, is less developed, error prone without additional laboratory data, and computationally too demanding, even with modern-day computers. However, the folding of an RNA structure is a hierarchical process; base pairs of the secondary structures are more stable than tertiary structure bonds and will remain in the final structure [58]. This allows us to use the secondary structure as a good abstraction of the full 3D structure of the molecule.

The successful computational prediction of many RNA secondary structures enabled us to ask the question whether we are able to find multiple RNA sequences with similar structures performing similar biological functions. There are two strategies with different requirements that are able to tackle the problem.

First, RNA homology search is the approach trying to answer the question that given a set of homologous RNA sequences, can we find further sequences from other species that belong to the same set. Sets of homologous RNA sequences can be found in the *Rfam* database [79] that holds RNA family models, which are stochastic constructs based on aligned homologous RNA sequences whose similarities on the sequence as well as structure level are considered. The core sequences of each family model are subjected to manual review; therefore, these models are based on human expert knowledge. For families with highly conserved primary sequences, `BLAST` [3] provides a heuristic for genome-wide similarity searches. But usually, structural ncRNAs conserve the secondary structure more than their sequence information. The previously mentioned RNA family models are covariance models (CMs) [30] that can be used to search whole genomes for further members of an RNA family by taking sequence and structure properties of the genomic regions into account.

Second, apart from RNA homology search, *de novo* discovery of RNA motifs shared among structured RNAs is among the most difficult problems in applied bioinformatics. The difficulty is that these motifs evolve by preserving the structure but not the sequence; therefore, computationally more demanding structure prediction and comparison methods have to be employed to find potentially rare motifs within a huge set of genomes. For this, we have to

differentiate alignment-based and alignment-free methods. Alignment-based methods try to exploit existing multiple sequence alignments using a sliding window and in each window a consensus structure is computed. In the end, various post-processing steps are performed and the windows are ranked by alignment score. A major drawback of these methods is that they require an alignment that is already compatible with the structure yet to be found. Those alignments are usually hard to obtain for low similarity regions. Apart from alignment-based methods, tools like `CMfinder` [115] search a set of unaligned sequences by selecting candidate subsequences based on minimum free energy and perform a pairwise structural alignment of two subsequences if they share common motifs according to `BLAST`. These alignments are used subsequently to build a new CM that is reapplied to the genomes to find further motif occurrences. `CMfinder` has been used extensively to identify novel ncRNAs in a variety of whole genome screens, but there are two major shortcomings: first, these screens are extremely time consuming. A screen in 2,946 sets containing a few hundred nucleotides of unaligned sequences upstream of the start codons of genes in 41 Firmicute species [113] took the whole `CMfinder` pipeline one CPU year to finish. Moreover, other screens in vertebrates have used decades and centuries of CPU time [93, 101]. The second main issue of all *de novo* screening methods is the high false positive rate around 50% [40].

In this thesis, we will start with a genome-wide screen for RNA motifs in 4,667 viral reference genomes in Chapter 3. Viruses are an excellent fit when it comes to the discovery of shared RNA motifs without the involvement of conserved primary sequence because of their high mutation rates. Their interactions with host cells are essential for the infection and the subsequent replication of the viral genomes. These interactions are often based on specific RNA motifs shared between different evolutionary distant viruses. We observe that convergent evolution is a potential mechanism that explains similar motifs in taxonomically distant viruses that infect common hosts by interacting with their cellular components. This is supported by the fact that for those specific RNA motifs, similar selection criteria prevail. In this regard, the hammerhead ribozyme is a well-studied example [44]. Based on the large number of viruses in our data set and the high computational costs of methods like `CMfinder`, we are going to develop faster methods for all-against-all motif search.

Our first approach uses exact structure matching to identify common motifs shared among viruses, which might represent examples of convergent evolution, based on well-known string matching techniques. We were able to find common motifs of some RNA families, like tRNAs, which are known for very conserved secondary structures, but also detected two shortcomings. First, the exact string matching using suffix trees or arrays does not exploit the properties of the well-formed RNA structure strings; a bottom-up processing of the tree also returns matches between multiple viruses that are not well-formed RNA structures but can be any possible substring of a secondary structure. Therefore, we designed a novel data structure, the viable suffix tree, an extension of the classical suffix tree, as a potential solution that stores only those

suffixes that are well-formed RNA structures. In general, our new data structure allows the use of an arbitrary grammar language that specifies all valid words that can be inserted into the tree. Second, the limitation with exact structure matching is the need for long common stretches of secondary structures that are not allowed to have a mismatch at any position. We need to allow small mismatches to find more potential targets but do not want to use computationally too expensive methods for sequence and structure comparisons. Consequently, we developed a new approach that uses smaller RNA sequence and structure seed motifs that do not require long stretches of the secondary structure to be identical. In contrast to `CMfinder`, primary sequence conservation is optional and we aim for much faster computation time by not requiring any time consuming sequence and structure alignment; this will be replaced by a constant time comparison of the hash values of our seed motifs.

While looking for a suitable benchmark set for the algorithms, we ran into an open problem in RNA bioinformatics – an unexplained behavior of algorithms on the most widely used benchmark in the field, the BRaliBase [33, 112]. In this analysis, we show that the data set is biased towards a certain RNA family which distorts the results. Furthermore, we show that a simple sampling approach can restore the right proportions and formulate rules that prevent biased benchmark data sets.

Outline In Chapter 2, we start by introducing the general background and definitions that are used throughout the thesis while background and definitions that are only needed for a specific topic are introduced in the respective chapter. Next, Chapter 3 describes the exact matching of the viral reference genomes and gives an overview of the common motifs that we were able to identify. Chapter 4 introduces the viable suffix tree and compares it to the classical suffix tree in terms of memory consumption and worst-case runtime. The disadvantages and consequences of unbiased benchmark data sets are shown in Chapter 5, using the example of the most widely used RNA benchmark BRaliBase. Our new seed-based approach for RNA motif discovery and its benchmark is presented in Chapter 6. In the final Chapter 7, we will discuss our results.

CHAPTER

2

BACKGROUND

Contents

In this chapter, we will introduce the most important concepts and definitions that will be used throughout the thesis. These include definitions of alphabets, strings, words, and languages; as well as data structures like the suffix tree or array including their construction; and representations of biological molecules, like RNA, on which the data structures can and will be used.

2.1. General definitions

Throughout the thesis, we will use the following writing and naming conventions for variables: those that hold only one value, e.g. integers and characters, will be written in italic and their name will start with a lowercase character, e.g. i; variables that might hold multiple values,

such as sets, lists, graphs, and trees, will also be written in italic, but their name will start with an uppercase character, e.g. G; nodes in graphs or trees will be written using typewriter font and their name will start with a lowercase character, e.g. v. Elements of sets and lists can be accessed using the [] operator.

Definition 2.1. Let Σ be a non-empty finite set of symbols[1] called the *alphabet*.

By convention, we will use the symbol Σ in any context solely for the purpose of representing the alphabet.

Definition 2.2. The *cardinality of a set*, like the alphabet Σ, is denoted as $|\Sigma|$ and represents the number of objects that are in Σ.

Definition 2.3. A *string* is a finite sequence of symbols from Σ. With ϵ we denote the *empty string*, the only string that does not contain any symbols from the alphabet. Furthermore, the length of the empty string is defined as 0.

The notation for the *length of a string* S is $|S|$.

Definition 2.4. By Σ^n we denote the *set of strings of length* n *over* Σ. The *set of all non-empty strings over* Σ is denoted by Σ^+. If the empty string is included, then the set is denoted by Σ^*.

Definition 2.5. Let S be a string from Σ^n. The first symbol of S is located at *position* 0 and the last symbol is at positioned at $n-1$. Let T be a string from Σ^m. The *concatenation* of the strings S and T is the new string starting with the n symbols of S followed by the m characters of T and is written as ST or $S + T$.

Definition 2.6. Let S, T, U, and V be strings such that $V = STU$. The string T is called *substring* of V, S is called *prefix* of V, and U is called *suffix* of V. The prefix of S ending at position i is denoted by S_i. The suffix of S starting at position i is denoted by S^i.

Definition 2.7. A *substring* of S that starts at position $i \in \{0, \ldots, n-1\}$ and ends at position $j \in \{i, \ldots, n-1\}$ is denoted by S_j^i. Substring notations in which $j < i$ are defined as ϵ.

2.2. (Context-free) grammars and languages

The definitions in this section are be based on [51]. The field of formal grammars and languages is based on the work of Noah Chomsky, who aimed to define and characterize the structure of natural languages in precise mathematical rules but eventually did not fully achieve his ambitious goals. Similar rules can also be defined for computer programming languages so that the compiler can use syntax analysis to determine if a statement in the programming code is syntactically correct and if so, how to execute this statement.

[1] In the thesis, the word *character* is used as synonym for alphabet symbol.

Definition 2.8. Given the alphabet Σ, a *language* L over Σ is a set of strings such that $L \subseteq \Sigma^*$. All members of L are called *words* of the language.

It is important to state that although the number of strings in a language can be infinite, these strings are restricted to the symbols from the finite alphabet.

The question that arises is: how can we define this language precisely so that we can decide whether a string is part of the language or not? Since the number of strings in the language can be infinite, simple enumeration of the language set does not solve this task. There are several ways to define different languages, such as regular expressions, pattern systems, and grammars. In this thesis, we focus on grammars, as they are the most powerful and general system for representing languages based on formal notations.

Definition 2.9. A grammar is quadruple (Σ, N, S, P), where:

- Σ is a finite set of *terminal symbols* (alphabet),
- N is a finite set of *nonterminal symbols*,
- $S \in N$ is the *start symbol*, and
- P is a set of production rules of the form $\alpha \rightarrow \beta$, where
  $\alpha \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $\beta \in (\Sigma \cup N)^*$.

The left hand side of the production rule (α) is a string of terminal symbols and at least one nonterminal symbol while the right hand side (β) is a string of an arbitrary number of terminal and nonterminal symbols.

Definition 2.10. Let $G = (\Sigma, N, S, P)$ be a grammar and $\gamma_1, \gamma_2 \in (\Sigma \cup N)^*$. We say that γ_1 *directly derives* γ_2, written as $\gamma_1 \underset{G}{\Rightarrow} \gamma_2$, if $\gamma_1 = \sigma\alpha\tau$, $\gamma_2 = \sigma\beta\tau$, and $\alpha \rightarrow \beta$ is a production rule in G. Furthermore, we say γ_1 *derives* γ_2, written as $\gamma_1 \underset{G}{\Rightarrow}^* \gamma_2$, if there are (zero or more) $\delta_1, \ldots, \delta_n \in (\Sigma \cup N)^*$ such that $\gamma_1 \underset{G}{\Rightarrow} \delta_1 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} \delta_n \underset{G}{\Rightarrow} \gamma_2$. The sequence $\gamma_1 \underset{G}{\Rightarrow} \delta_1 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} \delta_n \underset{G}{\Rightarrow} \gamma_2$ is called *derivation* of γ_2 from γ_1.

In order to define the language of a grammar, we use derivations. We first expand the start symbol by all production rules in P that it is part of on the left hand side. The resulting string is further expanded by replacing the remaining nonterminal symbols by the respective right hand side of the production rule until the string consists entirely of terminal symbols. The language of the grammar contains all strings that can be expanded in this way.

Definition 2.11. Let $G = (\Sigma, N, S, P)$ be a grammar. The *grammar language* of G is denoted as $L(G)$ and defined as $\{w \in N \mid S \underset{G}{\Rightarrow}^* w\}$.

The *Chomsky hierarchy* divides grammars into four classes based on the restrictions imposed on the form of the production rules.

Definition 2.12. Let $G = (\Sigma, N, S, P)$ be a grammar.

- G is called *unrestricted (Type-0)* grammar.
- G is called *context-sensitive (Type-1)* grammar if the right hand side of the production rule is from $(\Sigma \cup N)^+$.
- G is called *context-free (Type-2)* grammar if the left hand side of the production rule is from N.
- G is called *(right) regular (Type-3)* grammar if all productions are of the form $A \to cB$ ($A = B$ allowed), $A \to c$, and $A \to \epsilon$ with $A, B \in N$, and $c \in \Sigma$.

The difference between the context-sensitive and the unrestricted grammar is that the right hand side of the production rule might be the empty symbol in Type-1 grammars. We will focus on context-free grammars (CFGs), because their set of allowed production rules is enough to cover the complexity of the problems in this thesis. In contrast to context-sensitive grammars, they only allow exactly one nonterminal symbol on the left hand side of the production rule. Regular grammars only allow three specific types of production rules, which are not sufficient for the problems that will be covered here.

Now, we can recall the previous question: given an alphabet Σ, a grammar G over Σ, and a string $S \in \Sigma^*$. Is $S \in L(G)$? We will refer to this as the *word problem*. Luckily, the word problem for CFGs is decidable using the Cocke–Younger–Kasami (CYK) algorithm [116][2]. This algorithm uses dynamic programming (DP) to fill a triangular table T, where every cell $T_{i,j}$ stores a set of variables A such that $A \underset{G}{\Rightarrow}^* S_i^i \dots S_j^j$. The algorithm starts filling all cells $T_{i,i}$ by checking whether there is a single rule that can create the string S_i^i. Afterwards, the second row with all strings of length two is filled, and so on, until the cell of the full length string is reached. Note that the CYK algorithm also computes the cells for all possible substrings of S. Thus, we can decide if S_j^i is in $L(G)$ by checking whether S_j^i is part of the set of the cell $T_{i,j}$, which can be built in $\mathcal{O}(|S|^3)$ time.

2.3. The RNA molecule

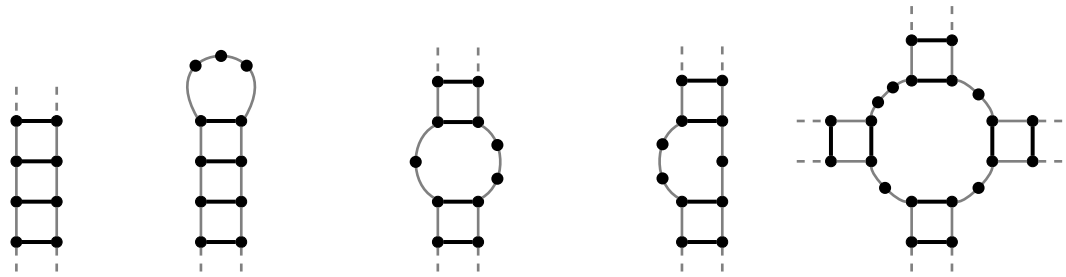
RNA is a linear chain molecule composed of four different nucleotides that is involved in various cellular processes. The chain consists of the bases Adenine (A), Cytosine (C), Guanine (G), and Uracil (U) in connection with a ribose and an additional phosphate group. The nucleotides are

[2] Cocke's work was never published and Kasami's algorithm only appeared in an internal US-Air-Force document [51].

strung together via the sugar-phosphate backbone. While DNA is more often found in nature as double-stranded molecule, RNA usually occurs single-stranded with the notable exception of some viral species. The molecule is known for its property to fold back onto itself due to base pair complementarity that allows two bases to be connected to each other using hydrogen bonds. The most commonly used base pairing models are the *Watson-Crick pairs* G·C and A·U as well as the *wobble base pair* G·U. Whenever multiple base pairs are stacked on top of each other, they form a helix that stabilizes the structure of the molecule. On the other hand, the formation of a stack always includes the formation of an unpaired and accessible loop region that destabilizes the structure of the molecule.

The RNA molecule is widely known for its involvement in the cellular machinery, especially in the protein biosynthesis. The central dogma of molecular biology places the RNA in the center between DNA and protein. The DNA is transcribed into mRNA that serves as an intermediate information transporter until the molecule reaches the ribosome, where it is translated into protein. Usually, the mRNA is solely used as linear molecule that carries information without any further structural formation. During the translation step, another RNA molecule, the tRNA, acts an adapter that helps with the incorporation of the correct amino acid into the peptide chain of the newly synthesized protein. The translation process takes places at the ribosome, which is a highly complex cellular machine that consists of several different ribosomal RNAs (rRNAs) and proteins. The rRNAs that are involved in the translation machinery were one of the first examples that proved the catalytic activity of an RNA molecule.

For a long time, it was believed that the main purpose of the RNA molecule was the role as messenger molecule between DNA and proteins, but discoveries over the last decades caused a paradigm shift in molecular biology. At the latest, with the discovery of the RNA interference mechanism and the miRNAs, which are small regulatory RNAs processed from a hairpin of a longer precursor molecule, a new view on RNA set in [31]. Recent studies in human have shown that only 1% of the genome encodes for proteins and two thirds are transcribed [26], so an important question remains: what are the remaining RNAs and what is their function? In the past two decades, we have seen a tremendous effort on detecting novel ncRNAs. The *Rfam* database in the current version 12.2 [79] consists of 2,588 different RNA families, which break down into broad functional classes, such as ncRNA genes, structured cis-regulatory elements, like the internal ribosome entry site (IRES), and self-splicing RNAs. The lengths of these ncRNAs can vary greatly from 20 to thousands of nucleotides for long non-coding RNAs (lncRNAs). The most prominent example of a ncRNA that is present in several phylogenetically distant organisms is the hammerhead ribozyme, which amongst others catalyzes the scission of its own backbone without co-factors and will serve as our search target in Section 6.3. Most ncRNAs have a conserved secondary structure that is better preserved than their nucleotide sequence, which is why we will focus on the structure in the experiments of this thesis.



(a) Stack **(b)** Hairpin loop **(c)** Internal loop **(d)** Bulge loop **(e)** Multiloop

Figure 2.1.: Possible RNA secondary structure motifs.

2.3.1. Structural levels

The structure of RNA molecules can be considered in three different levels with increasing complexity, molecular- as well as representation-wise. In general, an RNA molecules folds hierarchically starting with the base pairs and later the helical arrangements define a precise 3D structure [58]. Next, we introduce all three levels.

2.3.1.1. Primary structure

The primary structure of the RNA is the string that represents the sequence of subunits of the molecule, the nucleotides with different bases. As the RNA is composed of different nucleotides we define the alphabet as follows:

Definition 2.13. The set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$ is the alphabet of the four RNA bases Adenine, Cytosine, Guanine, and Uracil and is called the *RNA alphabet*. An *RNA sequence* is any string over $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}^+$.

The information content of the primary structure is based on the arrangement of the nucleotides in a molecule. In mRNAs, the primary structure of the RNA molecule is used to transfer information from the DNA to the protein level. The base sequence of the mRNA is usually a copy of the primary sequence of a gene and is later used during protein biosynthesis to assemble the amino acids of a protein in the correct order.

Furthermore, there are different classes of viruses whose genetic material is either single- or double-stranded RNA, among them prominent examples such as the Ebola viruses, the Hepatitis C virus, and the Influenza viruses. Here, the information for protein biosynthesis is encoded in the genomic RNA.

2.3.1.2. Secondary structure

The RNA sequence constitutes the basis for the secondary structure and its information content is extended by folding information. Single-stranded RNA molecules usually contain many

complementary regions that may form double helix structures when the molecule folds back onto itself. The resulting pattern of helices and interrupting unpaired regions is called the secondary structure.

Also, another RNA molecule and its secondary structure play a crucial role in the last stage of protein biosynthesis: the tRNA. While the mRNA specifies which amino acids are going to be incorporated into the protein's polypeptide chain, the tRNA helps to translate between the DNA sequence and the amino acids. The anticodon loop of the secondary structure of the tRNA holds three bases that form base pairs with three bases of the mRNA. These three bases specify exactly which amino acid is going to be added to the protein.

For building the secondary structure, we will consider the Watson-Crick base pairs C·G and A·U as well as the wobble base pair G·U. These three base pairs are called *canonical*. All other possible pairings are called *non-canonical* and will not be considered in this thesis.

Definition 2.14. Let R be an RNA sequence that can form a structure. A pair (i, j) with $0 \leq i < j \leq |R| - 1$ is called *base pair* if the pairing R_i^i and R_j^j is canonical. Two base pairs (i, j) and (k, l) are called *nested* if and only if $i < j < k < l$ or $i < k < l < j$. On the other hand, the pair is called *crossing* if and only if $i < k < j < l$.

Definition 2.15. Let R be an RNA sequence. The *secondary structure* S of R is defined as a set of nested canonical base pairs.

We do not consider any motifs in which a position can be part of two base pairs, such as base triplets, or kissing hairpins. An abstraction of possible secondary structure motifs is presented in Figure 2.1. On the other hand, crossing base pairs, also known as *pseudoknots*, will be considered as tertiary structure. Graphically, they can be described as two stem structures in which a part of one stem is enclosed between two parts of the other stem. Even though they should strictly be classified as secondary structure, most cubic time DP algorithms for predicting secondary structure cannot handle pseudoknots accurately. Therefore, it is widely agreed upon that they are seen as first step towards a prediction of the tertiary structure of RNA molecules. Usually, there are even more restrictions imposed on RNA secondary structures, e.g. hairpin loops must be at least of length three. See Section 2.3.2 for more details on predicting secondary structures and their restrictions.

In order to generate all possible structures of an RNA molecule, CFGs are the method of choice.

Example 2.1. The following CFG G_5 includes all possible sequences of the RNA alphabet in its language: $(\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}, N = \{B\}, S = B, P = \{B \rightarrow xB \mid xB\hat{x}B \mid \epsilon\})$. Some of the language members can be generated multiple times by using different derivations. In this case, every derivation represents a different RNA structure, even though the underlying RNA sequence might be the same. The production rules are written in a short form, where the nonterminal

symbol x represents any base from the alphabet and the rule $xA\hat{x}A$ implies that bases x and $\hat{x}$ must be able to form a canonical base pair. The first rule shows the generation of an unpaired base, the second rule generates a base pair, and the last rule allows an empty RNA sequence.

There exist multiple CFGs for the purpose of generating nested RNA structures, as discussed in [29], but G_5 is our choice, because it is compact and structurally unambiguous. A grammar is *structurally unambiguous* if every structure has only one derivation. This property proves itself useful in secondary structure prediction using stochastic context-free grammars (SCFGs). The relationship between RNA structures and derivations is illustrated in Figure 2.2c.

The current representation of secondary structures as sets of base pairs make it difficult to grasp the partitioning into smaller substructures at a first glance. *Dot-bracket strings*[3] provide an equivalent and machine-readable alternative. These strings represent unpaired bases by dots and a base pair (i, j) by matching brackets such that the opening bracket is at the i^{th} position and the closing bracket is at the j^{th} position.

Example 2.2. Let R be an RNA sequence. B is a set of possible base pairs and S is the equivalent dot-bracket string.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
R	=	C	G	A	C	G	G	G	G	U	A	U	G	A	C	G	C	G	U	C	G
S	=	(	(	(	(	.	.	.	)	)	.	(	(	.	.	.	)	)	.	)	)

$B = \{(0, 19), (1, 18), (2, 8), (3, 7), (10, 16), (11, 15)\}$

More secondary structure representations are presented in Figure 2.2.

Inspired by G_5, we can define another CFG that describes RNA structures in dot-bracket notation.

Example 2.3. Context-free grammar G_S, on the other hand, does not take RNA sequences into account but rather generates all possible RNA secondary structures in the dot-bracket notation: $(\Sigma = \{(,), .\}, N = \{B\}, S = B, P = \{B \rightarrow .B \mid (B)B \mid \epsilon\})$. As before, the first rule shows the generation of an unpaired base, the second rule generates a base pair, and the last rule allows an empty structure.

All possible nested RNA structures in dot-bracket notation are part of $L(G_S)$.

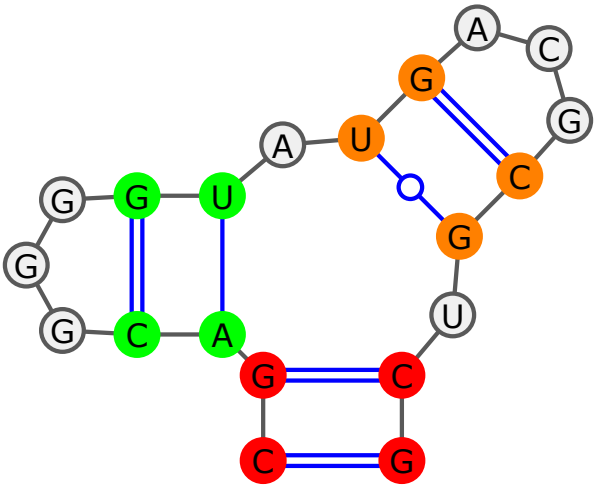
In the last step, we can unite G_5 and G_S into a CFG that reads an RNA sequence and a structure at the same time.

Example 2.4. The following CFG G_{5S} can generate all nested RNA secondary structures together with their underlying RNA sequences:

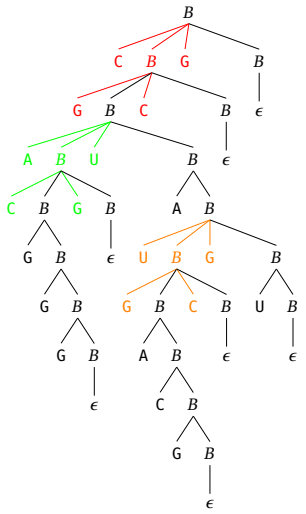
[3] Also referred to as *Vienna notation*, as they were first introduced in the `ViennaRNA` package.

C G A C G G G G U A U G A C G C G U C G
((((. . .)) . ((. . .)) .))

(a) RNA sequence and structure in dot-bracket style



(b) 2D visualization using VARNA [25]



(c) Derivation tree of G_5

Figure 2.2.: Different RNA sequence and structure representations in which the multiloop and the two hairpin loops are highlighted using different colors.

$(\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}, \mathtt{.}, (,)\}, N = \{B\}, S = B, P = \{B \longrightarrow x.B \mid x(B\hat{x})B \mid \epsilon\})$. For this grammar, the RNA sequence and structure characters are used interleaved so that they can be represented in one single string. This means that every secondary structure of any RNA sequence can be depicted in a unique way using G_{5S}. Let R be an RNA sequence of length n and S be a secondary structure of the same length. The interleaved string that represents both, sequence and structure, is of the form $R_0^0 S_0^0 \ldots R_{n-1}^{n-1} S_{n-1}^{n-1}$.

This combined grammar allows us to test whether any given structure S is *compatible* to sequence R, meaning that R can fold into structure S and S must represent a well-formed nested structure. Both sequence and structure are compatible if and only if their interleaved string is part of $L(G_{5S})$. The language of the combined grammar has the same power as the language of G_5, because G_5 is structurally unambiguous, which means that every structure generated by G_5 is uniquely represented by exactly one dot-bracket string. Note, we could also use coupled context-free grammars (CCFGs) [53] and rewrite rules for the same purpose, in which two CFGs are seen as input/output data structures such that the input RNA sequence can be converted to a secondary structure string as output.

2.3.1.3. Tertiary structure

The next organizational level is the three dimensional, also called tertiary, structure. Folding of an RNA structure is a hierarchical process; base pairs of the secondary structure will remain in the final structure and their associations through van der Waals forces, additional base pairs, and unusual pairs involving hairpin, internal, and bulge loops are described further by the tertiary structure. Tertiary structures can be experimentally obtained by time consuming biophysical methods (X-ray diffraction data or NMR couplings) or biochemical approaches (chemical probing or enzymatic attack) [109]. High throughput *in silico* methods for predicting RNA tertiary structures are still less developed, error prone without additional laboratory data, and computationally too demanding, even with modern-day computers. Therefore, we have decided to use the secondary structure of an RNA molecule as a good abstraction of the full 3D structure.

2.3.2. Prediction of secondary structures

In research on RNA sequences, secondary structure analysis is the prominent task of finding base pairing interactions within a single or a set of interacting RNA molecules. This structural bioinformatics task needs to take the primary sequence as well as interactions between nucleotides into account. Its importance and practical relevance is based on the fact that there are not many experimentally validated RNA structures available, but the number of potentially interesting sequences is high.

The first steps in research on predicting RNA secondary structures were taken using simple scoring schemes maximizing the number of base pairs [82]. The algorithm by Ruth Nussinov uses DP to fill all matrix entries $M(i, j)$ for RNA sequence S as follows:

$$M(i,j) = \max \begin{cases} 0 & \text{if } j \leq i \\ M(i, j-1) & \text{if } j > i \\ \max\limits_{i \leq k < j-1} \left(M(i, k-1) + M(k+1, j-1) + 1\right) & \text{if } S_k^k \text{ and } S_j^j \text{ can base pair} \end{cases}$$

Here, we present the improved version of the original algorithm, because the original one is highly ambiguous and produces the same secondary structure multiple times. The value in cell $M(i, j)$ is defined as the maximum number of base pairs in substring S_j^i. In this algorithm, the nucleotide at position j is either paired to the nucleotide at position k or it is unpaired. If it is paired, the sequence is divided into two subsequences: the inner part ranging from $k + 1$ to $j - 1$ and the part before the base pair from i to $k - 1$. The matrix can be calculated either in row by row or column by column fashion as long as the values for short subsequences are calculated first. The end result for the complete sequence is stored in cell $M(0, |S| - 1)$. The complete matrix can be calculated in $\mathcal{O}(|S|^3)$.

The algorithm by Zuker and Stiegler [117] was the first one that was capable of computing the minimum free energy structure with DP using the loop-based nearest neighbor energy model. It does so by taking the stabilizing effect of base pair stacking and the destabilizing effect of unpaired loops into account. For minimizing the free energy of a molecule, the energy parameters and model from [73, 102] are commonly used; they were updated quite frequently over the last twenty years. The idea is to uniquely decompose the structure into loops and external bases such that the free energy of a structure can be described as the sum of the energy contributions of the different loops at a given temperature. The loops are composed of a stem that consists of base pairs. The algorithm of Zuker and Stiegler can be computed using the following two tables:

$$W(i,j) = \min \begin{cases} W(i+1,j) \\ W(i,j-1) \\ V(i,j) \\ \min\limits_{i<k<j} (W(i,k) + W(k+1,j)) \end{cases}$$

$$V(i,j) = \min \begin{cases} hairpinLoop(i,j) \\ stack(i+1,j-1) + V(i+1,j-1) \\ \min\limits_{\substack{i<i'<j'<j \\ i'-i+j-j'>2}} (internalLoop(i,i',j',j) + V(i',j')) \\ \min\limits_{i+1<k<j-1} (W(i+1,k) + W(k+1,j-1)) \end{cases}$$

The first two cases of table W expect the nucleotides at positions i and j to be unpaired. In the third case, i and j are paired while in the last case, i and j are potentially paired to other nucleotides. In table V, on the other hand, the first case describes a hairpin loop between i and j while the second case scores a stacked pair. In the third case, it is either an internal or bulge loop and in the fourth case, a multiloop is constructed. The algorithm has a time complexity of $\mathcal{O}(|S|^4)$. Restricting the size of an internal loop to a constant value leads to an overall runtime of $\mathcal{O}(|S|^3)$.

Usually, if we are interested in analyzing folding patterns of RNA sequences, the single minimum free energy structure and its energy have no relevance, because the energy terms that are used to compute the minimum free energy structures are not exact, since some RNAs do not always fold into the thermodynamically optimal structure or even have two or more active structures, like riboswitches.

Definition 2.16. The *folding space* of an RNA sequence S is denoted by $F(S)$.

McCaskill's method [74] was the first that could analyze the complete folding space such that the importance of a structure could be determined by using a partition function.

Definition 2.17. Let S be an RNA sequence. The *partition function* of the folding space $F(S)$ of an RNA sequence S is defined as

$$Q = \sum_{X \in F(S)} \exp\left(\frac{-E(X)}{RT}\right),$$

where $E(X)$ is the free energy of structure X, R the universal gas constant, and T the temperature in Kelvin.

Using this, the probability of a single RNA structure can be computed.

Definition 2.18. The *probability of an RNA structure X* is defined as

$$Prob(X) = \frac{\exp\left(\frac{-E(X)}{RT}\right)}{Q},$$

where $E(X)$ is the free energy of structure X, R the universal gas constant, T the temperature in Kelvin, and Q the partition function.

Note, the probability of a particular structure increases proportional to the decrease in free energy and vice versa. Therefore, the probability does not provide a different view on the importance of an RNA structure. An elegant abstraction of the folding space is introduced in Section 3.3.1.

2.3.3. Identifying potentially functional RNAs

Until now, we discussed the different structural levels of the RNA and how to predict them using computational models and measured energy parameters. Identifying structured ncRNAs that have a potential function in an organism can be done in two ways: homology search using existing models of known RNA families, or *de novo* search for so far unknown RNA molecules.

In the discipline of homology search, there are two kinds of tools: the ones adapted to find a specific class or family of RNAs, and the more general ones that can cope with multiple RNA family models. The first group includes knowledge-based approaches, like `RNAMotif` [70], `Structator` [76], and `RaligNAtor` [77], that allow the user to define a sequence and structure pattern (using a specific syntax) that will act as query against a database. Small structural entities like hairpin loops are searched in the database and those matches are chained, up to a user-defined edit distance threshold. Furthermore, software with an appealing graphical user interface, like `Locomotif` [90], can help humans to specify RNA patterns that can be used

to generate a thermodynamic matcher [92], which performs restricted RNA folding of target sequences by allowing them to fold only into the predefined motif. In the second group, the `Infernal` software suite [80] is the reference implementation for using CMs to search sequence databases for homologs of annotated RNA families from the *Rfam* database [79]. `Infernal` uses the structure annotated multiple sequence alignment of the RNA family to build a profile with a position-specific scoring system similar to SCFGs. We will use the software in Section 3.4 to identify known RNA motifs from the results of the exact structure matching approach.

In *de novo* discovery, the goal is to capture similarities in sequences that are not related at first glance, because structurally constrained RNAs evolve while conserving the structure and not the nucleotide sequence. For this, we have to differentiate alignment-based and alignment-free methods. Alignment-based methods try to exploit existing multiple sequence alignments, but they require an alignment that is already compatible with the structure yet to be found.

`RNAz` [42, 106] is an example of a tool that needs a multiple sequence alignment as input. It folds the alignment with `RNAalifold` [48], which generates a consensus secondary structure. Next, `RNAz` measures structural conservation by calculating the ratio of the consensus folding energy and the energies of the single folds of every sequence in the alignment. Furthermore, it measures the thermodynamic stability of a sequence by comparing its minimum free energy to random sequences of the same length and base composition. The alignment is then characterized as "functional RNA" or not, based on the results of the two defined criterions.

The tool `CMfinder` [115] has been used extensively to identify novel ncRNAs in a variety of whole genome screens [101, 107, 114] and functions in three distinct steps: identification of sequences for an initial alignment, (re-)build the CM so that it fits better to the alignment, and use the CM to realign the sequences. The idea can be classified as an expectation maximization approach that does not need an alignment as input. Local regions of conserved primary sequences are searched using `BLAST` [3], which serve as anchors for the structural alignment using the tree-edit algorithm [49]. Based on this alignment, a CM is built, similar to the process `cmbuild` of the `Infernal` package. The realignment in the next step is computed using the parameters learned in the process of building the CM. But there are two major shortcomings: often, *de novo* screening methods show high false positive rates around 50% [40]. Furthermore, a screen using `CMfinder` in 2,946 sets containing a few hundred nucleotides of unaligned sequences upstream of the start codons of genes in 41 Firmicute species [113] took the whole `CMfinder` pipeline one CPU year to finish. Moreover, other screens in vertebrates have used decades and centuries of CPU time [93, 101].

2.4. Index data structures

Scanning a large amount of input strings for common patterns can easily lead to combinatorial explosion in terms of the number of comparative operations that have to be performed. For

large input sets, the sheer amount of data makes an online search unfeasible. An index is a data structure that is prepared before the search process takes place and organizes the input data in a way that facilitates efficient queries for search patterns. The queries can be of different types, e.g. checking the occurrence of one or multiple patterns in the input set, retrieving the longest common subsequences of multiple strings in the input set, or retrieving maximal repeats of a certain minimal length. Hereafter, we introduce two of the most prominent data structures for exact string matching: the suffix tree and the (enhanced) suffix array.

2.4.1. Suffix trees

Before introducing the construction of a suffix tree, we want to first define the data structure. But before defining the suffix tree, we will introduce some functions on general trees that will prove useful later.

Definition 2.19. Let T be a rooted, directed, edge-labeled tree and u be a node (internal or leaf) in T. The *path* of u, denoted as $path(\mathsf{u})$, is the ordered list of nodes that have to be passed from the root to u. Further, the *label* of u, denoted as $label(\mathsf{u})$, is defined as the resulting string from the ordered concatenation of all edge labels in $path(\mathsf{u})$. Last, the *depth* of u, denoted as $depth(\mathsf{u})$, is defined as $|label(\mathsf{u})|$, meaning the length of its label.

Using this formalism, we can now introduce the suffix tree.

Definition 2.20. A *suffix tree* T of string S is a rooted, directed, edge-labeled tree with the following properties:

1. The tree has exactly $|S|$ leaves that are labeled by $0, \ldots, |S| - 1$.
2. Each internal node, except the root, has at least two children.
3. Each edge is labeled with a non-empty substring of S.
4. No two edges out of the same node begin with the same character.
5. For leaf node l, $path(\mathsf{l})$ is equal to suffix S^l.

In order to ensure that no suffix is a prefix of any other, a *sentinel* character is added to the end of the input string S. The sentinel character must not be part of the alphabet; usually, the character \$ is used.

Next, we introduce an naïve construction algorithm that adds each and every suffix step by step into the tree. Assume S is the string we want to index and it has length n. In the first step, the tree is empty and the suffix S^0 is added as a leaf. Following, the suffixes S^1 to S^{n-1} are successively added into the tree by following the tree from the root node along the longest path that matches a prefix of this suffix. Either this path already ends at an internal node, then

the remaining characters of the suffix will be added as a leaf at this point, or the path ends in the middle of an edge, then a new internal node that splits the current edge into two parts will be introduced. The remaining characters of the suffix will be added as a leaf that is connected to the new internal node. Every suffix is now present as an unique leaf in the tree and the method takes $\mathcal{O}(n^2)$ time, because we have to insert n suffixes and during each suffix insertion the algorithm has to compare $\mathcal{O}(n)$ characters to follow the longest matching path in the tree.

The first algorithm that was able to construct suffix trees in linear time for a constant alphabet was given by Weiner in 1973 [108]. It reads the input string in reverse order and successively inserts all suffixes from short to long. However, the algorithm was denounced as overly complicated [36] using obscure terminology [35]. Shortly thereafter in 1976, McCreight published his simplified algorithm for the construction of suffix trees in linear time [75]. Later in 1995, Ukkonen presented a related approach that is also online, which means that not all input characters have to be present during the start of the computation [104]. Giegerich *et al.* showed that Ukkonen's and McCreight's algorithms perform the same abstract operations of edge splitting and adding leaves in the same order [35, 36]. We will explain Ukkonen's algorithm in more detail in Section 4.4, where we will alter it to insert only suffixes that are words in a given grammar.

In 2003, Giegerich *et al.* introduced the *lazy suffix tree* whose nodes are created only on demand [37]. The tree was created to utilize the write-only top-down (wotd) algorithm presented in [35] that creates the suffix tree top-down by expanding disjunctive subtrees independently. Each subtree below a branching node u is determined by the set of all suffixes that have $\mathit{label}(\mathsf{u})$ as prefix. The algorithm starts at the root by evaluating all suffixes. The set of all suffixes is divided into groups depending on the first character of each suffix. The longest common prefix of all suffixes in one group defines the label of the new edge that will be created for this group. Subsequently, all remaining nodes can be evaluated recursively in a top-down manner. The worst-case runtime of this algorithm for a string S is $\mathcal{O}(|S|^2)$, but the authors showed that the lazy suffix tree can significantly improve the practical runtime, amongst others for biological data.

2.4.2. (Enhanced) suffix arrays

Suffix trees are among the most important and most used data structures in computational biology and all disciplines that makes frequent use of string processing. However, the space requirements for suffix trees is an obstacle in large scale applications such as pattern matching using genomics data. In 1990, Manber and Myers [71, 72] introduced the *suffix array* data structure as a space efficient alternative to suffix trees. Suffix tree implementations may require 13 to 15 bytes per input character [65], whereas the suffix array only requires 4 bytes per character in its most basic form. Both values assume that an integer will be represented

by 4 bytes. Suffix arrays can be either constructed in linear time by using the suffix tree as template [43] or in a direct way from the input sequence in linearithmic time, as shown in [72]. Later in 2003, three independent studies [59, 62, 64] showed that suffix arrays can be constructed directly in linear time, too.

The basic suffix array, usually denoted as *Suftab*, for a string S is a table with $|S|$ entries that specifies the lexicographic ordering of all suffixes. The sentinel character is assumed to have a value smaller than any character from the alphabet. The suffix array can be used to find all occurrences of a pattern P in string $|S|$ in $\mathcal{O}(|P| \log |S|)$ time. This task can be solved by tackling the problem of finding all suffixes that start with P; and since all of the suffixes are in lexicographic order in the array, two binary searches that find the start and end point of the occurrence interval are necessary.

In addition to *Suftab*, Manber and Myers [71, 72] defined *Lcptab*, where $Lcptab[i]$ holds the length of the longest common prefix of $S^{Suftab[i-1]}$ and $S^{Suftab[i]}$. This table can be computed in linear time from *Suftab*. *Lcptab* helps to improve the runtime of pattern matching to $\mathcal{O}(|P| + \log |S|)$ by reducing the necessary steps for the binary search, because now no character of P needs to be compared to any character of S more than once.

In 2004, Abouelhoda *et al.* [2] introduced the concept of *enhanced suffix arrays* in order to replace all algorithms that use the suffix tree data structure with an algorithm that works on enhanced suffix arrays and solves the problem in the same time. The suffix tree applications require one or more of these traversal types: bottom-up (e.g. maximal repeats), top-down (e.g. pattern matching), and top-down plus suffix links (e.g. matching statistics for the approximate string matching problem [22]). Additionally, the enhanced suffix array data structure may include the *Bwttab* table, the *Burrows and Wheeler transformation* [20], which contains the character that precedes the suffix stored in the corresponding *Suftab* table cell.

Furthermore, they introduce the concept of lcp-intervals. This concept splits the lexicographically ordered *Suftab* into subintervals that share a common prefix of a certain length.

Definition 2.21. Let S be a string for which the enhanced suffix array should be constructed. An interval (i, j) with $0 \leq i < j \leq |S|$ is an *lcp-interval* with *lcp-value* l if

1. $Lcptab[i] < l$,
2. $Lcptab[k] \geq l \ \forall k \in \{i + 1, \ldots, j\}$,
3. $Lcptab[k] = l$ for at least one $k \in \{i + 1, \ldots, j\}$, and
4. $Lcptab[j + 1] < l$.

The idea behind lcp-intervals is that they represent the internal nodes of the suffix tree while the leaves cannot be represented using lcp-intervals, as they are singleton intervals of the form (i, i). The nesting of lcp-intervals can be represented using a virtual *lcp-interval tree*, which is

not stored in memory explicitly and where the interval $(0, |S|)$ is at the root. With the help of the lcp-interval tree, all problems that are usually solved by a bottom-up traversal of the suffix tree can be solved in the same time using the enhanced suffix array. An example of such a problem is the identification of maximal repeats and maximal repeated pairs, which will be discussed in Sections 3.3.2 and 4.6.

For solving problems that need top-down processing of the suffix tree, which will not be discussed in the thesis, such as pattern matching, an additional table *Childtab* needs to be computed. *Childtab*, the child table, stores the parent-child relationships of lcp-intervals, which correspond to the parent-child relationships of the internal nodes of the suffix tree. Using this table, all children of an lcp-interval as well as the branching brothers can be computed in constant time. Therefore, enhanced suffix arrays can be used instead of suffix trees for all problems that depend on top-down processing.

Last, the concept of suffix links can be incorporated into enhanced suffix arrays by creating *Linktab*, which stores the left and right boundaries of suffix link intervals. Using this table, the suffix link interval for any lcp-interval can be computed in constant time.

CHAPTER

3

EXACT MATCHING OF RNA SECONDARY STRUCTURES OF VIRAL GENOMES

Contents

In this chapter, we will analyze a data set of fully sequenced viral genomes from various species provided by the NCBI Viral Genomes Resource [18] in order to find examples of RNA secondary structures that are identical among different species. Viruses are an excellent fit when it comes to the discovery of shared RNA motifs without the involvement of conserved primary sequence regions because of their high mutation rates. Their interactions with host cells are essential for the infection and are often based on specific RNA motifs shared between different evolutionary distant viruses. Convergent evolution is a potential mechanism that explains similar motifs in taxonomically distant viruses for which similar selection criteria prevail.

To answer the question whether we can find potential examples of convergent evolution, the approach presented in this chapter will solely use the RNA secondary structure as representative for the folding of RNA molecules and does not depend on nucleotide sequence conservation.

The reason for this is that if we find identical secondary structures that are based on the same or almost similar primary sequences, then it is doubtful whether this is an example of convergent evolution or, more likely, a homologous relationship. On the other hand, if the examples show highly dissimilar nucleotide sequences, then the chance that this might be an example of convergent evolution increases. Alternatively, these dissimilar nucleotide sequences might be due to the extremely high mutation rates in viruses (for a review see [94]) that might cause mutations in homologous genes without interfering with the secondary structure of the resulting molecules. But just as well, the viral mutation rates might facilitate the rapid evolution of convergent structures from unrelated parts that provide an evolutionary advantage in specific hosts. In the end, only a detailed human examination of every putative example that can be found by a computational approach might reveal the answer to the question whether the similarity of the structures is due to convergent evolution or based on a homologous relationship. But this must necessarily include laboratory experiments to check whether the structured RNA molecules perform the same or similar functions within the host cell. So far, these questions cannot be answered by models and *in silico* simulation.

In Section 3.1, we start with a detailed look on the multiple types of convergence in evolution. In Section 3.2, we will present the extracted data from the NCBI Viral Genomes Resource and Section 3.3 will illustrate the methodology that is used to identify identical RNA secondary structures within several viruses from different species. Next in Section 3.4, we analyze whether the identified common RNA motifs show similarities to already known RNA families in *Rfam*. In Section 3.5, we will present a filtering process that can help to find the best candidates that might be the result of convergent evolution. Last, we summarize the results and show their connections to the following chapters.

3.1. Types of convergence

In phylogenetic analysis if two different species share a common ancestor, then they have been undergoing divergent evolution. During this process, independent traits are accumulated, which cause the divergence of two species. In contrast, convergent evolution is used to describe the phenomenon that two different species that are originated from two ancestors share related or similar traits. It is easy to understand on the morphological level that two species from two different ancestors could look similar to each other on the first appearance.

When it comes to the molecular level, Doolittle divided convergent evolution into three categories: functional convergence, mechanistic convergence, and structural convergence [28]. Functional convergence is what occurs when multiple molecules without sequence or structure similarity perform the same molecular function using different mechanisms. This can be observed in bacteria, plants, as well as in animals. A good example of functional convergence is the phenomenon that enzymes or ribozymes that evolved separately can catalyze the same

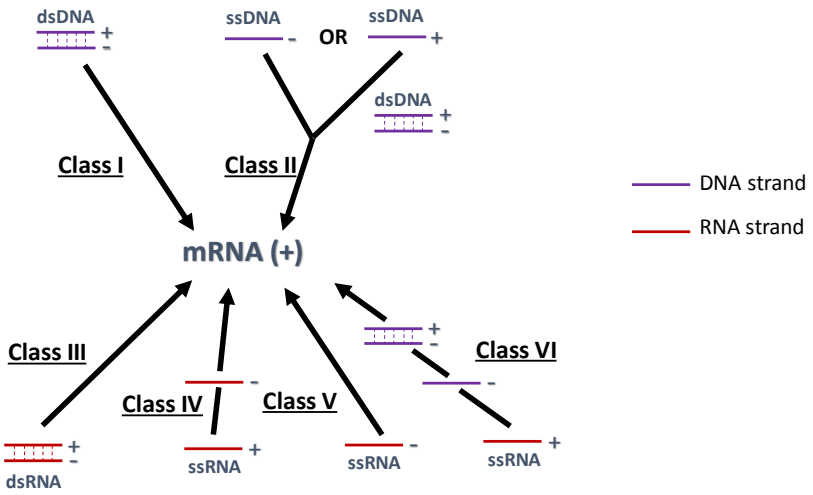


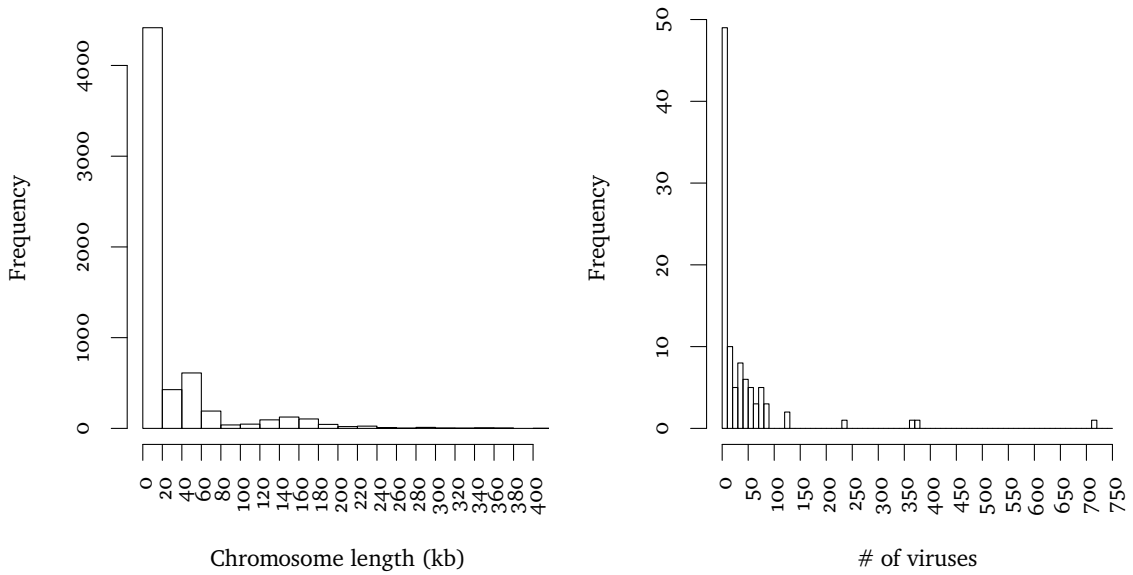
Figure 3.1.: Graphical representation of the Baltimore classes I to VI.

reaction. The molecules that show mechanistic convergence carry out the same function using the same molecular mechanism; however, they are different in terms of sequences and three-dimensional structures. Structural convergence refers to molecules that lack sequence similarities but contain identical structural motifs. These motifs can be found in different proteins and catalytic RNAs and might carry out the same molecular functions.

In this chapter, we will try to find new potential examples of structural convergence by using the secondary structure of an RNA molecule as an indicator. Whether this finding brings up molecules with the same functions can only be determined by laboratory experiments.

3.2. Preparation of the viral data set

The NCBI Viral Genomes Resource [18] is a catalog of all publicly available viral genome sequences with annotations, such as taxonomic classification, infected hosts, and neighbor information. The taxonomic information is based on the ICTV classification [63]. Viruses interacting with one or multiple host organisms are highly diverse in many ways, such as genome size and symptoms they cause. Despite the diversity, the process of synthesizing mRNA is common to all viruses. Therefore, the classification of viruses can be defined by the viral genetic material for replication as well as the formation of the mRNA. Based on this, Baltimore divided the viruses into six classes [9], which are illustrated in Figure 3.1. In this system, the mRNA strand that is used to encode proteins is labeled as plus strand, which requires a complementary minus strand as template. In the first two classes, double-stranded DNA (dsDNA) viruses (Class I) and single-stranded DNA (ssDNA) viruses (Class II) provide the templates for producing the mRNA. On the other hand, viruses from Classes III-VI use the RNA



(a) Length distribution of virus chromosomes **(b)** Distribution of viruses per family

Figure 3.2.: Database statistics for the reference sequences of all viral species and subspecies. In (a) pandoraviruses, megaviruses, mimiviruses, and pithoviruses were excluded for the sake of clarity.

strand as the complementary sequence to the viral mRNA. To be specific, Class III viruses carry double-stranded RNA (dsRNA), of which the minus one functions as template. Class IV viruses consist of the plus strand of the single-stranded RNA (ssRNA) as genetic material. They act as template for synthesizing the minus strand, which is followed by the production of the viral mRNA. Viruses from Class V contain the minus strand of the ssRNA to synthesize the mRNA directly. Class VI viruses, also called retroviruses, are special due to the fact that their RNA is reverse-transcribed to dsDNA, which is then used for forming the mRNA.

Usually, the first sequenced genome of a particular species is chosen as reference and the other sequenced genomes of the same species are labeled neighbors. But the model of only one reference sequence for a species does not capture all viral sequence variants, amongst others owing to the high mutation rates and horizontal gene transfer. Therefore, some species are either represented by multiple reference sequences, or are divided into subspecies with representative genome sequences that should capture most of the intraspecies variation. For this experiment, we haven chosen to retain all reference genomes for species as well as subspecies so that most of the variation is included. The neighbor sequences were excluded in order to reduce the size of the search space and to prevent multiple identical segments from the species to form the same secondary structure that would result in an exact match. Overall, our database contains sequences from viruses of all Baltimore classes, viroids, as well as satellites.

In total, the database contains 4,667 reference sequences with chromosome lengths ranging

from 3.5 kilobases up to 2.5 megabases for pandoraviruses, which infect amoebas [87] (see Figure 3.2a). But the majority of viruses has genomes with short lengths up to 20.5 kb. The usually small genome sizes enable us to perform the folding and exact matching with less computational effort compared to bacterial or eukaryotic genomes. Also, since we are interested in identical RNA structures with low primary sequence conservation, structural matches between viruses from different taxonomic families appear most promising in this regard. Figure 3.2b shows that most viral families include up to ten species while only six families have members from more than 100 species or subspecies. This distribution appears promising so that we can start with secondary structure folding of all genomes in the next step. Our local database can be used subsequently to access all genomes with identical RNA structures so that their host associations and taxonomic information can be compared.

3.3. Strategy for exact matching

The complete process of finding exact RNA secondary structure matches in multiple viral reference genomes is sketched out in Figure 3.3. The approach expects a local database of reference genomes of all viral species and subspecies based on the ICTV classification. The preparation of this database was discussed in the previous section. Before the search for exact structure matches in the sequences of our local database can begin, we need to take care of the RNA secondary structure folding of all of the reference genomes. On that account, all reference genomes will be folded using a window approach and the three secondary structures with different shapes and the lowest minimum free energies for each window will be collected an index data structure. A detailed introduction of the folding process can be found in Section 3.3.1. Subsequently, this index data structure can be used to identify maximal repeated pairs of nested RNA secondary structures. All maximal repeated pairs that represent the same maximal repeat will be grouped into a cluster. These clusters contain all occurrences of a specific nested RNA secondary structure in the viral reference genomes. Afterwards, the set of clusters can be filtered for low sequence similarity between all occurrences in a cluster. This is done to increase the chance of finding identical RNA structure occurrences that stem from convergent evolution. Additionally, we can divide clusters such that only occurrences in viruses that infect common hosts are grouped into one. Furthermore, occurrences that lie within an open reading frames (ORFs), or known protein-coding genes might be filtered out, because they probably do not correspond to ncRNAs that are of interest in our case. On the other hand, genomes of most viruses have overlapping genes and structural motifs [24]; therefore, removing all occurrences with overlapping annotations of protein-coding genes might exclude actual true positive results. In the best case, we will end up with a set of clusters with different structure lengths and properties that represent potential sites that were targeted by convergent evolution.

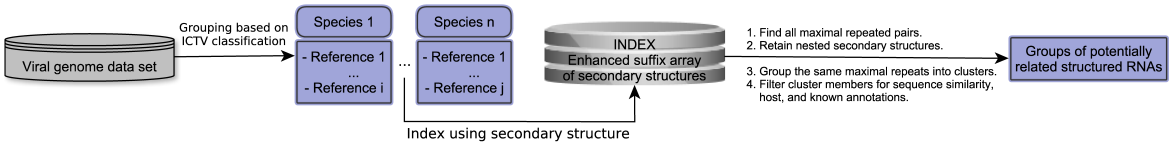


Figure 3.3.: Flowchart that shows the sequence of steps performed to identify exact RNA secondary structure matches in multiple viral reference genomes.

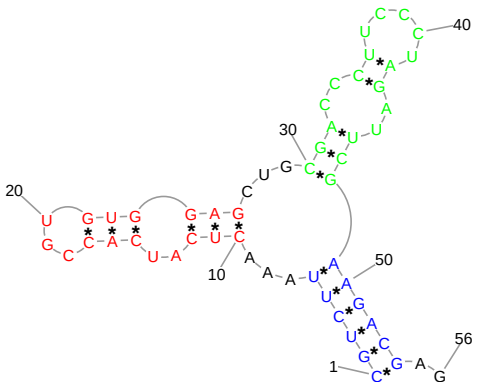
3.3.1. Folding genome sequences using windows

Before we can start the search for identical secondary structure regions, the possible secondary structures of the viral reference genomes have to be found and examined. In order to do this, we will fold all of those genomes separately using a sliding-window approach, in which a folding algorithm is run for a predefined number of sequence windows of size w. The number of sequence windows that will be subjected to folding depends on the step size s, which describes the distance in nucleotides between two adjacent windows. Before describing the way we use to decide which secondary structures will be added to the index data structure, the idea of abstract RNA shapes needs to be introduced.

RNA shape analysis A shape abstraction is a mapping from concrete RNA structures, e.g. represented as dot-bracket strings, to the class of abstract RNA shapes that were first defined in [38]. Often, the predicted minimum free energy structure is not necessarily the native structure of the molecule. Prominent examples are tRNAs, which can contain modified nucleotides that influence the structure such that the actual cloverleaf shape is not optimal under the energy model that was used for the *in silico* prediction. Instead of looking for significant differences in the whole suboptimal folding space manually, abstract shapes offer a way to partition the search space into groups of similar structures using multiple abstraction levels. In general, the abstract shapes approach is generic with respect to the used mapping function π, which has to preserve the nesting and adjacency of stems (tree homomorphism).

Definition 3.1. Let S be an RNA sequence, π a mapping function that preserves the nesting and adjacency of stems, and $F(S)$ the folding space of S. The *abstract shape space* of S is defined as $A(S) = \{\pi(x) \mid x \in F(S)\}$. The set of *P-shaped structures* in $F(S)$ is defined as $F(S \mid P) = \{x \mid x \in F(S), \pi(x) = P\}$.

The tool `RNAshapes` [56, 98] defines five different abstraction levels that disregard the size and the exact position of a stem. These abstraction levels are illustrated in Figure 3.4. Shape level 5 abstracts from all helix interruptions (bulge and internal loops) and single stranded regions. With shape level 4, helix interruptions by internal loops are also considered. Alternatively, shape level 3 records also helix interruptions on only one side, i.e. bulge loops. For further




```
Level 5: [[][]]

Level 4: [[][[]]]

Level 3: [[[]][[]]]

Level 2: [[_[]][_[]_]]

Level 1: [_[_[_]]_[_[_]_]]_
```

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

Figure 3.4.: Example of RNA shape abstraction showing all levels from 5 to 1. The three helices are visualized using different colors.

Algorithm 3.1 Generate secondary structures using a sliding window

```
1: function FoldStructuresWithSlidingWindow(I, w, s)
2:     for R ∈ I do                          ▷ all reference genomes in the input set
3:         i ← 0
4:         while i + w − 1 ≤ |R| do
5:             generate F(S^i_{i+w−1}) using RNAshapes
6:             add the three shreps from F(S^i_{i+w−1}) with the lowest free energy to the index
7:             i ← i + s
```

specification, shape level 2 accounts additionally for unpaired bases in all helix interruptions. Last, with shape level 1 all single stranded or helical regions are recorded explicitly.

Definition 3.2. Let S be an RNA sequence and $F(S \mid P)$ the set of P-shaped structures in $F(S)$. The *shape representative structure (shrep)* of shape P is defined as $\arg\min_{x \in F(S|P)} E(x)$.

A shrep is the structure whose free energy is minimal among all secondary structures that are mapped to the same abstract shape class.

Our approach uses the notion of shreps for identifying structures that should be inserted into the index data structure for exact matching. Algorithm 3.1 shows that all reference genomes are folded with the tool `RNAshapes` using a sliding window approach. By using `RNAshapes`, we do not have to inspect the whole suboptimal folding space and differentiate between many similar structures but can rather focus on the ones with *important* differences. This concept is illustrated in Figure 3.5, which shows a snippet of the suboptimal folding space of a tRNA sequence that consists of the twenty structures with the lowest free energies. The third column

```
     1  UCAUAUAGUGUAAUGGACAGCACAACGGUCUUCUAAGCCGUAAGGUCUAGGUUCGAAUCCUAGUAUGA  68
-14.40  (((((..((((........)))).(((((.......))))).....(((((.......)))))))))  [[][][]]
-13.50  .((((..((((........)))).(((((.......))))).....(((((.......))))))))).  [[][][]]
-13.40  .......((((........)))).(((((.......))))).....(((((.......)))))).....  [][][]
-13.40  (((((..((((........))))..(((.......))))......(((((.......)))))))))  [[][][]]
-12.90  ((((...((((........)))).(((((.......))))).....(((((.......))))).))))  [[][][]]
-12.50  .......((((........)))).(((((.......))))).....(((((.......))))).))..  [][][]
-12.50  .(((((..((((........))))..(((.......))))......(((((.......)))))))))  [[][][]]
-12.50  (((((..((((........))))............(((....)))(((((.......)))))))))  [[][][]]
-12.40  .......((((........))))..(((.......))))......(((((.......)))))).....  [][][]
-12.30  ...(((((((((........)))).(((((.......))))).....(((((.......)))))))))  [[][][]]
-12.00  (((....((((........)))).(((((.......))))).....(((((.......))))))..)))  [[][][]]
-12.00  .(((...((((........)))).(((((.......))))).....(((((.......))))).))).  [[][][]]
-11.60  .(((((..((((........))))............(((....)))(((((.......)))))))))).  [[][][]]
-11.50  .......((((........))))............(((....)))(((((.......)))))).....  [][][]
-11.50  .......((((........))))..(((.......))))....(((((.......))))).))..  [][][]
-11.30  ...(((((((((........)))).(((((.......))))).....(((((.......)))))))))  [[][][]]
-11.30  ..................((...(((((.......))))))...))(((((.......)))))).....  [][]
-11.20  ((((........))))...((...(((((.......))))))...))(((((.......)))))).....  [][][]
-11.10  .((....((((........)))).(((((.......))))).....(((((.......))))))..)).  [[][][]]
-11.10  ........(((.....)))((...(((((.......))))))...))(((((.......)))))).....  [][][]
```

Figure 3.5.: Part of the suboptimal folding space of a tRNA sequence that shows the structures with the lowest minimum free energies. The first column shows the free energy, the second column shows the structure in dot-bracket notation, and the last column shows the corresponding shape in level 5. The red highlighted lines are the shape representative structures that are inserted into the index data structure.

shows the abstract shape representation in level 5 of the dot-bracket string in the second column. It is apparent that most of the folds with a low free energy have the same abstract shape, the well-known cloverleaf shape, which implies a similar order and nesting of stems. But since we are interested in a greater variety of dot-bracket strings with significant differences, the tool `RNAshapes` helps us to reduce the number of structures in the suboptimal folding space by only outputting one shrep per shape class. In the example that is shown in Figure 3.5, these are the red highlighted structures.

We have chosen to use the shape level 5 for great structural variety and to insert the three shreps with the lowest free energies. Furthermore, the window size is 100, which is a trade-off between computational effort and ability to detect more complex RNA structures. By looking at the lengths of the secondary structures in *Rfam*, we can see that our chosen window size differs only slightly from the median 113. Additionally, we can highlight that even though the full structure of an RNA molecule of length greater than 100 nucleotides cannot be resolved fully, at least some substructures should be folded similarly. Last, the skip value s was set to 25 so that we have a total of four windows within 100 nucleotides. Overall, we ended up with a total of $10,655,100$ windows that have to be folded and $31,346,596$ structures that were inserted into the index. The latter number is slightly smaller than three times the total number of windows, because not every suboptimal folding space contained three different level 5 shapes.

3.3.2. Exact matching using maximal repeats and suffix arrays

After all secondary structures of interest from the viral reference genomes were collected, the next step is to identify identical segments of ≈ 31 million structures. Clearly, we cannot perform $\binom{31,346,596}{2}$ comparisons in a reasonable amount of time. Fortunately, standard string processing techniques provide a faster solution; we use enhanced suffix arrays as index data structure. To define common segments, we use the notion of *maximal repeats*.

Definition 3.3. Let S be a string. A pair of substring indices $((i, j), (k, l))$ is called a *repeated pair* if and only if $i \neq k$, $j \neq l$, and $S_j^i = S_l^k$. A repeated pair $((i, j), (k, l))$ is called *left maximal* if $S_{i-1}^{i-1} \neq S_{k-1}^{k-1}$ and *right maximal* if $S_{j+1}^{j+1} \neq S_{l+1}^{l+1}$. A repeated pair is called *maximal repeated pair* if it is both, left and right maximal. A substring R of S is called *maximal repeat* if there is a maximal repeated pair $((i, j), (k, l))$ such that $R = S_j^i$.

In order to identify maximal repeats of multiple input sequences, we interpret all secondary structures as one concatenated string where all structures are separated by a unique character that is not part of the alphabet. For the detection of all maximal repeated pairs of all secondary structures in the enhanced suffix array, we use the software suite `Vmatch` [66]. After collecting all secondary structures that should be part of the index, running the tool `mkvtree` of the `Vmatch` suite creates the enhanced suffix array. In order to compute maximal repeated pairs in a suffix tree, the tree has to be analyzed in bottom-up fashion. Therefore, we run `mkvtree` with the parameters for creating the tables *Suftab*, *Bwttab*, and *Lcptab*. Using these three tables, the computation of all maximal repeated pairs of input string S can be done in $\mathcal{O}(|S| + k)$ time, where k is the number of maximal repeated pairs. The procedure for identifying all maximal repeated pairs in a suffix tree is illustrated in Section 4.6. The main difference between the algorithm working on suffix trees and the one on enhanced suffix arrays is that the algorithm for the latter one uses the implicit lcp-interval tree to find the internal nodes that correspond to the maximal repeats.

Mapping our structure comparison problem to standard string processing techniques provides a computationally feasible solution for finding duplicates but comes with a drawback: we loose control over the fact that we are interested in duplicates of substrings that represent complete and well-formed structures, not in arbitrary substrings of an RNA structure.

Example 3.1. The following dot-bracket strings show examples of well-formed and not well-formed secondary structures.

well-formed:
```
(((((...)))).....((((...))))))
```

not well-formed:
```
))...(((((...)))).....((((...))))))
(((((...)))).....((((...)))))))...(((
((((((...)))).....((((...))))))))
```

The right side shows structural segments that do not constitute well-formed RNA secondary structures, but removing the red highlighted characters transforms the string into the longest substring that is a well-formed structure.

The example illustrates the next step after identifying all maximal repeated pairs. Since we are interested in the longest common structural motifs in multiple viral reference genomes that might be functional RNAs, we need to identify the longest substring of the maximal repeat that is a well-formed RNA secondary structure. For this task, we start with the output of `Vmatch`, which is a list of all maximal repeated pairs. Now, the structure of each maximal repeat gets analyzed in order to find the longest well-formed RNA structure that is a substring of the maximal repeat. By iterating over the structure of each maximal repeat, the runtime increases to $\mathcal{O}(|S|^2 + k)$ (see Section 4.6). In practice, the computation of all maximal repeats using `Vmatch` can be accomplished in short time. The bottleneck, on the other hand, is parsing the `Vmatch` output, which is a large flat file containing all maximal repeated pairs, and the subsequent analysis of the repeats to find the longest substrings that are well-formed RNA structures. In Chapter 4, we improve on the approach presented so far and will introduce a new data structure that enables us to compute all maximal repeated pairs that represent well-formed RNA structures directly from the tree.

Based on the processed output of `Vmatch`, all maximal repeated pairs that form the same maximal repeat are grouped together into a *cluster*. A cluster holds all occurrences of a maximal repeat in any viral reference genome. In total, we have identified 3,799,410 clusters with a minimal RNA structure length of 50. Figure 3.6 shows a graphical overview of the distribution of these clusters among different abstract shapes and the number of occurrences of each cluster in distinct viruses. The hierarchical clustering shows that the columns of shapes with similar structural complexity are grouped together. Simple shapes with one or two adjacent hairpin structures contain multiple clusters that are shared among different viruses as well as a large number of random matches. On the other hand, structurally more complex clusters with a large number of hairpin and multiloop structures have occurrences in only a few distinct viruses. All in all, one would expect a similar result using random genomes, as there does not seem to be a most favorable RNA shape among viruses. For the subsequent analysis steps, we discard all clusters that contain less than two hairpin structures and have occurrences in less than five different viral reference genomes. In this way, we can reduce the number of clusters that have to be analyzed further to 61,281 and lose only those clusters that represent a simple structure or have a low number of occurrences.

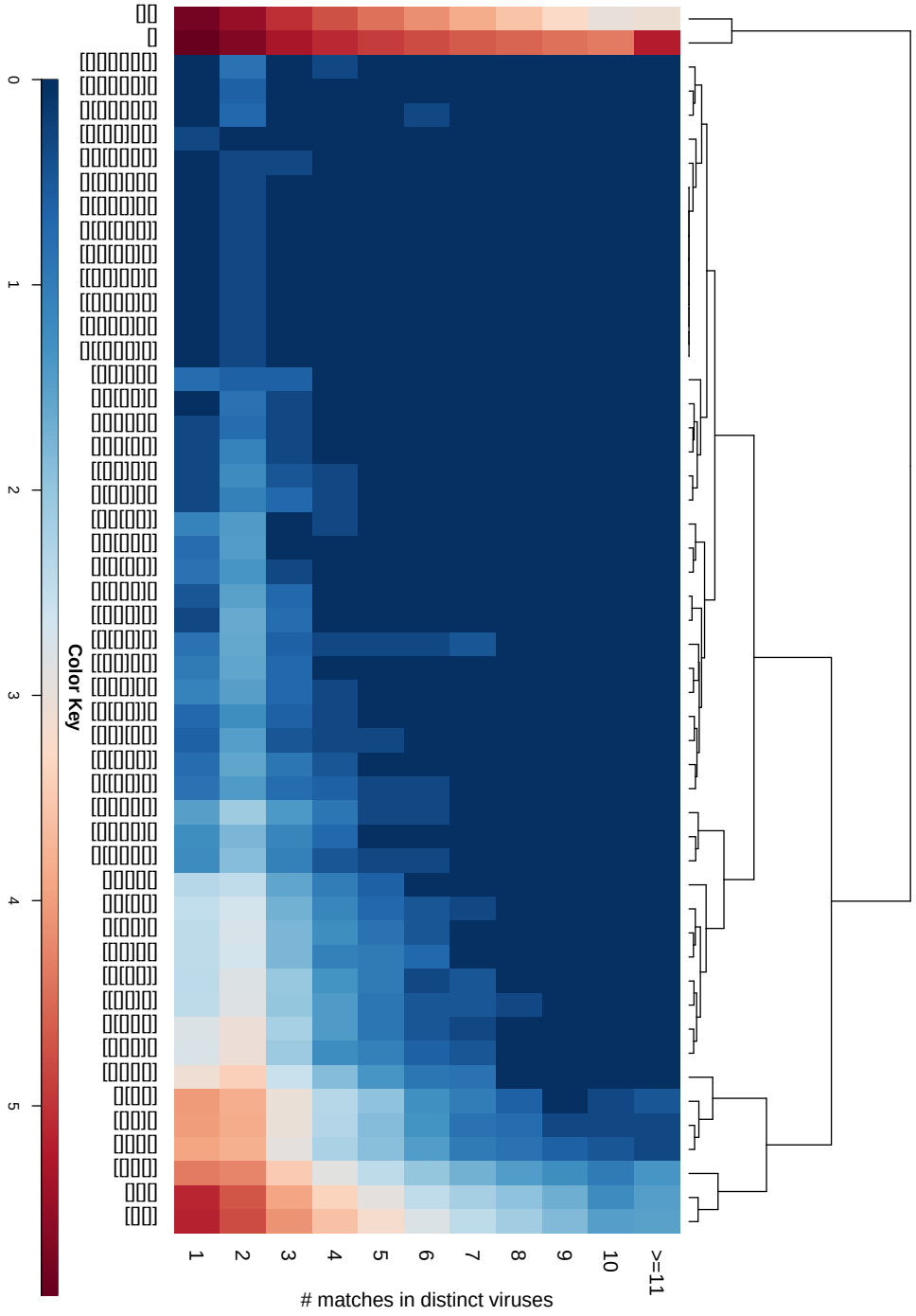


Figure 3.6.: Heatmap showing the segmentation of clusters into abstract shapes of level 5 on the x-axis and the number of occurrences in different viral reference genomes on the y-axis. The color code shows the number of clusters (log 10 scale) that can be grouped into the respective category. Furthermore, the columns have been subjected to hierarchical clustering using the Eucledian distance of the values in the column vectors.

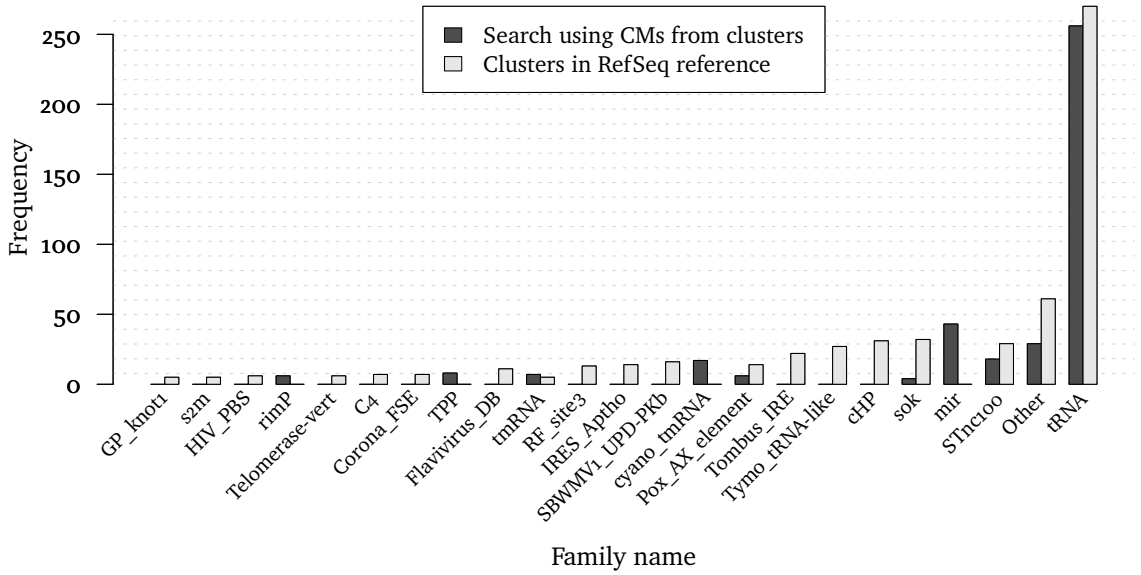


Figure 3.7.: Results of the two search strategies that match known ncRNAs in *Rfam* with the occurrences of the clusters.

3.4. Known families in clusters

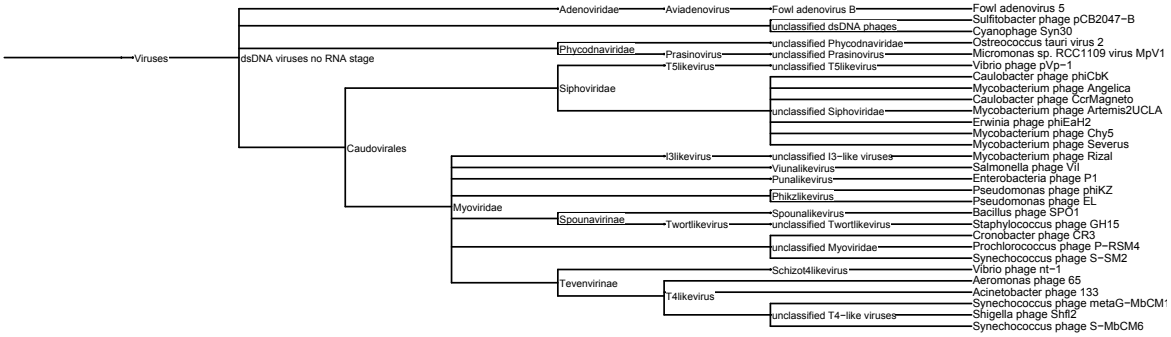
First, we wanted to find out whether any structures and the corresponding occurrences in the viral genomes can be assigned to any of the known RNA families of the *Rfam* database [79]. For this, we used the software suite `Infernal` [80] as reference implementation for CMs [30], which are probabilistic profiles of the sequence and secondary structure of an RNA family and can be used to quantify homology of an RNA sequence to the family. The training of the CMs updates the family independent prior probabilities with the posterior frequencies that can be observed from the structure annotated multiple sequence alignment of the RNA family. The result of the training are SCFGs (see [34] for an introduction) that are similar to hidden Markov models (HMMs) but more powerful in modeling coupled positions, which represent the base pairs of the annotated structure.

We have performed the search in both directions: first, we have searched the complete *Rfam* database for matches with any of the cluster occurrences. The *Rfam* database provides cross references to matches of RNA family models, which are built using manually selected seed sequences, to the ncRNA section of the NCBI Reference Sequence collection (RefSeq) [89]. Since all of our viral reference sequences are also part of RefSeq, we can just check whether our occurrences are part of the cross reference file and do not have to perform the time consuming search with `Infernal`. In the second direction, we use `cmbuild` of the `Infernal` suite to create a CM for each cluster. Usually, `Infernal` is used to search for further homologs of the known RNA families of the *Rfam* database, but in our case, we assume that each cluster forms

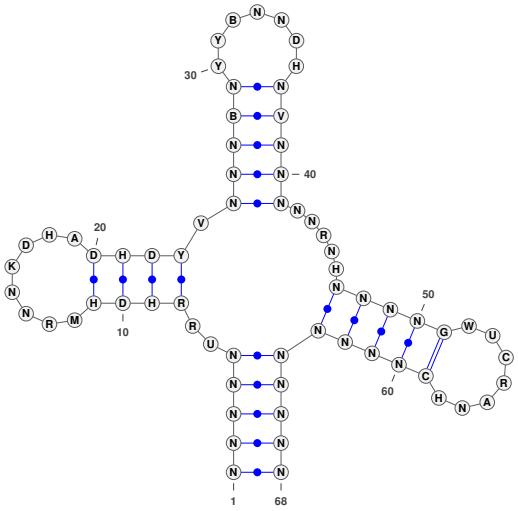
its own RNA family. This enables us to search for homologs in the full *Rfam* sequence database using our own CMs that are defined by the clusters. These newly generated CMs are further subjected to the previously mentioned training procedure so that the model can provide us with expectation values (E-values) for the bit scores. Subsequently, we have performed a local search with the tool `cmsearch` of the `Infernal` suite using an E-value cutoff 10^{-2}. A hit using this search suggests that the sequences in the respective cluster are related to the sequence of the *Rfam* database and can be classified as related to that RNA family.

The results of the searches are shown in Figure 3.7. First, when we compare the number of clusters that have been found using the two search strategies (394 and 581) with the total number of clusters (61,281), it is apparent that with both strategies less than 1% could be assigned to a known RNA family. Most of the clusters have been assigned to the tRNA family, the STnc100 family, whose ncRNAs can either bind to proteins or mRNAs that regulate gene expression, and the mir family of miRNA precursors. The tRNAs are found by both methods with a similar frequency while the other families with lower frequencies have diverging counts. The frequent occurrence of tRNA clusters can be explained by their strong secondary structure conservation even with low primary sequence identity (see Chapter 5 for a detailed analysis). The other diverging frequencies can be explained by the relatively small number of identified occurrences and by the different search strategies. The analysis of the search strategies might also explain why only 1% of the clusters could be assigned to an RNA family. First, the CM that is built for each cluster is based on a structure-annotated multiple sequence alignment. This alignment does not contain any gaps, because all sequences in a cluster fold into the same secondary structure. Often, the primary sequences of all cluster members are also very similar, which in the end results in a very restrictive CM that only matches sequences in the *Rfam* data set that also have similar primary sequence content. This might explain the low assignment rate of the second search strategy, but the first strategy that looks for matches in the *Rfam* cross references also could not assign much more clusters to a family. In general, new members of existing *Rfam* families are found using `Infernal`. In the first step of this process, all sequences of the manually curated seed alignment are aligned against a database of all targets using `BLAST`. All `BLAST` hits above a certain threshold are retained and are then searched using `Infernal` with all available family models. The first filtering step using `BLAST` exploits pure primary sequence similarity to prefilter potential hits and this might result in the loss of potential family members with different primary sequences and similar secondary structures.

Figure 3.8 shows one of the $\approx$ 250 tRNA clusters. The members of this cluster are mostly bacteriophages from different viral families that fold into the exact same secondary structure. The strongly conserved tRNA secondary structure for 29 viral genomes can be explained by the universally conserved translational machinery, where the interaction of host ribosome, tRNA, mRNA, amino acid, and polypeptide chain requires specific structural patterns to function properly. Interestingly, Figure 3.8b shows that the primary sequences of all cluster members are



(a) Taxonomy



(b) Secondary structure

Anticodon	Frequency	Amino acid
GUU	6	Gln
UCU	5	Arg
CCA	5	Gly
UGU	4	Thr
GAU	4	Leu
CAU	4	Val
GAA	2	Leu
UUU	1	Lys
UUG	1	Asn
UGC	1	Thr
UAA	1	Ile
GUC	1	Gln
CCU	1	Gly
CAC	1	Val

(c) Anticodons

Figure 3.8.: Example of a cluster that was assigned to the tRNA family. The sum of the frequencies in (c) is larger than the number of cluster members shown in (a), because some viruses have multiple genomic regions that are part of this cluster. (c) shows a graphic of the common secondary structure of all cluster members including IUPAC nucleotides generated using the union of all nucleotides at a specific position.

highly diverse, which results in multiple anticodons that are present in different cluster members (see Figure 3.8c). Each anticodon is specific for one amino acid, but the list of represented amino acids does not follow any observable pattern. We have observed similar characteristics for the other clusters assigned to the tRNA family. The large frequency of assigned tRNAs is in accordance with studies that have shown that they are the only translation-associated genes frequently found in bacteriophages. Furthermore, the studies found that frequently occurring tRNAs in phage genomes tend to correspond to codons that are highly used by the phage genes but are rare in host genes [8, 21]. Using the data gathered in Chapter 3, we can neither confirm nor refute this theory, because the host associations provided by the NCBI Viral Genomes

Resource only show the host domain and are not specific on the species level, where codon usage can be determined.

3.5. Filtering of the clusters

In the next step, we want to reduce the number of clusters containing sequences that might be potential examples of convergent evolution. Therefore, we use three different filtering methods. But before the filtering process starts, we first divide each cluster in multiple different subclusters, one for each host organism that is infected by at least two different viruses that are part of this cluster. The reason for this is that we expect convergent evolution to be a potential mechanism that explains similar motifs in taxonomically distant viruses that infect common hosts.

As first filtering step, we have used the `T-Coffee` [81] multiple sequence alignment tool, for the task of aligning all primary sequences of a cluster. The basic idea is to find clusters with low primary sequence similarities while knowing that the secondary structures of the these primary sequences exactly match. This could be an indicator for convergent evolution of function via structure, while on the other hand, similar primary sequences would suggest a homologous relationship. We ran `T-Coffee` with default parameters and the filtering was based on the alignment score, which shows the consistency with the previously computed pairwise alignments and is correlated with the overall accuracy but does not have the predictive power of an E-value. In general, the authors consider alignments below a score of 50 as "poor", which corresponds to low sequence similarity. Since we are interested in sequences with low primary sequence conservation and high secondary structure conservation, we only retain sequences with a score lower than 50[4].

Next, we used the tool `blastx` of the `BLAST` suite to look for sequences in each cluster that are associated with a protein sequence contained in the non-redundant database formed from `GenBank` [11] translations. Thereby, it compares the translation products of all six reading frames (both directions) of the query sequence against the protein database. If any of the translated queries finds a match in the database with an E-value lower than 10^{-2}, we remove the sequence from the cluster, because we are interested in ncRNAs and not potential structures of a transcribed mRNA.

In the last step, we analyze the taxonomic relationships of all sequences in a cluster. Using the database created in Section 3.2, we are able to link all sequences in a cluster to the corresponding species and its complete taxonomic lineage. For each cluster, we identify the lowest common ancestor (LCA) of all sequences and retain only those clusters that have a LCA above a certain taxonomic rank. In order to identify potential examples of convergent evolution, we have set the

[4]The authors of `T-Coffee` may not have anticipated this peculiar use of their tool, where we are interested in alignments of low quality.

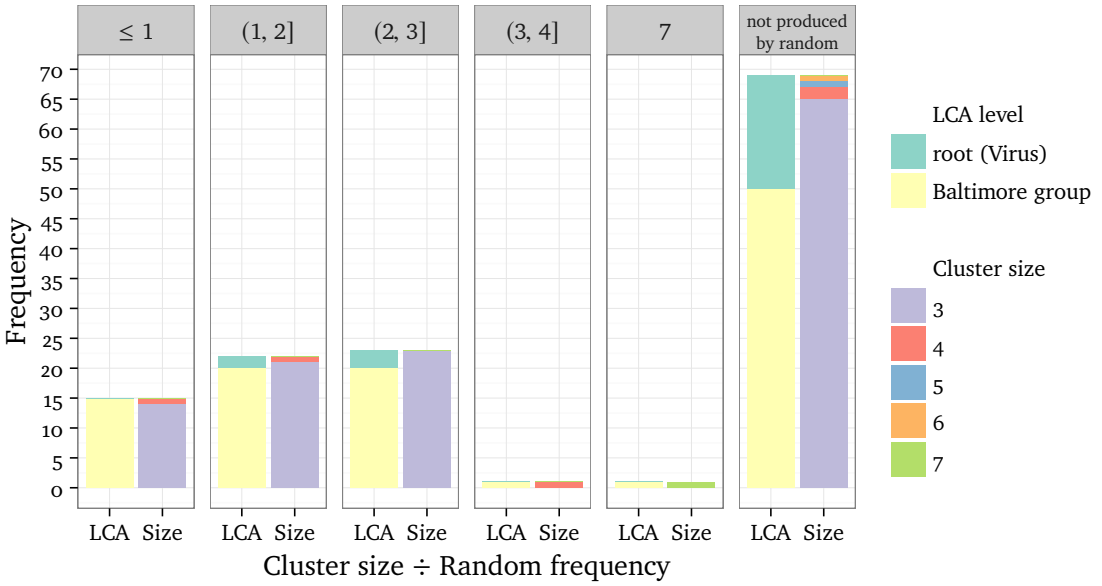


Figure 3.9.: Overview of the 131 remaining clusters after filtering. The x-axis is divided into six groups of clusters depending on the ratio $\frac{\text{Cluster size}}{\text{Random frequency}}$ that compares the cluster size with the number of occurrences of the respective secondary structure in the random data set. Each group is further divided into two bars showing the distribution of cluster sizes and LCA levels, respectively. Hence, the two bars of the same group have the same height and are representing the same number of clusters.

threshold for the taxonomic rank to a high level, the Baltimore classification groups, which are the second highest level, just below the root node. We presume that the secondary structures that are shared among species from different Baltimore groups are of great importance for the viruses, especially in the light of their short genomes and the high mutation rate.

The whole filtering process resulted in 131 remaining clusters and none of these clusters has been assigned to an RNA family in the previous experiment. All of these clusters represent secondary structures with lengths between 50 and 65 nucleotides that form two adjacent hairpin loops (level 5 shape `[][]`). In order to determine the significance of the identified clusters, we have repeated parts of the approach using random sequences. For this, we generated 31,346,596 random RNA sequences of length 100 assuming an equal distribution of the nucleotides. Following, `RNAshapes` was used again to fold the sequences and to choose the three best shreps from each folding space. These shreps have been used to identify all well-formed RNA substructures that have a length of at least 50 nucleotides. We have counted the occurrences of each substructure so that we can ask the question: how often can this RNA secondary structure be found in a random data set of equal size? If the structure occurs more often in the random data set than in the viral data set, we should classify this cluster as background noise rather than a potential example of convergent evolution. On the other

hand, if the count in the viral data set is significantly larger, we might have found an RNA structure that was either evolutionary preserved or evolved independently in multiple organisms for potentially the same function.

The result of this analysis is depicted in Figure 3.9. We have split the clusters into six groups based on the ratio of number of sequences in the cluster with the same secondary structure and the number of occurrences of this secondary structure in the random data set. There are 15 clusters that have a ratio ≤ 1 meaning that the specific structure of the clusters can be found more often in the random data set than in the viral data set. Therefore, we classify all matches of those clusters as background noise and will not consider them further. On the other hand, there is the group called "not produced by random" that contains 69 clusters that represent secondary structures that could not be found in the random data set. The majority of the secondary structures represented by the clusters in this group occur in three different viruses, but there are four clusters with occurrences in four to six different viral genomes. The LCA of most of the sequences in a cluster in the "not produced by random" group is on the Baltimore group level meaning that all sequences in a cluster are from only of the following groups: dsDNA, ssDNA, dsRNA, ssRNA, or retroviruses. But there also exist clusters that have occurrences in viruses from multiple groups meaning that a secondary structure is common to retroviruses as well as dsDNA viruses. This and the fact that those secondary structures cannot be found in the random data set are indicators that these structures might be novel regulatory RNAs associated with the infection of a common host organism.

3.6. Summary & additional steps

In this experiment, we have used 4,667 reference sequences from multiple viruses to find common RNA secondary structure motifs that are shared among multiple viral families and orders. The matching process for computing common motifs from RNA secondary structures was executed using an enhanced suffix array that performs regular string matching on the dot-bracket representation. The problem with this approach is that we have to perform a time consuming post-processing afterwards in order to shorten the exact string match to the longest well-formed RNA substructure. Considering this problem, we have worked on a novel data structure, called the viable suffix tree, which only retains suffixes that represent well-formed RNA secondary structures and returns only maximal repeats that are well-formed RNA secondary structures. This data structure is presented in Chapter 4.

After identifying 61,281 motifs that are shared among at least five different viral reference genomes, less than 1% of these motifs could be assigned to known RNA families. Most of the known motifs were assigned to the tRNA family, which can be explained by the extremely well preserved and known secondary structure. This also holds even with low primary sequence identity. This result puzzled and led us, among other things, to a critical examination of the

most widely used benchmark data set for RNA families. The detailed analysis can be found in Chapter 5.

Next, we filtered the clusters for low primary sequence conservation between all sequences that can fold into the same secondary structure in order to reduce the chance that the common RNA motif is based on homology between the DNA sequences and not a potential result of convergent evolution based on similar selection criteria for infecting the same or similar hosts. To make sure that all sequences within a cluster infect similar hosts, we have divided each cluster in multiple different subclusters, one for each host organism. Furthermore, we have removed all sequences from the clusters whose translation product can be found in the protein database. Last, we removed all clusters where the LCA of all sequences in the respective cluster is lower than a specified taxonomic level. In order to assess the significance of a secondary structure represented by a cluster, we have repeated the folding process using a random data set and counted how often a specific secondary structure can be found. Instead of using a random genome, an alternative would be to shuffle our viral data set while preserving the dinucleotide distribution of each genome, as it was shown that RNA gene prediction is influenced by the dinucleotide content [6]. In the end, we could identify 69 clusters of secondary structures that could not be found in the random data set, have low primary sequence conservation, and occur in at least three not closely related viral genomes. None of the clusters could be assigned to any known RNA family; therefore, we looked for further proximal primary sequence annotations that are shared among the occurrences of a cluster. But we could not identify any annotation that is close to at least two occurrences of a cluster in multiple viruses. Compared to other well-studied organisms, most of the viral genomes, with the exception of well-known ones, like the Hepatitis viruses or Tobacco mosaic virus, are considered "poorly described" [19]. Thus, some of these 69 clusters might be novel examples of sequences formed under selective pressure to build RNA structures that are necessary to infect a common host organism. Since the primary sequences of the clusters are highly dissimilar, we suspect that convergent evolution is a potential mechanism that might explain the similar secondary structures. But in the end, only laboratory experiments with the real RNA molecule can help answering the question whether the molecule is involved in the infection of the host organism and if so, what is its function.

The approach presented in this chapter considers only secondary structure conservation and uses primary sequence alignments to retain only those clusters whose sequences show low similarity. But often, known RNA families from the Rfam database have conserved sequences in the loop regions of the structure. So, it appears promising to search for structural similarities in paired regions and for sequence similarities in variable loop regions. In fact, in an additional experiment we have extended the exact matching approach by incorporating primary sequence information in the loop regions so that instead of a dot-bracket string `((...))`, a mixture of a dot-bracket string and a primary sequence `((CGA))` is inserted into the index data structure.

Subsequently, we have again searched for maximal repeats, now using the mixture of a dot-bracket and primary sequence string, and performed similar filtering steps with the exception of the multiple sequence alignment. In comparison with the previous experiment, we now have to optimize two objectives: dissimilar primary sequences in the paired regions and long identical stretches in the loop regions. The problem we are facing when we try to combine these two goals is that we are unable to find highly dissimilar stem sequences if we require identical sequences in the whole loop. Whenever we found a cluster where all loop sequences were identical, all other segments of the sequences were nearly identical, too.

The lesson from this extra experiment is that requiring exact matching, both on sequence and structure, makes the approach too restrictive. In Chapter 6, we present an approach based on short RNA sequence and structure seed motifs that can help to overcome the two main problems we identified here: the need for long exact structure matches without the possibility to allow single structural mismatches and the missing option to investigate primary sequence conservation in selected areas of the loop regions.

CHAPTER

4

THE VIABLE SUFFIX TREE

Contents

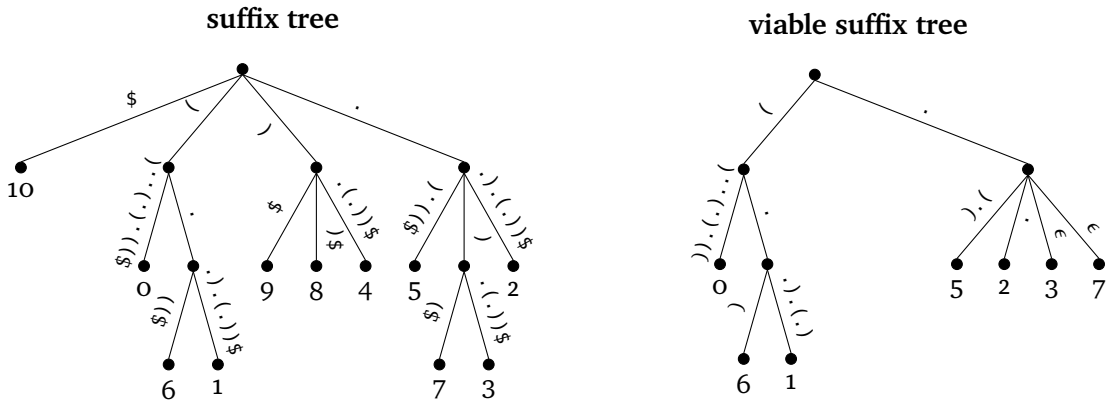
In the previous chapter, we have looked for segments of RNA secondary structures that are shared among different sequences of various viruses. We have employed the tool `Vmatch` in order to build the enhanced suffix array index data structure of all secondary structures of interest. Using this index, we have searched for maximal repeats of substrings of the RNA secondary structures that occur in multiple viral genomes. In this process, we noticed that a significant part of the maximal repeats that were reported by `Vmatch` were not well-formed RNA secondary structures but rather arbitrary dot-bracket strings. But since we are only interested in well-formed structures, these maximal repeats had to be post-processed in order to get the longest well-formed RNA structure that is a substring of the maximal repeat and is shared by all sequences in which this repeat was identified. Since all of the results of `Vmatch` are stored into flat files onto the hard drive first, the post-processing procedure has to read all the files from the disk, which results in a much slower processing time. Furthermore, the script that executes the post-processing has to be specifically adapted to the output format of the program that is being used for identifying maximal repeats.

This cumbersome and slow process led to the idea of an index data structure that takes the constraints on the RNA secondary structure into account and only outputs well-formed

structures. As we have previously used the suffix tree/array data structures, a modified and constraint suffix tree seems to be a good idea. For our case, this suffix tree should only include those suffixes that are well-formed RNA secondary structures themselves. If this does not hold, then we include the longest prefix of a suffix that is a well-formed RNA structure. On the other hand, if no such prefix can be found, then the suffix, or any prefix of it, will not be inserted into the tree.

Example 4.1. Assume that we want to build an index data structure of the following RNA structure, represented by a dot-bracket string that ends with the sentinel character $:

0	1	2	3	4	5	6	7	8	9	10
(	(	.	.	)	.	(	.	)	)	$



Compared to the classical suffix tree, our new data structure, the *viable suffix tree*, includes fewer internal nodes as well as leaves. Only those leaves are retained that do not start with a closing base character), because these strings will never represent a well-formed RNA structure independent of the additional characters in this suffix. The other leaves are shortened, at least the terminal symbol is removed, to the size of the longest prefix that represents a well-formed RNA structure. Informally, we can define the viable suffix tree as a pruned version of the suffix tree that holds shortened leaves, whose lengths are defined by the longest prefix that represents a well-formed RNA structure. If no such prefix exists, then the leaves will be removed. On the other hand, the longest prefixes of the suffixes starting at positions 3 and 7 are identical, which is why both are represented by the same string in the viable suffix tree that ends at an internal node. Both leaves are represented by an edge labeled with the empty word so that we can differentiate between the two positions. A more formal and complete definition can be found in Section 4.3.

During the design and development of such a data structure, specifically for our use case, we have noticed that there are demands as well as use cases for other problem domains, not necessarily related to computational biology.

Problem statement Given L and S, construct a *viable suffix tree* that holds, for every viable suffix, its longest prefix that is a word in L.

For $L = \Sigma^*$, all suffixes of any S are viable and words of L themselves, and the viable suffix tree becomes the classical suffix tree. Otherwise, only viable suffixes will be represented in the tree and will be truncated to the length of their longest prefix in L. We do not put any constraints on the language L in the definition, but for practical matters, L must have a decidable word problem and should be described by a suitable grammar from the Chomsky hierarchy. In terms of our RNA secondary structures that are represented by dot-bracket strings, this is the grammar G_S from Example 2.3.

In this chapter, we will describe existing approaches that can generate constrained versions of suffix trees and we will argue that none of the mentioned approaches is suitable for our RNA related problem. Subsequently, the general approach of generating viable suffix trees from classical suffix trees will be introduced in Section 4.2. Thereby, we will focus on explaining the approach in a simple manner ignoring constraints that make this approach rather complex. The full version including proofs of runtime and correctness can be found in Appendix A. The data structure, named viable suffix tree, will be defined in Section 4.3 and is followed by the description of an algorithm that can built the viable suffix tree for a nested RNA secondary structure and the language generated by G_S from scratch. In Section 4.5, we will describe a pre-processing procedure of a nested RNA secondary structure input string that can be generated by G_S, in order to ensure that the viable suffix tree can be generated in linear time. Section 4.6 discusses the way of finding maximal repeats that are well-formed RNA secondary structures. Last, we will show benchmarks of our implemented viable suffix tree in Section 4.7 and conclude this chapter in Section 4.8.

4.1. Previous work on constrained suffix trees

In addition to classical suffix trees that store all suffixes of the complete input, some applications that have structured input need more refined and flexible data structures. In 1996, Kärkkäinen and Ukkonen introduced the *sparse suffix tree* [60] that holds a subset of k suffixes of a string S with $k \leq |S|$. They showed that a k-evenly indexed string can be built in $\mathcal{O}(|S|)$ as sparse suffix tree.

Also in 1996, Andersson *et al.* presented a restricted version of the problem by introducing the *word suffix tree* [5]. Instead of using evenly indexed strings, they assumed that the k suffixes that should be preserved are the ones at the beginning of a new *word*[5]. This tree stores for a given input string only those suffixes that start at word boundaries.

Example 4.2. Let D be a set of strings from the alphabet $\{\mathtt{a}, \ldots, \mathtt{Z}\}$ that are concluded by the

[5]Here, the term word is used in its most prominent meaning as unit of natural languages.

separator character #. Let S = `This#is#a#string#` be a string from D^+ that will serve as input for the word suffix tree construction. The word suffix tree of S holds suffixes that either start at the beginning of the string or after the separator character. In the case of S, the word suffix tree holds the words `This`, `is`, `a`, and `string`.

In 2006, Inenaga and Takeda [55] introduced an algorithm that was able to build the word suffix tree in linear time. Uemura and Arimura [103] have generalized the concept to *variable length codes* in which the input string is built from words of a regular prefix code that can be recognized by a finite state automaton. Recently in 2013, Bille *et al.* [13] showed that general sparse suffix trees can be constructed in $\mathcal{O}(|S| \log^2 k)$ time and $\mathcal{O}(k)$ words space. The concept of sparse suffix trees is not suitable for solving our problem, since they only hold suffixes of full length and not their prefixes. The more general concept of Uemura and Arimura based on words of regular prefix codes is also not suitable for the RNA exact matching problem, because our used language is not regular.

Another related problem is the construction of *property suffix trees*, as introduced by Amir *et al.* [4]. The idea was inspired by pattern matching problems related to molecular biology and special areas of a genome that follow a specific structure, such as tandem repeats, short interspersed nuclear elements (SINEs), and long interspersed nuclear elements (LINEs). Here, pattern matching itself is not sufficient, the matched substring must also satisfy a given property.

Definition 4.1 (adapted from [4])**.** A *property* π of a string S is a set of intervals $\pi = \{(s_0, f_0), \ldots, (s_k, f_k)\}$ where for each $0 \leq i \leq k$ holds that $s_i, f_i \in \{0, \ldots, |S| - 1\}$ and $s_i \leq f_i$.

The idea of property matching is not just to find all occurrences of a query string Q in S, but also to validate whether the position at which the query was found is covered by an interval in π.

Definition 4.2 (adapted from [4])**.** For a property π of a string S the *end location* of $0 \leq i < |S|$ denoted by $end(i)$ is defined as the maximal f_k such that $(s_k, f_k) \in \pi$ and $s_k \leq i \leq f_k$. If no such interval exists, then $end(i) = -1$.

The property suffix tree includes a prefix of any suffix starting at position i until the end position $end(i)$, which is the string $S^i_{end(i)}$, if the end location is not -1. At first glance, it seems that property suffix trees can be used for indexing nested RNA secondary structures by adding all well-formed substructures of the input string to π.

Lemma 4.1. *Let S be a string and π the property of S. For any two positions i and j with $0 \leq i < j < |S|$ and $end(j) \neq -1$ holds $end(i) \leq end(j)$.*

Proof by contradiction. Assume that $end(i) > end(j)$. Then, there is an interval $(s_k, end(i)) \in \pi$ with $s_k \leq i \leq end(i)$. We have excluded the case $end(j) \neq -1$; therefore, $end(j) \geq j$. But since

we assume $end(i) > end(j)$, $s_k \leq i < j \leq end(j) < end(i)$. By contradiction $end(j) = end(i)$, because $end(j)$ will be set to the end of the interval. □

Lemma 4.1 illustrates that property suffix trees are not suitable for solving our general problem as well as the problem related to exact matching of RNA secondary structures, because once any suffix S^i has an end location with a great distance to i, all other suffixes S^j with $i < j \leq end(i)$ will also have an end location that is greater or equal to $end(i)$.

After intensive screening of the various published manuscripts that discuss the topic of restricted suffix trees, we did not find a data structure or concept that would be sufficient to solve the problem mentioned in this thesis.

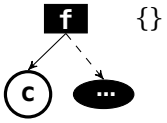
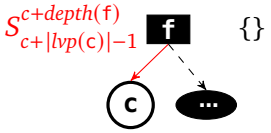
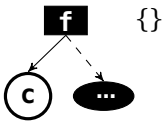
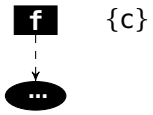
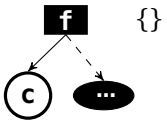
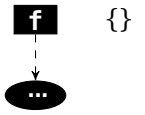
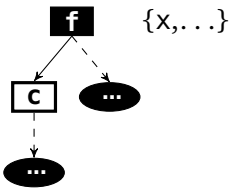
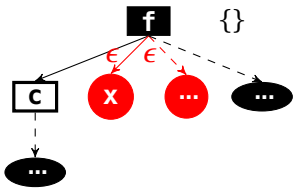
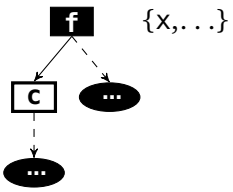
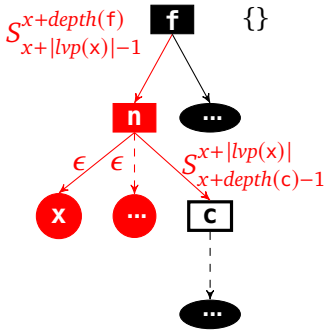
4.2. Pruning the suffix tree

In this section, we will present a general pruning approach that uses the suffix tree of a string S and generates the viable suffix tree based on those suffixes of S that have a prefix that is a word in the language L. In order to do so, we introduce the function $lvp(i)$ with $0 \leq i < |S|$ that, for given S, returns for every suffix S^i the longest viable prefix (lvp) of this suffix. The lvp of a suffix S^i is the longest prefix of S^i that is a word in L. The *lvp* function is related to the *end* function described in Definition 4.2, but instead of returning the end location of a lvp, it returns the lvp string. A formal definition of these functions for viable suffix trees follows in Section 4.3. For the purpose of giving a simplistic introduction to viable suffix trees, we will show a shortened version of the whole process that will not include the compaction of the tree. The version in this section only discusses the operations that are necessary to relocate shortened suffixes, i.e. prefixes of these suffixes, to the correct internal node. The complete pruning algorithm including the proof of correctness and the runtime analysis can be found in Appendix A. Note that this algorithm will work on any input string S and language L, so no restrictions are imposed here, other than the availability of the *lvp* function.

In order to create the viable suffix tree, the algorithm processes the existing suffix tree using a depth-first search. Therefore, it starts at the root node and recursively visits every internal node and leaf once. The processing of all nodes in the tree is done in post-order, which ensures that all children of a node are processed before the node itself. This becomes important during the pruning of the tree. Furthermore, note that the algorithm directly alters the existing suffix tree; therefore, no copy operation of the tree is necessary so that only one tree has to be stored in memory at any time. The pruning process is illustrated in Table 4.1 and can be divided in five distinct cases (named I to V). These cases differentiate between leaf and internal nodes and since the algorithm uses post-order, we start by discussing the cases of leaf nodes first.

In case I, the currently processed node is a leaf node and the length of the lvp is greater or equal to the depth of the parent node. This means that the current node does not have to be

Table 4.1.: The table lists a shortened number of cases that can occur while pruning the nodes of the suffix tree. Note that the tree graphics only show a subtree, i.e. `f` is not necessarily the root node. Internal nodes (including the root) are represented by rectangles; leaves are drawn as circles; ellipses represent nodes that can be either one. The current node is shown in white with the label `c`; the parent node has the label `f`. Newly inserted internal nodes are labeled `n` and leaves are called `x`. Leaf characters that are used within formulas represent the starting position of the suffix in the input string. All newly inserted nodes are shown in red. Also shortened or newly inserted edges are drawn in red. Furthermore, all changed edges are labeled with the new string that they are representing. The set of leaves that have to be reinserted are drawn next to the parent node. This set is cleared as soon as these leaves are inserted into the tree. The label `...` in either a node or the set represents 0 or more nodes while the label `x` represents exactly one leaf in the tree or in the set. The edges leading to these putative nodes are drawn dashed.

Case	Before	After
I c is leaf $lvp(c) \neq \epsilon$ $\lvert lvp(c)\rvert \geq depth(f)$		
II c is leaf $lvp(c) \neq \epsilon$ $\lvert lvp(c)\rvert < depth(f)$		
III c is leaf $lvp(c) = \epsilon$		
IV c is internal node $\lvert lvp(x)\rvert = depth(f)$		
V c is internal node $\lvert lvp(x)\rvert > depth(f)$		

relocated, because it has the label of the parent node as its prefix. But case I actually covers two distinct scenarios: first, if the suffix represented by the current node is a word of the language L, then the edge from the parent to the current node stays untouched. Second, if the suffix represented by the current node is not viable, then there exists a prefix that is longer or has equal length to the label of the parent node. Whenever this holds, the edge leading to the current node has to be shortened to the length difference between the lvp and the depth of the parent node. But this might lead to a new scenario that arises with viable suffix trees, namely the shortened edge can have length zero. This is the case if the length of the lvp is equal to the depth of the parent node. One might ask: why does the algorithm not always replace the parent node by the current node? The reason is simple: since multiple suffixes might share the same lvp, they must also share the same parent node. If this holds, the algorithm creates multiple edges of length zero under the same parent node. On the other hand, if this is not the case, then we should replace the parent node by the current node, but this compaction of the tree is not covered here.

Next, case II shows the scenario that the length of the lvp is smaller than the depth of the parent node. Here, the current node has to be relocated, because the label of the parent node is not a prefix of the lvp of the current node's suffix anymore. As result, the algorithm removes the leaf and the edge from the parent node. But we have to store the current node so that it can be reinserted into the tree later. The reasoning behind this is that the length of the suffix is shortened to the length of its lvp, and the correct position of the current node is below another internal node that is closer to the root. Therefore, the algorithm introduces a new set that collects all leaf nodes that have to be reinserted into the tree later. This set is passed from the leaves to their parent nodes, and from these internal nodes, it is also be passed to their parents until the right location for reinsertion has been found. This procedure works, because the leaves and internal nodes are processed in post-order so that nodes that are closer to the root are handled last.

Case III, on the other hand, can be dealt with quickly. The current node is a leaf node, but the suffix that this leaf represents does not contain any prefix that can be found as a word in language L. Therefore, the algorithm can remove the current node and the edge from the parent node without reinserting them later. This concludes the actions that have to be applied to leaf nodes.

Now, in cases IV and V, the algorithm analyzes the topology of the subtree of the current node including its parent node. The algorithm has to alter this subtree if and only if the current node received sets of leaves, which have to be relocated, from its children nodes and the length of the lvp of the suffixes represented by the nodes is at least as long as the depth of the parent node. Obviously, this means if this set is empty, we do not have to alter the tree, because there are no leaves that have to be relocated. Furthermore, if the length of the lvp of the suffixes that are represented by these former leaf nodes is shorter than the depth of the parent node,

then the insertion of these nodes has to be handled even further up in the tree, because at this point the algorithm processes only leaves that have to be inserted in between the parent and the current node. Here, we should emphasize that all leaf nodes in this set are inserted at the same location, so it is sufficient to only check the lvp of one of those suffixes represented by the leaf nodes. This holds, because all suffixes in this set share a common prefix that is represented by the current node.

Case IV handles the situation where the length of the lvp of the suffixes in this set is equal to the depth of the parent node. Now, the algorithm can simply insert all the nodes stored in the set by adding edges of length zero to the parent node, because they share the same depth as the parent node. The subtree starting at the current node is not altered here, even though this internal node could have no children, but in this simple description this situation is neglected.

In case V, the leaves in the set have to be reinserted somewhere in between the parent and the current node. Therefore, we introduce a new internal node, named n in Table 4.1, that is connected to the parent node with an edge that covers the interval between the depth of the parent node and the length of the lvp of all suffixes in the set. All leaves in the set are reinserted at this point, using edges of length zero that are connected to the new node. Furthermore, the current node now has an edge that originates in the new node instead of the previous parent node, because it shares a prefix with all reinserted suffixes from the set that is longer than the prefix it shares with the parent node. Obviously, the length of the new edge is shortened accordingly.

These fives case represent all necessary operations on the suffix tree that have to be performed in order to relocate all leaves to their new positions. But, as emphasized before, using these simplified operations, the algorithm cannot ensure that the resulting viable suffix tree will be compact. If the compactness of the tree is of importance, then the algorithm gets more complicated and the number of cases increases threefold, as shown in Appendix A.

4.3. Definition

In the previous sections, we started with a descriptive introduction, but now we arrived at the point where the viable suffix tree will be defined. We will start with the term lvp.

Definition 4.3. Let S be a string and L be a language. Any prefix S^i_j of suffix S^i with $0 \leq i \leq j < |S|$ that is a word in L will be called *viable prefix* of S^i. The *longest viable prefix (lvp)* of S^i, written as $lvp(S, i)$[6], is the viable prefix that has the longest length. If a suffix does not have a viable prefix, then the lvp is defined as ϵ[7]. The *end location* is the position of the last symbol of the lvp of S^i in the string S. More formal, it is defined as $end(S, i) = \arg\max_{i \leq j < |S| \wedge S^i_j \in L} \left|S^i_j\right|$ if a viable prefix exists. Otherwise, $end(S, i) = i - 1$.

[6] If the input string is clear from the context, then we will also use $lvp(i)$, as in Table 4.1.
[7] In this case, the suffix is not viable.

In contrast to the *lvp* function, the *end* function returns the end location of the longest viable prefix. Therefore, we might also describe the lvp using the *end* function as $lvp(S^i) = S^i_{end(i)}$.

Definition 4.4. A *lvp-repertoire* of string S is a multiset that contains $lvp(S, i)$ for every suffix S^i with $0 \leq i \leq j < |S|$ if and only if $lvp(S, i) \neq \epsilon$. Since multisets are a generalization of sets, we will use the characteristic function's symbol $\mathbb{1}_M(t)$ to denote the *number of occurrences* of string t in M. Further, the *cardinality* of M is denoted by $|M|$ and defined as $\sum_{t \in M} \mathbb{1}_M(t)$.

The lvp-repertoire contains the lvp of every suffix that has a viable prefix. Since multiple suffixes might have the same lvp, the number of occurrence of every set member has to be stored.

Example 4.3. Given are string S and the language generated by the grammar G_S from Example 2.3. The following example illustrates the *end* function as well as the lvp-repertoire of S.

i		0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	=	.	(	.	(	.	(	.	)	)	.	)	(	.	)
$end(S, i)$	=	13	13	9	9	7	7	7	6	7	9	9	13	12	12

Lvp-repertoire	Frequency
.(.(.(.)).)(.)	1
(.(.(.)).)(.)	1
.(.(.)).	1
(.(.)).	1
.(.)	1
(.)	2
.	3

Based on this preparatory work, we are now able to define our new data structure, the viable suffix tree. This definition is based on the one of the original suffix tree from Definition 2.20.

Definition 4.5. A *viable suffix tree* of string S with lvp-repertoire M is a rooted, directed, edge-labeled tree with the following properties:

1. The tree has exactly $|M|$ leaves that are labeled by values from the set $\{0, \ldots, |S| - 1\}$; no label can be used more than once.
2. Each internal node, except the root, has at least two children.
3. Each edge is either labeled with a non-empty substring of S or the empty symbol ϵ; the latter have to end in a leaf node.
4. No two edges with non-empty labels out of the same node begin with the same character.

5. For leaf node $\mathtt{l}$, $path(\mathtt{l})$ is equal to $lvp(S, l)$.

In comparison to the definition of the suffix tree, multiple items were altered and one is added. Item 1 now includes only those suffixes that have a lvp. In addition to that, Item 5 is now also responsible for keeping the labeling of the shortened leaves accurate such that the path leading to a leaf is equal to the lvp. Items 3 and 4 allow for edges with length zero (labeled by ϵ). Edges with length zero always start from internal nodes and lead to leaf nodes. In general, an internal node $\mathtt{i}$ has exactly $\mathbb{1}_M(label(\mathtt{i}))$ outgoing edges labeled with ϵ.

4.4. Direct construction of viable suffix trees

The following imperative algorithm presents an efficient way of constructing a viable suffix tree and is based on Ukkonen's algorithm [104] for online and linear time suffix tree construction. Differences and extensions in the algorithm compared to Ukkonen's description in his article are marked in red.

For constructing the viable suffix tree of string S with length n, we represent it as a pair $T = (V, E)$, where V denotes the set of nodes and E the set of edges. An edge is represented as a quadruple $(\mathtt{o}, \mathtt{d}, i, j)$, where the first component $\mathtt{o}$ denotes the origin of the edge, the second component $\mathtt{d}$ denotes the destination of the edge, and the two last components define the edge label S_j^i.

In general, the algorithm is divided into $|S|$ phases while the string is processed symbol by symbol from left to right. In each phase $i+1$, the (viable) suffix tree T^i from the previous phase is used to generate the tree of the current phase T^{i+1}. For that, the phase $i+1$ can be further divided into $i+1$ extension operations. Extension operation j starts by identifying the path with the label S_i^j. Then, we can simply use these three rules according to Gusfield [43]:

1. We find a leaf $\mathtt{v}$ with $label(\mathtt{v}) = S_i^j$ in T^i. Then, the edge leading to the leaf can be extended by character S_{i+1}^{i+1}.

2. The path with the label S_i^j does not continue with the character at string position $i+1$. Instead, it continues with one or multiple different characters. Then, we introduce a new leaf as well as a new edge leading from the end of the path S_i^j to the new leaf. The new edge is labeled with S_{i+1}^{i+1}.

3. There is already a path S_{i+1}^j that includes S_{i+1}^{i+1}. In this case, the suffix is implicit, but it will be explicit, at the latest with the processing of the sentinel.

The first rule does not have to be performed with Ukkonen's algorithm, because we know that once a leaf is created, it will always be updated with rule one in any successive phase. Therefore, whenever a leaf node is created, we simply set the end of the edge that leads to the leaf to the

last index of the input string. Thus, no leaf has ever to be extended again. Next, whenever rule three applies, it will also apply in all following extension steps of the same phase. Therefore, we can stop the current phase at this point, because all following suffixes are already represented implicitly. This describes the general strategy of Ukkonen's algorithm, but there are several tricks to make it work in linear time for alphabets of constant size.

Additionally, we introduce the notion of *suffix links*.

Definition 4.6. Let $S = cR$ be a string that is built from the concatenation of character c and string R. Furthermore, let v be a node such that $label(\mathsf{v}) = S$. If there exists another node u such that $label(\mathsf{u}) = R$, then the pointer from v to u is called *suffix link*.

It is important to note that if v is a branching node, then the endpoint of its suffix link is a branching node, too. (Gusfield [43] presents an excellent summary of the suffix tree construction according to Ukkonen, including the proof of the previous note.) However, this does not necessarily hold for viable suffix trees. The viable suffix tree in Example 4.1 shows that there is an internal node representing string "`(.)`", but "`.)`" does not exist, because it is not well-formed. Therefore, for viable suffix trees, we use the lvp for the suffix link if the actual one does not exist. In this case, the lvp of "`.)`" is "`.`"; therefore, the suffix link of "`(.)`" would point to the internal node representing "`.`".

In order to define the current state of the suffix tree construction, Ukkonen uses the notion of *reference pairs*.

Definition 4.7. Let T be a rooted, directed, edge-labeled tree and S a string. The pair (v, R) is called *reference pair* of S with respect to T if v is a branching node from T and $S = label(\mathsf{v}) +\!\!+ R$. If $label(\mathsf{v})$ is the longest prefix of S represented by a branching node of T, then the pair is called *canonical reference pair*.

If a node v directly represents string S, then the canonical reference pair of S is (v, ϵ). Furthermore, Ukkonen introduces the *active point*, which is the point at which the traversal of any extension process starts. It is used so that the traversal for every extension does not have to restart at the root of the tree. Usually, the previous extension makes sure that the active point is set properly for the next extension. This can be achieved by using the *end point* of the previous extension phase. The end point of a phase is the pair at which the processing for this phase stops. Often, this happens when rule three is applied.

The construction of a new (valid) suffix tree starts with a tree that only consists of the root node. The table *Link* stores all suffix links of the tree. Algorithm 4.1 processes the string from left to right and calls the UPDATE function at the start of every new phase. For the first step, the reference pair is set to $(\texttt{root}, S_0^0)$. In general, the active point of phase i is described here as (v, S_i^k). The call of function UPDATE returns the end point of phase i and function CANONIZE returns the active point, as canonical reference pair, for the next phase.

Algorithm 4.1 Direct construction of the viable suffix tree

```
1:  function CONSTRUCT VIABLE SUFFIX TREE(S)
2:      add node root
3:      Link[root] ← NULL
4:      v ← root
5:      k ← 0
6:      i ← 0
7:      while i < |S| do
8:          (v, k) ← UPDATE(v, k, i)                ▷ (v, S_i^k) is the active point of phase i
9:          (v, k) ← CANONIZE(v, k, i + 1, end(S, k − depth(v)))
10:         i ← i + 1
```

Algorithm 4.2 Update the tree for phase i

```
1:  function UPDATE(v, k, i)
2:      oldR ← root
3:      i′ ← end(S, k − depth(v))                   ▷ end position of the lvp of suffix S^{k−depth(v)}
4:      (endPoint, r, edgeLen) = TESTANDSPLIT(v, k, i, i′, S_i^i)
5:      while not endPoint do
6:          pos ← i − depth(r)
7:          if edgeLen > 0 then                     ▷ check whether the new edge has empty label or not
8:              add leaf with label pos and edge (r, pos, i, end(S, pos))
9:          else if edgeLen = 0 and r ≠ root then
10:             add leaf with label pos and edge (r, pos, −1, −1) with empty label
11:         if oldR ∉ {root, NULL} then
12:             Link[oldR] ← r
13:         oldR ← r
14:         if v ∉ {root, NULL} then
15:             k ← k − (depth(v) − depth(Link[v]) − 1)   ▷ suffix link distance may be > 1
16:         i′ ← end(S, k − depth(Link[v]))          ▷ end position of the lvp of suffix S^{k−depth(Link[v])}
17:         (v, k) ← CANONIZE(Link[v], k, i, i′)
18:         i′ ← end(S, k − depth(v))                ▷ end position of the lvp of suffix S^{k−depth(v)}
19:         (endPoint, r, edgeLen) = TESTANDSPLIT(v, k, i, i′, S_i^i)
20:     if oldR ∉ {root, NULL} then
21:         Link[oldR] ← r
22:     return (v, k)
```

Algorithm 4.2 transforms T^{i-1} into T^i. For every suffix starting at position $k - depth(\mathsf{v})$, position i' denotes the end position of the lvp. This new variable is used to call the function TESTANDSPLIT, which returns three values. The first one is a boolean value that tells us whether the current pair (v, S_i^k) is the end point of the current phase i. If not, then the new variable *edgeLen* tells us whether the new edge has an empty label or not. We add the new leaf and edge according to the value of the variable. Obviously, we do not add edges with empty labels to the root, because then we would represent empty suffixes, i.e. suffixes without an lvp.

Next, the algorithm follows the suffix link of v to the next shorter suffix. For the classical suffix tree, it is guaranteed that v has a suffix link to a node u that has a label that is only one character shorter than the one of v. As explained before, this does not hold for viable suffix trees. Instead we follow the suffix link, but will decrease the value of k such that it reflects the number of characters by which the suffix link of v is shorter than v itself. This is necessary, because otherwise the reference pair for the next extension phase would be wrong.

The question is now: how does this influence the runtime of the viable suffix tree construction? First of all, whenever the length difference between v and its suffix link is greater than one, it means that there are closing base pair characters that do not have a corresponding opening base pair character in the suffix that we are currently creating. This means also that the insertion of the suffixes starting with closing base pair characters will be skipped. Now, if we have node "((..))", its classical suffix link would point to a node "(..))", which is not valid. Instead now, it points to node "(..)". Furthermore, the new suffix link of "(..)" now points to ".." instead of "..)". Compared to the construction of the classical suffix tree, we have decreased the value of k already twice as often as usual. But, on the other hand, in the next steps, we can skip the insertion of the two suffixes starting with the closing base pair characters. Also, during these steps, the algorithm will not walk down the tree any further. Instead, function CANONIZE will make sure that we walk up the necessary edges before inserting the next suffix after processing the two closing base pair characters. This means that the total number of down and up walking for constructing the viable suffix tree is still bounded by $\mathcal{O}(n)$. This holds for the language of the grammar that generates well-formed RNA structures, because we have exactly one character, the closing base pair character, that might cause the suffix to get invalid at some position. Additionally, it is the character that causes the suffix to be invalid right from the start. All other suffixes that start with (or . are guaranteed to have a lvp of at least length one, because the input string of the tree construction is well-formed. For more complex grammars, this is not guaranteed to work.

The additional check whether the variable for the node `oldR` equals *NULL* is just an artifact of our implementation of the construction algorithm. In the original approach by Ukkonen, the parent of the root node is handled implicitly by having an edge of length one to the root via any character from the alphabet. We have chosen to handle this case explicitly; therefore, the check against *NULL*, which is the artificial value for the parent node of the root, needs to be

Algorithm 4.3 Check whether the tree for this step is complete

```
1:  function TESTANDSPLIT(v, k, p, p', t)
2:      if p = k then
3:          if v = NULL then
4:              return (True, v, _)
5:          else if there is an edge (v, _, _, _) starting with t then
6:              return (True, v, _)
7:          else
8:              return (False, v, p' + 1 − k)
9:      else if p' < k then                          ▷ invalid suffix; empty leaf possible
10:         return (False, v, p' + 1 − k)
11:     else
12:         minP ← min(p, p')
13:         search for an edge (v, v', i, j) starting with S_k^k
14:         if t = S_{i+p−k}^{i+p−k} then
15:             return (True, v, _)
16:         if j − i > minP − k then                  ▷ regular case for new branching node
17:             split edge (v, v', i, j) at position i + minP − k using branching node mid
18:             return (False, mid, end(S, minP − depth(mid)) + 1 − minP)
19:         return (False, v', end(S, minP − depth(v')) + 1 − minP)
```

considered here.

Algorithm 4.3 tests whether the current point is the end point of phase p. Function TESTANDSPLIT returns true if it is the end point. The answer can be found by only testing whether there is path that continues with character t. If $p = k$, then we have to check whether there is an outgoing edge from v that starts with t. On the other hand, if $k < p$, then the algorithm has to check whether the edge that describes the continuing path also contains character t at the right position. If one of the two cases is true, then rule three applies and the current phase can be ended.

The new condition $p' < k$ checks whether the suffix is invalid. In this case, the leaf that has to be inserted at node v has to have an empty label, because $p' + 1 = k$. If neither p nor p' is smaller than k, we have to correctly calculate the end point by taking the minimum from both. Further, node mid is introduced to cover two different cases: first, the already known case from Ukkonen's algorithm $j - i > minP - k$, which requires us to split the edge at position $i + minP - k$ and introduces a new branching node. Second, the case $j - i = minP - k$, which means that an lvp exists multiple times in the lvp-repertoire of string S. For this, we do not need to introduce a new branching node, because the needed branching node already exists and is v.

There is no change in Algorithm 4.3 that might influence the runtime of the overall algorithm.

Algorithm 4.4 is given a reference pair for some node u and finds the canonical reference pair $(v', S_p^{k'})$ for u. This is done by walking down as many edges as necessary to arrive at v' such

Algorithm 4.4 Find the next canonized reference tuple

```
1:  function CANONIZE(v, k, p, p')
2:      if p = k then
3:          return (v, k)
4:      if v = NULL then                ▷ In the original algorithm, this is implemented implicitly.
5:          v ← root
6:          k ← k + 1
7:          p' ← end(S, k)
8:          minP ← min(p, p')
9:          if minP + 1 ≤ k then                ▷ Check if current lvp is invalid.
10:             return (v, k)
11:     minP ← min(p, p')
12:     while there is an edge (v, v', i, j) starting with S_k^k and j − i ≤ minP − k do
13:         k ← k + j − i + 1
14:         v ← v'
15:     return (v, k)
```

that it is the closest explicit ancestor of u, or u itself if u is explicit. In addition to Ukkonen's original algorithm, we now have to consider that the current node is *NULL*. This is handled implicitly in Ukkonen's version, as the parent node of the root node has an edge of length one to the root via any character from the alphabet. We have chosen to handle this explicitly here, but in the end, this is only an implementation and depiction detail. The only real difference to Ukkonen's version is again the introduction of variable *minP*, which is used to find the right end point of the edge in the case the lvp of the current suffix needs to shorten the edge. But this also does not influence the runtime of the algorithm.

In the end, the direct construction of the (viable) suffix tree in linear time can be used for RNA structure strings. For other, more complex grammars and languages, we can still use the linear time pruning approach.

4.5. Computation of lvp lengths for RNAs

In the construction algorithms for viable suffix trees, the lvps of all suffixes have so far been regarded as pre-computed. But in general, this pre-processing is crucial for the determination of the running time of the viable suffix tree construction. With Algorithm 4.5 and Figure 4.1, we will present the pre-computation of the lvp of every suffix from an input string that constitutes a well-formed RNA secondary structure. As before, the RNA secondary structure will be represented by dot-bracket strings. That is why the language represented by this viable suffix tree is the language of the grammar G_S from Example 2.3. By the way, the following algorithm will only work if the input string itself is a well-formed RNA secondary structure and not just any string

over the dot-bracket alphabet.

In order to store our precomputed values, a table named *End* is used. The values in this array are equivalent to the ones computed by the *End* function in Definition 4.3. In addition to the computation of the *End* table, the algorithm computes a second table named *Depth* as by-product[8]. This additional table shows the branching depth at a specific position of the RNA structure and is defined as the number of base pair open characters that have not yet been accompanied by their corresponding base pair closing characters. This additional table is required to judge (in constant time) whether any substring of the RNA structure constitutes a well-formed RNA secondary structure, i.e. the substring can be generated using G_S.

The algorithm iterates over all characters of the input string S. It distinguishes four cases: the current character stands for either base pair open, base pair close, unpaired base, or any other character that is not part of the alphabet, such as the sentinel character \$.

In the first case, the linked list *LL* that contains all elements inside the current branch, meaning all elements in between the opening and closing base pair characters, is extended by the current position. Subsequently, a new branch level for all characters in between the position i and the corresponding closing base at position j is created by pushing the list for the current level on top of the stack *ST* and emptying the list *LL*. All new characters in between i and j are added to *LL* unless a new opening or closing base pair character appears in the string. In the end, the current branching depth is increased.

In the second case when a closing base pair character is observed, the end positions for all elements of the current list *LL*, including the current position, are set to $i - 1$, which is the last position before the closing base pair character. Note that the corresponding opening base pair character is not part of the current list, because there might be unpaired bases following the current closing base pair character. In such a case, the lvp does not have to end at position i. After this, the top list of the stack *ST* that holds all elements of the branch level that follows this position becomes the current active list *LL* again. Subsequently, the current branching depth is reduced by one.

All unpaired base characters are added to the list *LL* that represents all positions of the current branch level. Since there is no change in the branch level, the branching depth remains unchanged, too. Last, for all characters that are not part of the alphabet, the procedure of case two can be used. That means the entry in the *End* table for all elements of the current list is set to $i - 1$, including the current position. There are no lists left on the stack *ST* if the input string S is a well-formed RNA structure.

Lemma 4.2. *Let S be a well-formed RNA sequence string of length n. The End table correctly holds all end positions for the lvps of all suffixes and the Depth table holds all branch levels.*

[8]For the construction of viable suffix trees and in other parts of thesis, we assume that the values of the tables *Depth* and *End* are accessed by using the corresponding functions *depth* and *end*.

Algorithm 4.5 Computation of the *End* and *Depth* table for a well-formed RNA structure

```
1:  function computeEND(S, Σ)
2:      initialize LinkedList LL and Stack ST        ▷ ST is used as stack of lists of type LL
3:      i ← 0
4:      Depth[−1] ← 0
5:      while i < |S| do                             ▷ iterate over every character in the string
6:          if S_i^i = ( then                        ▷ case 1 for base pair open
7:              LL.append(i)
8:              ST.push(LL)
9:              LL.clear()          ▷ Remove all elements from LL. ST won't be affected by this.
10:             Depth[i] ← Depth[i − 1] + 1
11:         else if S_i^i = ) then                   ▷ case 2 for base pair close
12:             LL.append(i)
13:             for x ∈ LL do
14:                 End[x] ← i − 1
15:             LL ← ST.pop()
16:             Depth[i] ← Depth[i − 1] − 1
17:         else if S_i^i = . then                   ▷ case 3 for unpaired base
18:             LL.append(i)
19:             Depth[i] ← Depth[i − 1]
20:         else if S_i^i ∉ Σ then          ▷ case 4 for other characters, like the terminal symbol
21:             LL.append(i)
22:             for x ∈ LL do
23:                 End[x] ← i − 1
24:             LL.clear()
25:             Depth[i] ← Depth[i − 1]
26:         i ← i + 1
```

Initialization S is input string. ST, LL are empty lists. $Depth[-1] \leftarrow 0$.

Transitions

S	ST	LL		S	ST	LL	***End*** & ***Depth*** **tables**
$(_i$R	T	M	$\Rightarrow$	R	$(i : M) : T$	[]	$Depth[i] \leftarrow Depth[i-1] + 1$
$)_i$R	T	$k : M$	$\Rightarrow$	$)_i$R	T	M	$End[k] \leftarrow i - 1$
$)_i$R	$M : T$	[]	$\Rightarrow$	R	T	M	$End[i] \leftarrow i - 1$ $Depth[i] \leftarrow Depth[i-1] - 1$
$._i$R	T	M	$\Rightarrow$	R	T	$i : M$	$Depth[i] \leftarrow Depth[i-1]$
$\$_i$R	[]	$k : M$	$\Rightarrow$	$\$_i$R	[]	M	$End[k] \leftarrow i - 1$
$\$_i$R	[]	[]	$\Rightarrow$	R	[]	M	$End[i] \leftarrow i - 1$ $Depth[i] \leftarrow Depth[i-1]$

Figure 4.1.: Function computeEND as transition system. The subscript shows the current position in the text, [] denotes the empty list, : denotes the prepend operator, and \$ represents all characters that are not part of the alphabet.

Proof. In the case of a closing base pair character, according to the definition, there is no lvp. This is covered in case two: the current position of the closing base pair character i is added to list *LL* and subsequently set $i - 1$. Since $i - 1 < i$, the closing base pair character does not have a lvp. The branch level of this position gets reduced by one in table *Depth*.

Next, in the case of an unpaired base character (case 3), we assume that the current position has branch level x and remember that it is guaranteed that the input RNA structure is well-formed. Based on this, we can conclude that the lvp only comprises the next characters that have a branch level $\geq x$ and end either at the position before the closing base pair character that has branch level $x - 1$ or one position before the sentinel. Each branch level is kept separate; all characters of the current level are present in list *LL* and the other ones are ordered descending on the stack *ST*. Once the level ends at a closing base pair character, all positions that are stored in *LL* are set to one position before the closing base pair character (case 2). In case of the sentinel, the same holds (case 4). The branch level of this position does not change and is copied from the previous one in table *Depth*.

Finally, the opening base pair character does not belong to the same branch level as the following characters that are enclosed by this and the corresponding closing base pair character; it belongs to the previous branch level. The reason for this is that the lvp starting at the opening base pair character position does not necessarily end at the corresponding closing base pair character position, because there may be unpaired base characters that follow the corresponding closing base pair character; those would still extend the lvp. Therefore, the current position is added to list *LL* before this is pushed onto the stack *ST* (case 1). In case of the *Depth* table, the branch level increases by one already for this position, even though the new branch level only starts with the following character. □

Lemma 4.3. *Let S be a well-formed RNA sequence string of length n. The End and Depth tables can be computed in $\mathcal{O}(n)$ using the* computeLVP *function.*

Proof. The algorithm iterates over all characters of the string of length n. Every character is added to the linked list *LL* exactly once when the while loop iterates over its position in the string. In case one, the current list is pushed on top of the stack *ST*; afterwards, the current list is cleared. This way, every element can only be in one list, either in the current one, namely *LL*, or in a list on the stack *ST*.

In cases two and four, there will be additional iterations over all items of *LL*. For case two, the current list is emptied and replaced by the one that is on top of the stack *ST*. This replacement of *LL* is not necessary for case four, because it is a terminal symbol and *ST* is empty at this point. So, in addition to the while loop every element's *End* value is set either in case two or four. Hence, the runtime stays linear and is $\mathcal{O}(n)$. □

In order to decide whether a substring S_j^i of S is viable, we use the function isValid. The decision can be made in constant time if the tables *End* and *Depth* are already computed.

Algorithm 4.6 Check if a substring represents a viable prefix of a suffix starting at position i

```
1: function isValid(S, i, j)
2:     return (end(i) ≥ j and depth(i − 1) = depth(j))
```

Lemma 4.4. *Let S be a well-formed RNA sequence string of length n. If the substring S_j^i is a well-formed RNA sequence itself, then the function* isValid *can decide this in constant time using the precomputed tables End and Depth.*

Proof. Obviously, if the lvp starting at position i does not cover position j, then the substring cannot be viable.

Since S is well-formed itself, only nested base pairs are accepted. Therefore, it is sufficient to check whether the branch levels of positions $i-1$ and j are equal. The position $i-1$ is used instead of i, because the branch level of the opening base pair character in the *Depth* table is already increased by one, even though this position still belongs to the previous level. Furthermore, this does not have an influence on unpaired base characters at position i, because in this case, the branch level in the *Depth* table is always equal to the previous position. In case position i is a closing base pair character, the first clause fails.

In all other cases, the branch levels are different, which proves that the substring S_j^i cannot be a well-formed RNA sequence. □

4.6. Finding maximal repeated pairs in viable suffix trees

The main motivation for introducing viable suffix trees is the search for maximal repeated pairs of RNA secondary structures that should be well-formed RNA structures themselves. Using classical suffix trees or arrays, we need to manually parse the output of all maximal repeated pairs and subsequently analyze them to find the longest substring of the maximal repeat that is a well-formed structure. The post-processing step increases the time complexity in the worst-case by another factor $|S|$.

Gusfield [43] describes how to find all maximal repeated pairs in a suffix tree of string S in $\mathcal{O}(|S| + k)$ time, where k is the number of maximal repeated pairs that can be found. This procedure expects a given suffix tree for S and uses a bottom-up approach that starts at the leaves of the tree. For each leaf m, the algorithm reports the character S_{m-1}^{m-1} to the left of the suffix to its parent internal node. Any internal node n needs to create at most $|\Sigma|$ linked lists in order to store the leaf positions into the corresponding list indexed by the characters S_{m-1}^{m-1}. All leaves that are grouped into one list are preceded by the same character and share a common prefix until n. In order to create the lists for any internal node n, the algorithm needs to link the lists of the children with the same index character together. The creation of all lists for all nodes can be done in $\mathcal{O}(|S|)$ time, because the alphabet size can be expected to be finite.

Algorithm 4.7 Find all maximal repeated pairs of S using viable or classical suffix tree T

```
1:  function FindMaximalPairs(S, T)
2:      ProcessNode(S, T, root)
3:  function ProcessNode(S, T, n)
4:      let I be an index of lists for all characters in Σ
5:      if n is child then
6:          add position n to list I[S_{n-1}^{n-1}]          ▷ list at index position S_{n-1}^{n-1}
7:      else
8:          let LC be the list of all children of n
9:          let II be the index of the indices returned by all children
10:         for m ∈ LC do                                    ▷ process each child
11:             II[m] ← ProcessNode(S, T, m)
12:             for c ∈ Σ do                                 ▷ link the list from the children
13:                 link I[c] with II[m][c]
14:         for x ∈ {0, ..., |LC| − 1} do                    ▷ build Cartesian product
15:             o ← LC[x]                                    ▷ current node o
16:             for c ∈ II[o] do                             ▷ loop through all lists at o
17:                 for y ∈ {x + 1, ..., |LC| − 1} do        ▷ loop through remaining nodes p
18:                     p ← LC[y]
19:                     for d ∈ (II[p] \ {c}) do             ▷ loop through every character except c
20:                         for (i, j) ∈ (II[o][c] × II[p][d]) do
21:                             report maximal pair (i, j, depth(n))
22:     return I
```

To report all maximal repeated pairs of the form $(i, j, depth(\mathsf{n}))$[9] at internal node n, the algorithm loops through each character c and child o of n and forms the *Cartesian* product of c's list at o with the union of every list for a character that is not c at any other child of n that is not o. Every pair generated by this algorithm represents the starting positions of a maximal repeated pair of S while $depth(\mathsf{n})$ represents the length of the repeat. An illustration of the procedure is depicted in Algorithm 4.7.

This algorithms works on classical suffix trees as well as viable suffix trees, but it does not guarantee that all maximal repeats represent viable words. This becomes apparent by looking at Example 4.1, where not all internal nodes of the viable suffix tree represent viable words. So, if we are interested in maximal repeated pairs of all viable suffixes independent of the question whether the repeats they are representing are viable words, then Algorithm 4.7 is sufficient. Otherwise, we need to find the lvp of the maximal repeat and need to shorten the length of the pair accordingly before reporting it. We will use the term *viable maximal repeated pairs* to denote such pairs. In general, this can be done by iterating over every character of the

[9] Note, before we used a tuple of two pairs of indices $((i, j), (k, l))$ as maximal repeated pair, but for the sake of a simpler argument in this section, we replace the end positions in both pairs by the length of the maximal repeat as third element in the now 3-tupel.

Algorithm 4.8 Compute table *Prev*

```
1:  function computePREV(S, Σ)
2:      initialize Stack ST                          ▷ ST stores the current lvp for each branch level
3:      hp_loop ← False      ▷ hp_loop stores whether the current position is in a hairpin loop
4:      cur ← −1                                     ▷ cur stores the current lvp for the current branch level
5:      i ← 0
6:      ST.push(−1)
7:      while i < |S| do                             ▷ iterate over every character in the string
8:          if S_i^i = ( then                        ▷ case 1 for base pair open
9:              ST.push(cur)
10:             hp_loop ← True                       ▷ inside hairpin loop
11:         else if S_i^i = ) then                   ▷ case 2 for base pair close
12:             ST.pop()
13:             cur ← i
14:             hp_loop ← False                      ▷ outside hairpin loop
15:         else if S_i^i = . and not hp_loop then   ▷ case 3 for unpaired base
16:             cur ← i
17:         Prev[i] ← ST.top()
18:         i ← i + 1
```

maximal repeat resulting in a runtime of $\mathcal{O}(|S|^2 + k)$, because this has to be done only once for every maximal repeat and not for every maximal repeated pair. At first, it seems that there is no benefit in using viable suffix trees over classical suffix trees for the purpose of identifying viable maximal repeated pairs, but Example 4.1 illustrates that the number of internal nodes is significantly smaller in viable suffix trees than in classical suffix trees, at least for our RNA example, resulting in a faster practical runtime.

For the purpose of identifying all viable maximal repeated pairs of well-formed RNA structures, we can further improve the approach. Therefore, we introduce a new table *Prev*[10], whose values represent the end of the lvp of the previous branch level. The computation of the table is illustrated in Algorithm 4.8 and takes $\mathcal{O}(|S|)$ time. Note that table *Prev* may also be computed at the same time as tables *End* and *Depth* in Algorithm 4.5.

Example 4.4. The following example shows the values for the *Prev* table for a well-formed RNA secondary structure *S*.

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S	=	(	(	.	)	(	(	(	.	)	.	(	.	)	)	(	(	.	)	.	)	)	)
$depth(S, i)$	=	1	2	2	1	2	3	4	4	3	3	4	4	3	2	3	4	4	3	3	2	1	0
$prev(S, i)$	=	-1	-1	-1	-1	3	3	3	3	3	3	9	9	3	3	13	13	13	13	13	3	-1	-1

Before we explain how to use the *Prev* table to speed up the computation of viable maximal repeated pairs for well-formed RNA structures, we will define a new function *fdepth* that produces equivalent values to the ones in *Prev*.

[10] Similar to the previous tables, the values of table *Prev* can be accessed by using the function *prev*.

Definition 4.8. Let *fdepth* be a function for well-formed RNA structure string S and input position i that is defined as follows:

$$fdepth(i) = \max\left(\left\{j \in \{0, \ldots, i-1\} \mid S_j^j \in \{\,.\,,\,)\,\} \wedge depth(j) < depth(i)\right\} \cup \{-1\}\right).$$

Function *fdepth* needs table *Depth* to be filled. For every position of structure S, the function *fdepth* returns the end position of the lvp that is closest to i and has a branch level smaller than i. If there is no lvp before i, then the function returns -1. We can be sure that the lvp ends at position j, because a viable prefix must end at a position with a base pair close or unpaired base character. In the case that the unpaired base lies within a hairpin loop and cannot be the end position of a viable prefix, its branch level will be greater than the one of position i.

Lemma 4.5. *Algorithm 4.8 fills table Prev for every position i in S with $fdepth(i)$.*

Proof. The algorithm starts with the closest lvp for the current branch level initialized to -1, because there has not been a viable prefix yet. In case an opening base character is read, it is set to the closest lvp of a smaller branch level using variable *cur* that holds the value of the previous branch level. As the branch level increases with this character, the closest lvp end position for the previous branch level is pushed onto the stack to save it.

The base pair close character is a special case, because it marks the end position of the previous branch level; but on the other hand, it already belongs to the smaller branch level. By setting the variable *cur* to i, it marks the end of the greater branch level. Furthermore, by removing the top element from the stack, the branch level decreases. Now, the algorithm just needs to look up the position of the closest end position of the lvp of a smaller branch level on top of ST.

In the case of an unpaired base, the algorithm needs to decide whether the current position is within a hairpin loop or not. This can be done by checking a variable that was set in cases one and two. If it is not within a hairpin loop, then the end position of the lvp of the greater branch level needs to be set to position i, because an unpaired base that is not within a hairpin loop always marks a viable end for such a lvp. Subsequently, the closest end position of the lvp of a smaller branch level can be looked up using the stack ST.

Last, the sentinel always has branch level 0 and is set to -1, which is the last value on top of ST. □

Table *Prev* can now be used to speed up the search for the lvp of a maximal repeat. The approach is illustrated in Algorithm 4.9. Instead of reporting the maximal repeated pair $(i, j, depth(\mathsf{n}))$ at internal node n, we can now output the viable maximal repeated pair $(i, j, \textsc{Lvp}\textsc{Prev}(S, i, i + depth(\mathsf{n}) - 1)$ that shows the length of the lvp of the maximal repeat rather than the full length of the path leading to n.

Algorithm 4.9 Report length of the lvp of the suffix starting at i and ending at a position $\leq j$

```
1: function LvpPREV(S, i, j)
2:     while j ≥ i do
3:         if isValid(S, i, j) then
4:             return j − i + 1
5:         else
6:             j ← prev(j)
7:     return 0
```

Lemma 4.6. *Algorithm 4.9 can be used by Algorithm 4.7 at every internal node to identify all viable maximal repeated pairs of well-formed RNA structures in $\mathcal{O}(|S|\, d + k)$, where d is the greatest branch level of S.*

Proof. Gusfield [43] has shown that maximal repeated pairs can be reported in $\mathcal{O}(|S| + k)$. Therefore, it remains to be shown that the lvp of a maximal repeat from i to j, which is represented by an internal node n, can be identified in $\mathcal{O}(d)$ time.

First, the case that the maximal repeat starts with a closing base pair is not possible, because such a string can never be viable; therefore, it does not have to be handled.

Next, assume position i is an opening base pair with the corresponding closing base pair being at position k. In case $j < k$, there cannot be a viable prefix for this repeat. The function LvpPREV starts at position j and looks up the value for $prev(j)$, which is guaranteed to have a lower branch level. Since no viable prefix that ends before position j exists, a lookup in table *Prev* is performed at most d times until the looked up value is smaller than i. No viable maximal repeated pair has to be reported. On the other hand, if $j = k$, then the maximal repeat is the lvp itself and the isValid function shows that on the first call. For the case $j > k$, the lvp of the maximal repeat must exist. The end position of this lvp is the base pair closing or unpaired base character that is closest to j and whose branch level is one level lower than the branch level of position i, because table *Depth* is assigning the base pair open character already to the new, greater branch level it introduces. Function LvpPREV finds this position by (multiple) *Prev* table lookups that reduce the branch level step by step until the lvp of the repeat has been identified. This also needs at most d lookups in table *Prev*.

Last, if position i corresponds to an unpaired base character, then we have to distinguish whether a base pair close or open character appears first in the suffix. In the former case, the maximal repeat is the lvp itself, because a viable suffix tree cannot contain a path in which a base pair close character appears before a base pair open character. In the latter case, if the next base pair open character at position l has a corresponding base pair close character $k \leq j$, then this case will be handled as explained before. Otherwise, if the corresponding base pair close character $k > j$, then the end position of the lvp will be the first looked-up value that is smaller than l, because $depth(i) = depth(l) - 1$. In any case, the algorithm will at most need d

lookup operations to report the length of the lvp. □

We have analyzed all RNA secondary structures from the current *Rfam* database [79] in order to examine the branch levels. The maximal level encountered was 17, but more than 96% of the structures have a branch level that is 3 or lower. Based on this analysis, it is safe to assume that at least for well-formed RNA structures, our new approach for finding all viable maximal repeated pairs can greatly reduce the asymptotic as well as practical runtime.

4.7. Implementation & benchmarks

All three presented approaches for constructing either classical suffix trees using Ukkonen's algorithm, viable suffix trees for well-formed RNA structures using a modified version of Ukkonen's algorithm, or viable suffix trees for any given language using the pruning approach have been implemented using the programming language C++. The data structure for the (valid) suffix tree is organized as described by Kurtz in [65]. The internal nodes can be represented using five integers: start position in the string, length of the path, branch brother, first child, and suffix link. Leaves, on the other hand, only need to store one additional integer, their branch brother. Viable suffix trees additionally require the storage of the *Lvp* table, which has an entry for every symbol in the input string. The tables *Depth* and *Prev* are not required during the generation of the viable suffix tree, but are useful afterwards whenever the tree is processed and analyzed further.

To analyze the space requirements for the classical suffix tree as well as the viable suffix tree, we took all 368,500 unique structure strings that were generated by using minimum free energy folding of all sequences present in the *Rfam* database.

Table 4.3 shows the space requirements for the viable suffix tree as well as the classical suffix tree. The table *Lvp* is the only additional table that is needed to traverse the viable suffix tree. If this representation is chosen for storing the tree, then the viable suffix tree reduces the required space by 30% compared to the classical suffix tree. But, if we additionally want to store *Depth* and *Prev*, then the space requirement is only slightly smaller than that of the classical suffix tree. The storage of these two additional tables is not required for improving the asymptotic worst-case runtime, because they can be computed on-demand in linear time. The reason for the reduced space requirement is the smaller number of internal nodes, because the smaller number of leaves does not influence the number of cells for the stored tables. These findings are similar for all data set sizes from 10% of the *Rfam* data set up to 100%.

Additionally, we have repeated the experiment for a different alphabet and the results are shown in Table 4.4. Instead of using the . character for unpaired bases, the symbol was replaced by the actual nucleotide at this position. With the increased alphabet size, the difference in space usage becomes smaller. Compared to the 60% to 70% reduction in the previous experiment, the

Table 4.3.: The table shows the space requirements of the viable suffix tree in the normal and the extended version (including tables *Depth* and *Prev*) as well as the classical suffix tree as bytes per symbol. Integers were assumed to consume 4 bytes, except for the *Depth* table, where we assume 1 byte. The table lists ten different benchmark data set sizes from 10% of the *Rfam* database up to 100%. The alphabet of the structure strings is $\Sigma = \{(,), .\}$.

		viable suffix tree				classical suffix tree		
%	symbols	$\frac{\text{bytes}}{\text{symbol}}$	$\frac{\text{bytes}}{\text{symbol}}$ ext.	internal nodes	leaves	$\frac{\text{bytes}}{\text{symbol}}$	internal nodes	leaves
10	5,195,245	13.13	18.13	1,332,406	3,641,990	18.81	3,847,754	5,195,245
20	10,375,549	12.94	17.94	2,565,299	7,273,634	18.64	7,595,297	10,375,549
30	15,569,011	12.84	17.84	3,768,084	10,914,540	18.54	11,320,363	15,569,011
40	20,754,683	12.76	17.76	4,944,466	14,550,745	18.47	15,014,590	20,754,683
50	25,946,689	12.71	17.71	6,107,351	18,190,564	18.41	18,698,044	25,946,689
60	31,147,162	12.66	17.66	7,259,423	21,836,058	18.37	22,374,157	31,147,162
70	36,327,228	12.62	17.62	8,395,460	25,467,702	18.33	26,021,842	36,327,228
80	41,514,035	12.59	17.59	9,523,483	29,104,058	18.29	29,665,658	41,514,035
90	46,703,942	12.56	17.56	10,643,898	32,742,799	18.26	33,301,797	46,703,942
100	51,897,293	12.53	17.53	11,757,339	36,383,489	18.23	36,934,437	51,897,293

Table 4.4.: Same as above, but instead of using the `.` character for unpaired bases, the symbol was replaced by the actual nucleotide at this position. Therefore, the alphabet of the structure strings is $\Sigma = \{(,), \mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$.

		viable suffix tree				classical suffix tree		
%	symbols	$\frac{\text{bytes}}{\text{symbol}}$	$\frac{\text{bytes}}{\text{symbol}}$ ext.	internal nodes	leaves	$\frac{\text{bytes}}{\text{symbol}}$	internal nodes	leaves
10	5,195,245	12.93	17.93	1,275,335	3,637,101	14.45	2,711,291	5,188,312
20	10,375,549	12.81	17.81	2,496,879	7,276,462	14.39	5,393,119	10,378,787
30	15,569,011	12.75	17.75	3,708,027	10,922,455	14.37	8,074,946	15,580,084
40	20,754,683	12.72	17.72	4,907,055	14,557,813	14.35	10,745,512	20,764,921
50	25,946,689	12.70	17.70	6,095,867	18,189,159	14.34	13,410,702	25,945,544
60	31,147,162	12.68	17.68	7,276,877	21,828,370	14.33	16,076,394	31,135,341
70	36,327,228	12.65	17.65	8,449,773	25,465,905	14.32	18,736,160	36,323,843
80	41,514,035	12.63	17.63	9,617,091	29,103,554	14.31	21,391,977	41,512,710
90	46,703,942	12.62	17.61	10,780,430	32,745,382	14.30	24,048,010	46,708,004
100	51,897,293	12.60	17.60	11,938,424	36,383,489	14.29	26,698,016	51,897,293

viable suffix tree now needs to store around half the number of internal nodes that the classical suffix tree stores. Now, the extended viable suffix tree that includes the tables *Depth* and *Prev* needs more space than the classical suffix tree. Therefore, we can conclude that the increased alphabet greatly effects the number of internal nodes in the classical suffix tree while the space requirement of the viable suffix tree seems to be independent from the alphabet size, at least for our example of nested RNA secondary structures.

4.8. Conclusion

In this chapter we have presented a new data structure, the viable suffix tree, that can be used to reduce the memory footprint of suffix trees by only inserting the viable suffixes. The data structure can be generated either by pruning the classical suffix tree or, for nested RNA secondary structures, by direct construction using a modified version of Ukkonen's algorithm. Note that our pruning approach is not able to retain the suffix links in comparison to the modified Ukkonen algorithm. Also, the viable suffix tree can be directly created using the wotd algorithm, but this will not be discussed in this chapter. The construction using the wotd algorithm is almost self-explanatory, because using the *Lvp* table, the remaining suffixes of any internal node can be distinguished easily from the suffixes that have no lvp. On the other hand, the construction using the wotd algorithm needs quadratic time.

The theoretical importance of our pruning approach lies in the following: it shows that the viable suffix tree can be constructed in linear time for any decidable grammar language L, given that the *lvp* function can be computed in linear time. The complexity of L is reflected solely in the effort to compute the *End* table. Our direct construction for the (context-free) language of RNA secondary structures achieves a linear-time viable suffix tree construction. How this task is solved in general for languages in the Chomsky hierarchy is an open research challenge.

Finally, note that the viable suffix tree can be used to solve the word problem for a given language L and string S. Given the viable suffix tree for S, simply check whether the leaf that is representing the whole string is present in full length in this tree. This shows that the overall complexity of constructing the viable suffix tree cannot be lower than the complexity of the word problem for L.

CHAPTER

5

AN UNBIASED RNA BENCHMARK DATA SET

Contents

The content of this chapter, together with the corresponding Appendix B, was published in [69] with co-authors Cédric Chauve, Yann Ponty, Robert Giegerich, and myself as the main contributor. It covers the explanation of the inglorious BRaliBase dent and was a prerequisite to create an unbiased sampled benchmark data set later.

5.1. Benchmark data sets for biological data

The role of benchmarks data sets As much as biosequence databases are an invaluable resource in molecular biology research, bioinformatics benchmarks are a crucial aid in the continued development and evaluation of bioinformatics tools and algorithms (see for example [23] for multiple sequence alignment). They allow us to compare old and new approaches to the same problem with respect to their use of computational resources, as well as with respect to their qualitative performance. When tool developers can make use of established benchmarks, reproducibility of results benefits greatly.

Good benchmarks data sets must satisfy a number of criteria. They should contain the best, curated data sets available at their time, they should reflect the diversity in their problem domain, and they should not be biased towards problem subdomains of a specific nature. Naturally, these criteria are not easy to fulfill, especially when the problem domain itself is a

new or difficult research topic, and available data at a given time may only partially reflect its diversity and complexity. For practical use, a benchmark should be subdivided such that users can extract subsets with characteristic properties, such as sequence length, degree of conservation, or phylogenetic classification.

Note that benchmarks do not need to encompass the complete data available in their domain, contrasting in that respect with databases used in applications. However, if domain knowledge grows, benchmark data sets need to be updated once in a while in order to remain representative of their problem domain.

Two successful benchmarks: BAliBASE and BRaliBase Widely known and used in sequence analysis is the BAliBASE Benchmark, first compiled in 1999 by Thompson *et al.* [100]. It provides hand-curated multiple sequence alignments categorized by core blocks of conservation sequence length, similarity, and the presence of insertions and N/C-terminal extensions. It has seen several updates, the most recent being version 3.0 compiled in 2005 [99]. It has been used in countless sequence analysis studies[11] and how to use and interpret the scores provided by BAliBASE has become a research topic of its own [14].

In the first comparative evaluation of structural alignment algorithms [32] in 2004, the lack of an established benchmark was strongly felt. Motivated by the success of BAliBASE, but targeting a smaller research community, the BRaliBase benchmark was first compiled in 2005 [33] and enhanced in 2006 [112]. It makes available a test-set of *structural* alignments of RNA sequences that has been an important resource to many researchers, including ourselves, in comparative RNA bioinformatics.[12]

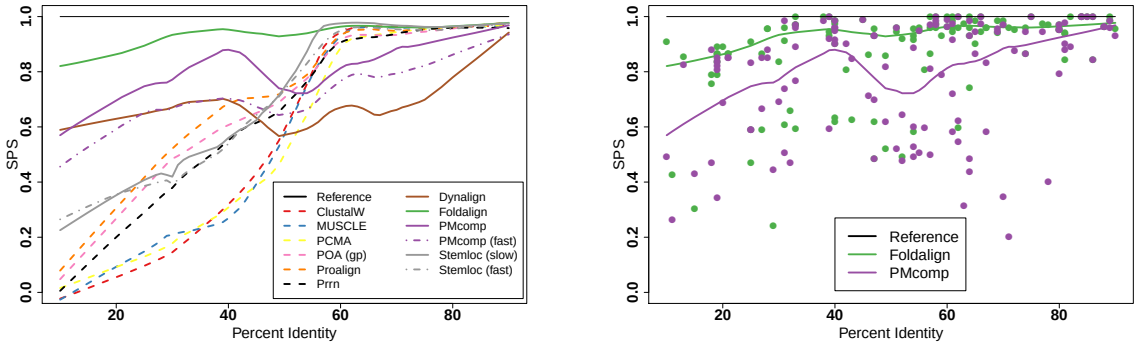
This study is dedicated to a peculiarity of the BRaliBase benchmark that we call the "BRaliBase dent".

5.2. The BRaliBase dent

Here, we will describe a phenomenon called the BRaliBase dent, and using this name is already half the message. Our claim is: what has long been considered a flaw in the performance of the algorithms is better explained by a property of the benchmark data set. The BRaliBase dent refers to the puzzling phenomenon that even the best programs for structural RNA alignment seem to exhibit a weakness of performance in a range of moderate sequence similarity. At least, this is what we used to think. Let us take a closer look.

Originally, BRaliBase was used to demonstrate that enhanced sequence alignment approaches, which in some way take account of conserved RNA structure, yield more realistic alignments than the best (structure-unaware) sequence alignment programs.

[11] The Google Scholar reference count for the above two publications comes close to 800 at the time of this writing.
[12] Again according to Google Scholar, both BRaliBase publications have been cited above 300 times.



(a) **(b)**

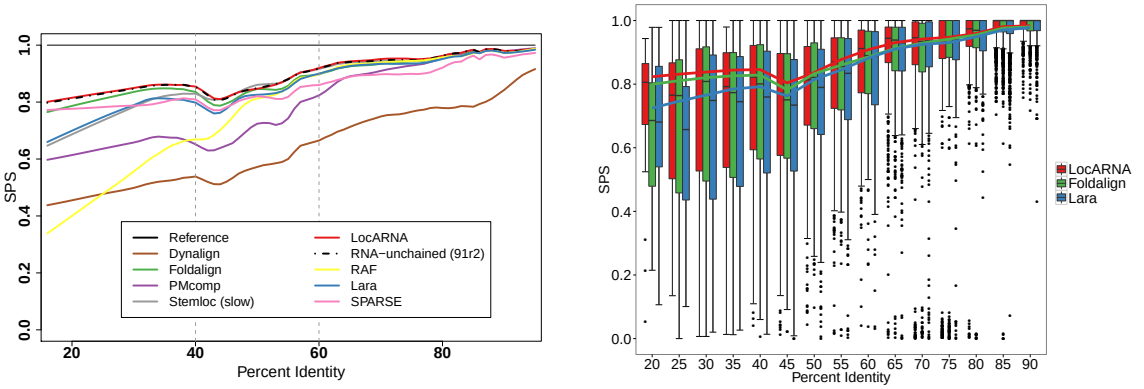
Figure 5.1.: (a) Original BRaliBase evaluation of 2005 [33]. Dashed lines show pure sequence aligners, solid lines show structural aligners, and dotted-solid lines show structural aligners with varying parameters. (b) Extended evaluation for `Foldalign` and `PMcomp` that shows all results for the 118 pairwise alignments for both tools using the original data. SPS (sum-of-pairs score) is a measure of alignment accuracy compared to a reference data set introduced in [100].

The study of Gardner *et al.* [32] focused on comparative RNA structure prediction and classified structural RNA alignment algorithms in three categories:

- Plan A aligns related sequences by pure sequence alignment. Then, it folds the alignment as a whole. Thus, while the second phase assigns the best possible structure to the given alignment, this alignment is determined solely in the first phase, and its quality is expected to degrade with decreasing sequence similarity.
- Plan B implements some form of the Sankoff algorithm [95], simultaneously optimizing sequence similarity and a folding score. Such algorithms have much higher resource requirements than Plan A but are expected to give better results for diverged sequences.
- Plan C first suggests a set of alternative structures for each sequence separately, aligns the structures (possibly ignoring their sequence content), and then derives a sequence alignment from the structure alignment. At the time of the study [32], no Plan C approach was known.

Besides the above, there are a number of approaches that do not fit these categories, avoiding dynamic programming over the whole search space and using heuristic or probabilistic methods. However, many of the most used methods follow one of the three plans outlined above and provide a large enough corpus for our study of the BRaliBase dent.

The BRaliBase evaluation of 2005 (Figure 5.1a) showed that the expectations indeed hold. Pure sequence alignment methods quickly loose quality when sequence identity drops under



(a) **(b)**

Figure 5.2.: (a) Re-evaluation of the 2006 BRaliBase data from [112] with currently available structural aligners. (b) The same re-evaluation with only three tools and box plots showing the detailed distribution of the SPS. Here, we have chosen to add `LocARNA` as the best performing tool and substituted `PMcomp` by `Lara`, because some of `PMcomp`'s alignment computations resulted in errors and `Lara` represents an interesting alternative not fitting into the previously mentioned categories. (See Table B.1 in Appendix B for details.)

75%, and folding the alignment cannot compensate. (As we know today, post-processing Plan A with a Plan C-type approach can locally improve the alignment and provide a partial compensation [17].) Plan B type algorithms, as of 2005, perform better. With decreasing sequence identity, their performance degrades more gracefully overall. Figure 5.1b shows the effect of smoothing on two of the curves. The green (`Foldalign`), purple (`PMcomp`), and brown (`Dynalign`) curves represent the three best performing Plan B approaches. To a varying degree, they show a decline in the area between 60% to 40% identity, while their performance improves again when sequence conservation becomes even lower. This phenomenon is what we call the BRaliBase dent.

Today, the dent persists. Figure 5.2a shows a recent re-evaluation of the extended 2006 BRaliBase data with currently available algorithms.[13] The shape of the dent has slightly changed, but the position in the range between 60% to 40% identity is consistent. This can be accounted to the 76-fold increase in data points, for which the effect of smoothing can be seen in Figure 5.2b.

In 2007, a Plan C approach was designed. It first computed a set of abstract consensus shapes [91, 105], aligned their shreps with `RNAforester` [46], and picked the best structure alignment to derive the final sequence alignment from it. Aside from better speed, we hoped for a performance improvement in the dent zone. Testing the new approach, it perfectly

[13] The ability to run a re-evaluation a decade later is a benefit of having persistent benchmarks in the first place. Note, [112] only tested their extended data set on pure sequence alignment methods. To our knowledge, [10] was the first publication to use the 2006 BRaliBase data set for benchmarking their structural aligner `Lara`.

fell between `Foldalign` [45] and `PMcomp` [47], exactly reproducing the dent in the twilight area. This was puzzling, since the approach is quite different in nature from the Sankoff-type approaches. We felt that human curators do something special in the twilight zone of 60% to 40% identity that the algorithms cannot do. Anyway, the approach was never published. (For details see Appendix B.2.)

Researchers kept working on the structural alignment problem, using BRaliBase to evaluate their ideas [10, 16, 27, 45, 47, 96, 111]. And, in fact, there was some indication that the dent had been mastered. For example, publications [83, 110] performed benchmarks of their tools using the 2006 version of BRaliBase, indicating a dent-less performance of the algorithms. However, a closer look shows that this impression is due to the use of higher smoothing factors for creating the regression curves. In Figure 5.2b, we find a clear collapse of the SPS around 45%, which indicates that the use of a lower smoothing factor is key in order not to artificially mask the dent.

In 2015, one of the co-authors (Cédric Chauve) contributed to an algorithm not fitting the above categories but mimicking the way a human curator might work [15]. The algorithm `RNA-unchained` first picks significant sequence and structure matches, takes them as alignment seeds, and then uses `LocARNA` to align the rest. Evaluations in [15] also suggested a dent-less performance. However, since the algorithm is a hybrid of seed recognition and Plan B-type alignment using `LocARNA`, it happens that no seeds are found for data of low similarity. In such a case, the result is purely determined by `LocARNA` and these points were dropped from the diagrams, as they do not indicate a contribution of the new algorithm tested. If we include these points of measurement (Figure 5.2a), the dent comes back even with this approach. (For details see Appendix B.3.)

Understanding the reason for the BRaliBase dent is a natural question. In particular, one can wonder if it is due to an intrinsic weakness of the algorithms applied on RNA sequences with average identity, in which case addressing this weakness can be seen as a methodological goal. However, with so many algorithms based on independent ideas showing the same performance characteristics, we were led to suspect that the dent was a feature of the data rather than the algorithms.

5.3. Explaining the BRaliBase dent

For our exploration of the origin of the dent, we will use a single Plan B tool, `LocARNA`, and the k2 benchmark data set of the 2006 version of BRaliBase that consists of 8,976 pairwise alignments from 36 different RNA families that are based on seed alignments of *Rfam* 7.0 [41].

The consensus structure: An abstraction *Structural* alignment uses sequence comparison and structure prediction to elucidate a consensus structure for a group of functionally related

RNA molecules. Such a consensus structure may provide hints regarding the mechanisms underlying the molecule's biological function. However, it must be kept in mind that the consensus structure is only a theoretical construct. Each molecule performs its function as an individual, by a structure it folds into all alone. Not all parts of the sequence are equally determined by the function; hence, they might fold into a conserved structure to a varying degree.

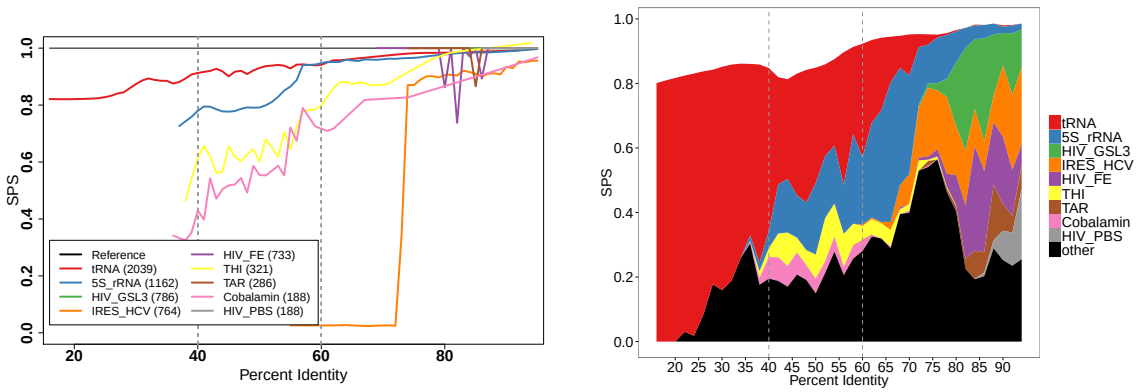
Take two extreme examples: tRNAs must attain their typical L-shaped 3D structure to fit in the active site in the ribosome as a whole. Hence, this structure and the underlying 2D cloverleaf structure is strongly preserved, even when the sequence is diverged. On the other hand, consider self-splicing introns: they have a strongly conserved catalytic site, embedded in a pseudo-knotted structure, but may also include large introns (even including nested self-splicing introns) that are irrelevant for self-splicing; thus, they cannot be expected to admit a good structural alignment overall. When a benchmarked tool achieves a lower alignment score for RNA family A compared to family B, this does not necessarily mean that it performs worse – it may simply mean that the structure is less conserved in family B and the tool reflects this fact.

Our best test data: tRNAs. Next to rRNA, the tRNA may be the most well-studied class of functional RNA molecules. Since structure conservation extends over the whole molecule, and since tRNAs are part of the universally conserved translational machinery, we have an enormous amount of well-curated structural alignments, where structure conservation is high even with low sequence identity. Since tRNAs are so short that all tools can handle them in reasonable time, they constitute a favorite test case for algorithm developers. This is a strong argument to include a large number of tRNA alignments in a benchmark data set. In fact, tRNAs is the most highly represented family in BRaliBase, contributing 2,039 data points. Can too much of the good be hurting in the (bitter) end?

Performance variation on different RNA families Before including all measurements into a single benchmark, it may be wise to look at the results for individual families. We do so in Figure 5.3a.

It is apparent that the tRNAs have a special position within the data set. They are represented by the red line. Compared to other families, two observations stand out: first, the computed tRNA alignments perform much better over the whole range of sequence identities than those in other families. This is surprising, since the performances of most other families seem to decline noticeably with decreasing sequence identity. Even though the curves for some families like THI or Cobalamin behave in a rather erratic manner, all of them are decreasing more strongly in principle. (See Figure B.2 for the plots of all 36 families.)

Second, since these curves are generated using local regression, the volatility is an indicator of strongly fluctuating values as well as a small amount of data points for various ranges of



(a) **(b)**

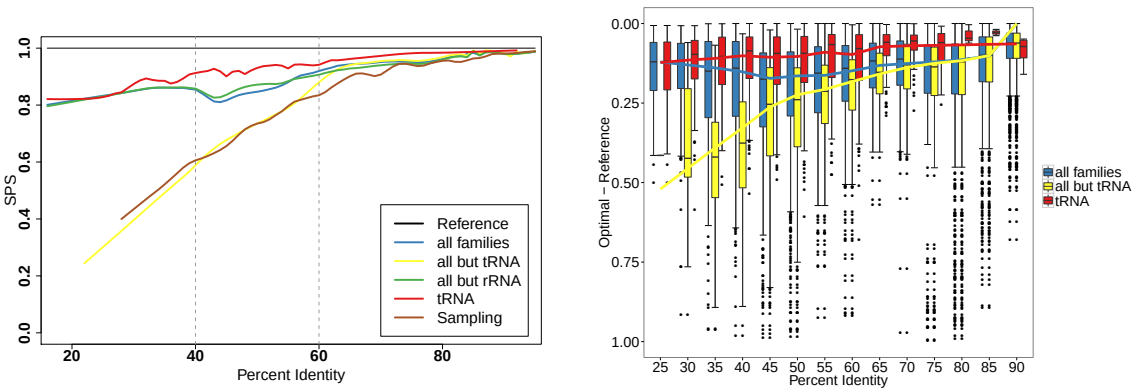
Figure 5.3.: The two plots show 9 of 36 RNA families with at least 180 alignments. (a) Familywise performance of LocARNA. The family names in the legend are further accompanied by the total number of alignments for each family in square brackets. (b) Each families share of LocARNA's SPS (after local regression) per sequence identity. The remaining families with less than 180 alignments are grouped into "other".

identities. Figure 5.3b confirms this assumption by showing that the number of alignments of similar identities fluctuates strongly even within the same RNA families. The most important point is, however, that the tRNA alignments have many data points in the range between 20% and 60% identity, while the other families have the highest number of data points in the high identity region. This is taken to the extreme for alignments with sequence identities lower than 40%, where there are almost no data points other than the tRNAs. We can separate the data set into different groups: first, the tRNAs with a rapidly shrinking share in the high identity area and the overwhelming majority in the low identity region. Second, the 5S_rRNAs resemble the remaining families, but their share has slightly different peaks for identities bigger than 55%. Last, all other families follow the same trend but with different peak heights and can be considered as background. (The proportion of alignments in all families are individually reported in Figure B.3.)

The combination of all these observations suggests that the dent is mainly caused by the conserved alignments of the tRNA family as well as the unbalanced composition of the data set: with tRNAs aligning especially well, and few other data points in the area of low similarity, when combining all performance curves the resultant ends up rising again in the low similarity area.

In order to verify this hypothesis, we have considered in details the performance of LocARNA

[14] LocARNA was substituted by PMcomp for the ease of a scoring scheme that is easy to reverse engineer and implement.



(a) **(b)**

Figure 5.4.: Separate evaluation of tRNA alignments, non-tRNA alignments, and the complete data set. Comparison (a) by BRaliBase SPS and (b) as length normalized score differences between the optimal version and the reference using `PMcomp`'s scoring scheme.[14] Additionally, (a) shows a curve for a sample approach in which no families share is bigger than 20% per sequence identity and the non-5S_rRNA alignments.

as representative for all tools for tRNA and non-tRNA alignments separately (Figure 5.4a). Both show a decline with decreasing similarity, but the decline for non-tRNAs (yellow) is much steeper. Where non-tRNA data become sparse (around 40% identity), the tRNA curve (red) almost entirely determines the shape of the overall curve (blue). Additionally, the non-5S_rRNAs curve (green) shows a behavior similar to the overall curve, indicating that the second most abundant family does not influence the presence of the dent. Therefore, we can summarize that the dent in the 2006 version of BRaliBase is caused by the interaction of two data set-specific factors: the exceptionally good performance of tools over tRNAs, as well as the lack of non-tRNA alignments, which would probably yield lower performances, in the identity region below 40%.

In order to eradicate the imbalance, we tested a sampling approach in which no single RNA family is allowed a bigger representation than 20% within each sequence identity region. The resulting curve (purple) shows a clear decreasing SPS tendency for decreasing identities with some smaller dents in the high identity regions that cannot be avoided.

The BRaliBase SPS measures the agreement of computed alignments to a reference. But during the computation of the predicted alignment, different alignments have to be computed and scored internally in order to choose the optimum. This allows us to reverse our view point. Namely, we score the reference alignment using a tool's internal scoring scheme and compare it to the prediction, under the rationale that the special status of tRNAs in the benchmark should also be apparent in this comparison.

Figure 5.4b compares the `PMcomp` score of the optimal alignment and the reference alignment by representing the length normalized differences for various sequence identities. This

evaluation confirms the observations of the SPS-based analysis. Here, the scores of the tRNA reference alignments (red) are close to the ones of the optimal alignments for the whole range of sequence identities. The score differences of the optimal alignments and the non-tRNA reference alignments (yellow) increase with decreasing sequence identities. Together, the score differences of all RNA alignments show the same properties as the SPS-based analysis, with the biggest difference being around 45% identity. This further supports our conclusion that the BRaliBase dent is in fact not an artifact of the SPS or the curve smoothing factor but rather arises from the biased composition of the overall data set.

5.4. Concluding the dent

The BRaliBase dent, which for a decade has puzzled many colleagues in the field of comparative RNA structure prediction, can be explained by a bias in the benchmark data set, which holds *too much of the good* data in some conservation regions. The lesson learned from this experience is simple:

- When different data subsets in the benchmark show deviating overall characteristics, one should refrain from merging them into an overall performance diagram.
- When a benchmark holds data with diverging characteristics and a potential representation bias, it should provide means for sampling unbiased subsets.

In the case of BRaliBase, the bias was caused by too much of the best data dominating the performance analysis in low similarity regions, leading to a perceived decay of performance, *the dent*, in the consecutive 40% to 60% identity region. Quite understandably, this observation has attracted our attention on this region, and this feature of the benchmark data ended up being misdiagnosed as a localized weakness in the algorithms. This conclusion was seemingly confirmed by the apparent demonstration, in recent publications, that the dent could be overcome by innovative approaches. However, as explained above, this evidence turned out to be deceptive.

A new version of the BRaliBase benchmark, which draws from the increased knowledge about RNA families collected in the past decade, is clearly desirable. While our study can be taken as advice, marking a pitfall to avoid, the construction of a comprehensive new benchmark is a substantial contribution of its own. After Yann Ponty, one of the co-authors, presented the results of this study at a conference in the beginning of 2017, a group of volunteers has formed to tackle the task of creating an unbiased and up to date benchmark data set based on the current version of the *Rfam* database.

What we can do, on the other hand, is to prepare a new unbiased sampled data set that uses the current full BRaliBase as a start. As mentioned before, we do not want to allow any

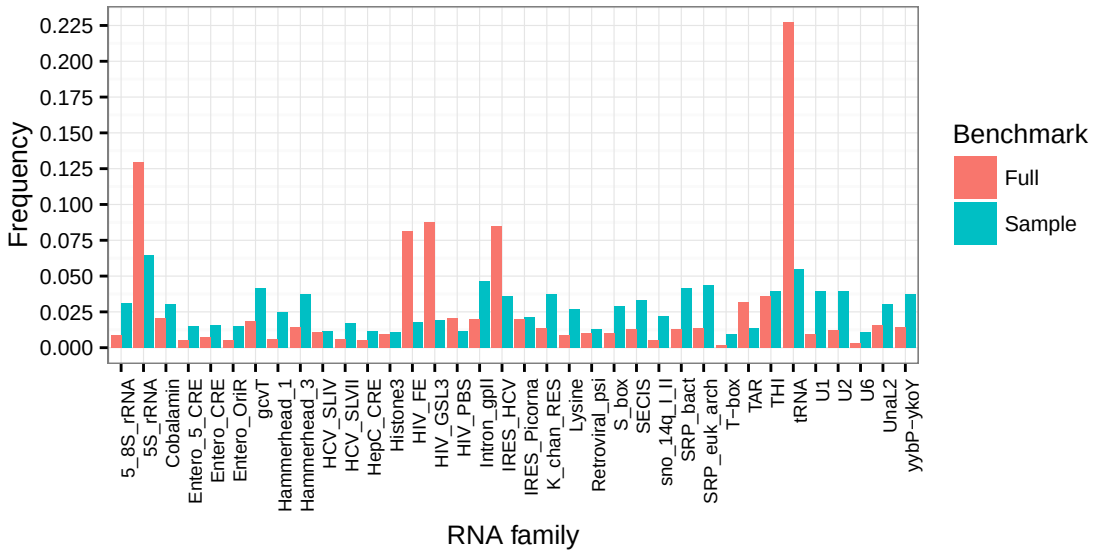


Figure 5.5.: Distribution of the various RNA families in the full and sampled benchmark data set. The full BRaliBase set holds 8,976 alignments while the sampled set only consists of 1,777.

RNA family to have a bigger share than 20% within each sequence identity region. We have shown that such a sampled data set prevents the formation of the dent in the performance measurements. Figure 5.5 shows each families share of the full BRaliBase and our sampled data set. It is apparent that the tRNA family now only accounts for 5% of the alignments, as compared to 22% before. In general, the shares of the families in the sampled data set are more equally distributed than previously in the full BRaliBase data set. This sampled data set could now also be used for the performance evaluation of other experiments, like the one in Chapter 6.

CHAPTER

6

SEED BASED DETECTION OF COMMON RNA MOTIFS

Contents

The previous attempts in Chapter 3 have shown that exact matching of RNA secondary structures for detecting common RNA motifs in viruses of various species is able to identify only a small number of motifs. Therefore, the next step, in order to identify common motifs, is to lift the restriction on the exact matching of secondary structures towards an approach that looks for smaller motifs (seeds) and scores their complexity in terms of structure, primary sequence, and co-occurrence in other species. Using this technique, our approach does not depend on one or multiple long structure matches anymore but can use multiple smaller structure matches and combine their score if they lie within close proximity in multiple genomes or sample sequences. Furthermore, using the seed approach, we are now able to incorporate primary structure conservation up to a user-definable degree.

In this chapter, we will first discuss the existing seed and extend procedures in Section 6.1. Next in Section 6.2, our approach for finding common RNA motifs in a potentially large group of samples will be described. After that, we will apply our approach to carefully selected test data sets in order to evaluate it. Last, we discuss the results of the evaluation in Section 6.4.

6.1. Sequence and structure matches using seed and extend

The comparison of similar sequences and structures as well as the search for a large number of samples or a specific sample against a large database has motivated the development of seed and extend strategies, which date back to the tools `BLAST` [3] and `FASTA` [67]. The general methodology of the tools using this approach is similar and can be divided into these steps [1]:

1. compute index values of short sequences called *seeds*;
2. find the highest scoring *chain* of non-overlapping seeds that have the same order in two or multiple samples;
3. use the seeds as *anchors* and compute the pairwise or multiple alignment between the anchors in the chain.

These strategies can be generally used to identify common patterns within sequences using quick index queries. Afterwards, the alignment process can be speed up by placing restrictions on the algorithm through the calculated optimal chains. The runtime of the complete approach is usually determined by the used alignment algorithm. And since classical hash functions for the seeds can be computed in linear time with respect to the input length, an optimal chain for two sequences of n seeds can be computed in $\mathcal{O}(n \log(n))$ [57].

Existing work in common motif discovery has already been presented in Section 2.3.3, but most of these approaches either need preannotated RNA family information or use a structural alignment that is too time consuming for a large number of input sequences. Compared to `CMfinder` and the other tools, we are maybe not able to improve the false positive rate for *de novo* screening, but we aim to significantly reduce the practical runtime by not using a full structural alignment. Our goal of an all against all comparison cannot be implemented using preannotated RNA families, because no specific motifs are known beforehand. The algorithms and tools that compare two or more structures on the sequence and structure level have been discussed in Chapter 5.

Bourgeade *et al.* introduced a program called `RNA-unchained` [15] that uses a method for comparing a query RNA to a previously hashed database of target RNAs. Part of this approach is a reversible hash function that encodes secondary structure as well as parts of the primary sequence into an integer. Small structural entities can be transformed into these integers and can then be matched against the hashed database. These matches will be used to compute an

optimal chain of seeds that can be applied as restriction to the complete sequence and structure alignment of multiple RNA sequences.

Even though this approach was designed for comparing one or multiple queries to a pre-processed database, we can use the general idea of the presented hash function in order to quickly identify structural entities that are shared among multiple samples, without having to use the whole framework of `RNA-unchained`.

6.2. Identifying RNA motifs using sequence and structure seeds

In this section, we will present the complete approach for the identification of common RNA motifs that uses the seed definition presented by Bourgeade *et al.* in [15]. In contrast to their approach, our version will not search for one specific query in a pre-processed database that holds all hash values of the seeds, but it will index all sequences of interest and search for common seeds within all other sequences in the database. Afterwards, windows in different sequences will be compared to each other in order to find windows that share a certain number of seeds. In the end, windows that share at least a predefined number of seeds of high complexity are grouped together and will form a potential group of common RNA motifs that will be given to the user for further investigation.

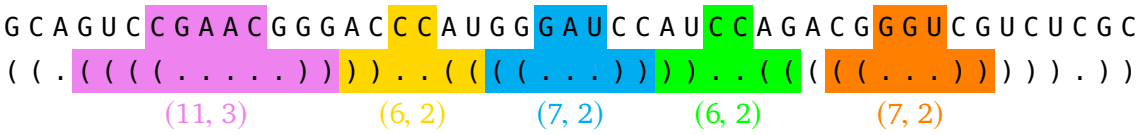
6.2.1. Seeds and their hash values

For finding common RNA motifs without *a priori* knowledge about the shape of the secondary structure and the base composition of the primary sequence, we first need to index a large number of motifs in order to efficiently match them afterwards. For this, we use the *(l,d)-centered seeds* introduced in [15] that allow us to represent the structure and sequence of an RNA segment using an integer value.

Definition 6.1 (adapted from [15])**.** Let R be an RNA sequence and S be a nested and well-formed secondary structure of R. Let l and d be two integers such that $2d \leq l$. The *(l,d)-centered-seed* at position $i \in \{0, \ldots, |S| - l\}$ is defined as the pair $(R^{i+d}_{i+l-1-d}, S^{i}_{i+l-1})$. Two (l,d)-centered seeds (U, V) and (X, Y) form a *match* if and only if $U = X$ and $V = Y$.

Using this seed definition, we are able to impose different length restrictions on the secondary structure and the primary sequence. The seeds can be seen as one structural motif of length l that does not allow any mismatch and one primary structure motif of length $l - 2d$. The latter is always centered such that the first and last d nucleotides are not part of the seed. Seed matches have two meaningful boundary cases: an $(l, \frac{l}{2})$ seed match is a purely structural match if l is an even number, while an $(l,0)$ match is an exact match in both, sequence as well as structure. This shows that (l,d)-centered-seed matches generalize exact matching and can become as abstract as pure structural matches without any sequence similarity.

Example 6.1. Given is one RNA sequence and the corresponding secondary structure. In this example, we highlight exemplary (l,d)-centered seeds with different parameters.



The graphic shows five different seeds with three different (l,d) parameter pairs. Here, the yellow seed and the green seed match while the blue seed and the orange seed do not match, because the nucleotides in the loop regions, `GAU` (blue) and `GGU` (orange), are different.

Definition 6.2 (adapted from [15])**.** Given an (l,d)-centered-seed (R, S), the hash function is defined as follows:

$$\underbrace{hash((R,S),l,d)}_{\text{hash value}} = \underbrace{4^{l-2d} \times \sum_{i=0}^{|S|-1} \left(encSS(S_i^i) \times 3^{l-1-i} \right)}_{\text{encode secondary structure}} + \underbrace{\sum_{i=0}^{|R|-1} \left(encNT(R_i^i) \times 4^{l-2d-1-i} \right)}_{\text{encode nucleotide sequence}}$$

The single characters of the structure string are encoded using the function *encSS*, and the single nucleotides are encoded using *encNT*:

$$encSS(x) = \begin{cases} 0 & x = \texttt{.} \\ 1 & x = \texttt{(} \\ 2 & x = \texttt{)} \end{cases} \qquad encNT(x) = \begin{cases} 0 & x = \texttt{A} \\ 1 & x = \texttt{C} \\ 2 & x = \texttt{G} \\ 3 & x = \texttt{T/U} \end{cases}$$

The two strings, the dot-bracket and the primary sequence, are jointly encoded into one integer value that is unique for any seed and combination of parameters l and d. If seeds from different combinations of l and d should be stored together, than the respective parameter combination needs to be stored too; otherwise, the uniqueness of the seeds cannot be retained. The conversion of the nucleotide sequence into a hash value is done by interpreting the sequence as a number in the quaternary system, because each nucleotide can be represented using two bits. Based on the complete nucleotide sequence of length $l - 2d$, the primary sequence is encoded as value in the interval $\{0, \ldots, 4^{l-2d} - 1\}$. The hash value of the dot-bracket string, on the other hand, can be generated by interpreting the structure string as a number in the ternary system. Since the complete hash value has to combine nucleotide and structure string, the multiplication of the hash value of the structure string with 4^{l-2d} is necessary.

In order to reduce the space requirement for this approach, it is not necessary to save the sequence and structure strings that have been used to compute the hashed seeds. Even though it is not mentioned in the original publication and since all hash values are unique for any

parameter combination, they can be decoded yielding the original sequence and dot-bracket strings.

Lemma 6.1. *Given the hash value h and the parameters l and d that were used to build the seed, the functions decodeSS and decodeNT correctly recreate the secondary structure as well as the nucleotide sequence.*

$$decodeSS(h) = \underbrace{\mathop{\Vert}_{i=0}^{l-1} decSS(S_i^i)}_{\text{digits to structure}}$$

$$\text{with } S = \underbrace{toStr(s)}_{\text{integer to string}} \mathbin{++} \underbrace{\left(\mathop{\Vert}_{i=0}^{l-|toStr(s)|} 0\right)}_{\text{string extension}}$$

$$\text{with } \underbrace{(s)_3 = \left(\left\lfloor \frac{h}{4^{l-2d}} \right\rfloor\right)_{10}}_{\text{decimal to ternary}}$$

$$decSS(x) = \begin{cases} . & x = 0 \\ (& x = 1 \\ ) & x = 2 \end{cases}$$

$$decodeNT(h) = \underbrace{\mathop{\Vert}_{i=0}^{l-2d-1} decNT(R_i^i)}_{\text{digits to nucleotides}}$$

$$\text{with } R = \underbrace{toStr(r)}_{\text{integer to string}} \mathbin{++} \underbrace{\left(\mathop{\Vert}_{i=0}^{l-2d-|toStr(r)|} 0\right)}_{\text{string extension}}$$

$$\text{with } \underbrace{(r)_4 = (h \bmod 4^{l-2d})_{10}}_{\text{decimal to quaternary}}$$

$$decNT(x) = \begin{cases} A & x = 0 \\ C & x = 1 \\ G & x = 2 \\ T/U & x = 3 \end{cases}$$

Proof. Based on Definition 6.2, we know that $h = 4^{l-2d}s + r$, where s and r are numerical representations of the structure and nucleotide string. Furthermore, we also know that $0 \leq r < 4^{l-2d}$ and s is an integer. Since $s = \frac{h-r}{4^{l-2d}}$, the fraction must result in an integer. We know the value of h; therefore, r can be calculated as $r = h \bmod 4^{l-2d}$. Following, $s = \lfloor \frac{h}{4^{l-2d}} \rfloor$, because the remainder r is removed in the numerator of the fraction.

The remaining steps are simple: for the structure string, we need to transform r into the ternary system; for the primary sequence, s needs to be transformed into the quaternary system. In some cases, the resulting transformed numbers must be extended by additional digits in order to yield the original strings. Assume that $(s)_{10} = 0_{10} = 0_3$ and $l = 7$. In this case, the ternary number must be extended by the remaining six zero digits, which symbolize the . for an unpaired base. The last step in both cases is the alphabet transformation, which is handled by the functions *decSS* and *decNT*, respectively. □

The decoder functions help us to regenerate the RNA sequence and structure motifs after common motifs have been identified by our approach. This helps our method to successfully eliminate the usage and storage of string data. Note that instead of combining the hash values for the nucleotide and structure string, we can also return and store them as pairs. In this case,

we need to perform two checks for identical integers during the matching process and would also need to add storage capacity for the data structure of the pair. In the end, this is a design choice and we continue to use one combined integer, because it can be easily decomposed into the two original hashes (see Lemma 6.1).

In total, there can be at most $3^l 4^{l-2d}$ different hash values depending on the choice for l and d. But how many actually realizable structures are there? This question has not been covered in the original publication of the seed approach. To simplify the analysis, we assume in the following that we examine only the secondary structure and no nucleotides of the primary sequence. We start with a well-formed RNA secondary structure S. As the concatenation of well-formed structures forms another well-formed structure, we do not have to care whether S holds one structure or many structures combined. Parenthesis in S are well balanced and S does not contain illegal motifs, i.e. no lonely base pairs and the loop of the hairpin motif has to have at least length three. We are interested in substrings of S that are considered irrespective of substructure boundaries, taken from all positions. These substrings do not necessarily represent a well-formed structure, but they always can be extended to the left or right to form a well-formed structure. And of course, they do not contain illegal motifs.

Which and how many different substrings of fixed length k can occur? Those that occur in a given set of structure strings are called *realized* k-patterns in S. On the other hand, those that can occur in any string S, we call *realizable* k-patterns. The number of realized k-patterns is an empirical question that we will address later. But first, we address the number of realizable k-patterns that can be described using the following grammar. For simplicity, we just specify the production rules; E is the axiom.

$$
\begin{aligned}
E &\rightarrow .A \mid)Z \mid (B \mid \epsilon \\
A &\rightarrow .A \mid)Y \mid (X \mid \epsilon \\
B &\rightarrow (B \mid .C \mid \epsilon \\
C &\rightarrow (X \mid .D \mid \epsilon \\
D &\rightarrow (X \mid .A \mid \epsilon \\
X &\rightarrow (B \mid \epsilon \\
Y &\rightarrow)Z \mid \epsilon \\
Z &\rightarrow .A \mid)Z \mid (X \mid \epsilon
\end{aligned}
$$

Nonterminals E, A, and Z represent the standard situation when the next structure character is not within a hairpin loop. They differ slightly, because start symbol E allows a lonely base pair character once in the beginning. This is necessary, because there might be another (or) character to the left of it that is not part of the substring. Lonely base pair characters at the right

end of the substring are possible, because the empty symbol can be read in any nonterminal; therefore, the string can end at any nonterminal. Nonterminals X and Y make sure that there are no lonely base pairs at other positions that are not the left or right end. Next, nonterminals B, C, and D make sure that the hairpin loop is at least of length three.

Now, the number of k-patterns is described by the following recurrences:

$$\begin{aligned}
E(n+1) &= A(n) + Z(n) + B(n) \\
A(n+1) &= A(n) + Y(n) + X(n) \\
B(n+1) &= B(n) + C(n) \\
C(n+1) &= X(n) + D(n) \\
D(n+1) &= X(n) + A(n) \\
X(n+1) &= B(n) \\
Y(n+1) &= Z(n) \\
Z(n+1) &= A(n) + Z(n) + X(n)
\end{aligned}$$

All recurrences of the form $E(0), \ldots, Z(0)$ have the result one.

Assume we choose $l = 10$ and $d = 5$ so that we do not take any primary sequence nucleotides into account when computing the hash value. The original publication expects at most $3^{10} = 59,049$ different structures. Using the previously defined recurrences, we can calculate $E(10)$ and see that there are $1,771$ realizable 10-patterns. This means that only $\approx 3\%$ of the hash values are actually be used. We performed an experiment with a random genome that contained $100,000$ nucleotides and were able to identify $1,578$ realized 10-patterns. With increasing genome size, the number of realized 10-patterns should approach the number of realizable 10-patterns. In this case, using an array data structure to store the hash values would result in lots of empty cells. Therefore, it would be better to use a hash table.

6.2.2. Creating database from hash values for seeds

Using the functions introduced in the previous section, we can now start by processing all sequences in the set I to calculate hash values for the seeds. The set I is the user input and consists of multiple RNA sequences that should be screened for common sequence and structure motifs. Therefore, the first step for producing seeds is to generate RNA secondary structures for all input sequences. In order to do so, we use the tool `RNALfold` [50] from the `ViennaRNA` package [68] for computing locally stable RNA structures. The tool uses a sliding window approach that is defined by the size of the windows w. Every substring of the RNA sequence with size smaller or equal to w is folded and all thermodynamically favorable local

Algorithm 6.1 Generating seeds from sequences and local RNA structures folded by `RNALfold`

```
1:  function GenerateSeeds(I, l, d, w)
2:      for R ∈ I do                                   ▷ all sequences in the input set
3:          for (S, i) ∈ RNALfold(R, w) do             ▷ set of all local folds S with position i
4:              for j ∈ {i, ..., i + |S| − l} do       ▷ all substructures of S with length l
5:                  if filter(S^j_{j+l−1}) ≥ threshold then   ▷ optional filter for structural complexity
6:                      h ← hash((R^{j+d}_{j+l−1−d}, S^j_{j+l−1}), l, d)
7:                      add pair (R, j) to Seeds[h]    ▷ set of all seeds with the same hash value
8:                      for k ∈ {j + l − w, ..., j} do
9:                          add h to Windows[(R, k)]   ▷ add seeds to all overlapping windows
10:     return (Windows, Seeds)
```

RNA structures, which are secondary structures with a free energy value below a threshold, are reported. The step size for the sliding windows approach in `RNALfold` is one so that no possible secondary structure can be missed. The complete folding runs in $\mathcal{O}(nw^2)$ time with n being the combined length of all RNA sequences in the set I.

Algorithm 6.1 depicts the strategy that is used to index all seeds into a database for the fast identification of common RNA motifs. First, the program `RNALfold` is called for every sequence in the input data set I separately. This call results in a set that is comprised of all locally stable RNA structures that are annotated with the respective start positions. Based on these information, we can now compute the seeds and their hash values by enumerating all possible $|S| - l$ substructures of structure S that have length l. The computation of the combined hash value might be sped up by storing the hash values of the previous nucleotide and structure substring. Assume that h_i is the hash value for the structure substring that starts at position i, then h_{i+1} can be computed as $(h_i \bmod 3^{l-1})3 + encSS(S_{i+l}^{i+l})$. This can be done in similar fashion for the nucleotide substrings. Furthermore, among the substructures might be several that only consist of unpaired bases, base pair open symbols, or base pair closing symbols, depending on the value chosen for l and the lengths of the stem or loop regions in the local structures. Therefore, our approach includes an optional filter in order to judge the complexity of the substructures. Here, we chose to only retain those substructures that contain at least one base pair open and one base pair closing symbol so that only motifs of hairpin loops or stem junctions are considered.

If a potential seed passes the filter, then its position and an identifier for the sequence itself are added to the set of all seeds with the same hash value. The table *Seeds* for all hash values can be either implemented as array, because $3^l 4^{l-2d} - 1$ represents the upper bound for the hash values, or as hash table if l is too large. Based on the results in the previous section, we have chosen the implementation as hash table. The table cells are implemented as hash multisets, because one seed might occur multiple times per cell.

Last, table *Windows* that holds windows of length w is filled with the hash value of a seed if the specific table cell encloses the complete seed. Assume that the seed starts at position $k = 537$, the window length is $w = 100$, and the length of the seed is $l = 11$. Then the hash value of the seed would be part of all windows, i.e. table cells, in the interval $\{448, \ldots, 537\}$. Note that the windows size that is used for generating the table *Windows* does not necessarily need to be the same as the parameter w, which is used to determine the window size of `RNALfold`.

The runtime of Algorithm 6.1 is $\mathcal{O}(nw^2 + m(w - l) + ml)$ with m being the combined length of all local RNA structures returned by `RNALfold`. The first addend is from `RNALfold` and the second one describes the time for inserting the computed seed value in $w - l$ windows. The time for computing the hash value is $\mathcal{O}(l)$ if we expect the compiler to detect the exponentiations in the two loops as loop invariant so that the exponentiations are computed and tabulated before entering the loops. Since there are at most m substructures, this computation can be done $\mathcal{O}(ml)$ time. Additionally, we assume that the term for the combined length of all locally stable RNA structures can be expressed as $m = nw$, because we expect that at most one local structure starts per sequence position. By using the faster method for computing consecutive seeds, we can reduce the runtime to $\mathcal{O}(nl + n(w - l)) = \mathcal{O}(nw)$, because the complete hash computation has to be done only once per locally stable structure and the hash values for the remaining $w - l$ substructures can be computed in constant time. In total, the term for the asymptotic worst-case runtime can be simplified to $\mathcal{O}(nw^2)$.

For the sake of a simple and short depiction of the algorithm, it does not cover the necessary computations for the reverse complement of every input sequence. The asymptotic runtime of this step or any following step is not affected by this, because the reverse complement only adds a constant factor of two to the combined length of all input sequences n.

For a detailed example of the computation of the hashed seed values see Table 6.1.

6.2.3. Filter windows for duplicates and by complexity

To avoid unnecessary comparisons between different windows in order to find common seeds, we can reduce the number of windows by eliminating those ones that hold the same information content. The second filtering step removes windows whose seed content does not show enough secondary structure complexity. One easy way to assess the complexity of the secondary structure of the seeds is to consider the number of nucleotides that form base pairs. Algorithm 6.2 filters the windows separately for each input sequence. Thereby, we assume that the list of windows for each sequence R is in ascending order regarding the position of the seeds. The algorithm needs to check whether the seeds of the current window at position k are a subset or equal to the seeds of the reference window at position *prev*. If this holds and both windows are in close proximity (distance smaller or equal to w), then the current window can be deleted, as it does not offer any new information. Otherwise, if the seeds of the window at position *prev* are a

Table 6.1.: The left column shows the selected seed of sequence R and local structures $S1$ as well as $S2$. The right column shows all database operations that have to be performed. The following parameters were chosen for this example: $w = 100$, $l = 12$, and $d = 4$. Furthermore, the filter function from Algorithm 6.1 only accepts secondary structures that contain both, opening and closing base pairs.

Seeds	Database operations
$\downarrow i = 10522$ **R** ...GCAUCAUA**UAGU**GUAAUGGACAGCACAACGGUCUUCUAAG... **S1** (**(((.....)**)))(((((.....))))) **S2** ((((.....))))	$hash(R^{10526}_{10529}, S1^{10522}_{10533}) = 65505739$ $windows[(R, 10434)] \uplus \{65505739\}$ ⋮ $windows[(R, 10522)] \uplus \{65505739\}$ $seeds[65505739] \uplus \{(R, 10434), \ldots, (R, 10522)\}$
$\downarrow i = 10523$ **R** ...GCAUCAUAU**AGUG**UAAUGGACAGCACAACGGUCUUCUAAG... **S1** ((**((.....)**))).(((((.....))))) **S2** ((((.....))))	$hash(R^{10527}_{10530}, S1^{10523}_{10534}) = 60468270$ $windows[(R, 10435)] \uplus \{60468270\}$ ⋮ $windows[(R, 10523)] \uplus \{60468270\}$ $seeds[60468270] \uplus \{(R, 10435), \ldots, (R, 10523)\}$
$\downarrow i = 10524$ **R** ...GCAUCAUAUA**GUGU**AAUGGACAGCACAACGGUCUUCUAAG... **S1** (((**(.....)**))).(((((.....))))) **S2** ((((.....))))	$hash(R^{10528}_{10531}, S1^{10524}_{10535}) = 45356475$ $windows[(R, 10436)] \uplus \{45356475\}$ ⋮ $windows[(R, 10524)] \uplus \{45356475\}$ $seeds[45356475] \uplus \{(R, 10436), \ldots, (R, 10524)\}$
$\downarrow i = 10539$ **R** ...GCAUCAUAUAGUGUAAUGGACAGCA**CAAC**GGUCUUCUAAG... **S1** ((((.....)))).(**((((.....)**)))) **S2** ((((.....))))	$hash(R^{10543}_{10546}, S1^{10539}_{10550}) = 67185217$ $windows[(R, 10451)] \uplus \{67185217\}$ ⋮ $windows[(R, 10539)] \uplus \{67185217\}$ $seeds[67185217] \uplus \{(R, 10451), \ldots, (R, 10539)\}$
$\downarrow i = 10540$ **R** ...GCAUCAUAUAGUGUAAUGGACAGCAC**AACG**GUCUUCUAAG... **S1** ((((.....)))).((**(((.....)**))) **S2** ((((.....))))	$hash(R^{10544}_{10547}, S1^{10540}_{10551}) = 65507078$ $windows[(R, 10452)] \uplus \{65507078\}$ ⋮ $windows[(R, 10540)] \uplus \{65507078\}$ $seeds[65507078] \uplus \{(R, 10452), \ldots, (R, 10540)\}$
$\downarrow i = 10541$ **R** ...GCAUCAUAUAGUGUAAUGGACAGCACA**ACGG**UCUUCUAAG... **S1** ((((.....)))).(((**((.....)**))) **S2** ((((.....))))	$hash(R^{10545}_{10548}, S1^{10541}_{10552}) = 60472858$ $windows[(R, 10453)] \uplus \{60472858\}$ ⋮ $windows[(R, 10541)] \uplus \{60472858\}$ $seeds[60472858] \uplus \{(R, 10453), \ldots, (R, 10541)\}$
$\downarrow i = 10542$ **R** ...GCAUCAUAUAGUGUAAUGGACAGCACAA**CGGU**CUUCUAAG... **S1** ((((.....)))).((((**(.....)**)))) **S2** ((((.....))))	$hash(R^{10546}_{10549}, S1^{10542}_{10553}) = 45370219$ $windows[(R, 10454)] \uplus \{45370219\}$ ⋮ $windows[(R, 10542)] \uplus \{45370219\}$ $seeds[45370219] \uplus \{(R, 10454), \ldots, (R, 10542)\}$
$\downarrow i = 10529$ **R** ...GCAUCAUAUAGUGUA**AUGG**ACAGCACAACGGUCUUCUAAG... **S1** ((((.....)))).(((((.....))))) **S2** (**(((.....)**)))	$hash(R^{10533}_{10536}, S2^{10529}_{10540}) = 65505594$ $windows[(R, 10441)] \uplus \{65505594\}$ ⋮ $windows[(R, 10529)] \uplus \{65505594\}$ $seeds[65505594] \uplus \{(R, 10441), \ldots, (R, 10529)\}$
$\downarrow i = 10530$ **R** ...GCAUCAUAUAGUGUAA**UGGA**CAGCACAACGGUCUUCUAAG... **S1** ((((.....)))).(((((.....))))) **S2** ((**((.....)**))	$hash(R^{10534}_{10537}, S2^{10530}_{10541}) = 60468456$ $windows[(R, 10442)] \uplus \{60468456\}$ ⋮ $windows[(R, 10530)] \uplus \{60468456\}$ $seeds[60468456] \uplus \{(R, 10442), \ldots, (R, 10530)\}$
$\downarrow i = 10531$ **R** ...GCAUCAUAUAGUGUAAU**GGAC**AGCACAACGGUCUUCUAAG... **S1** ((((.....)))).(((((.....))))) **S2** (((**(.....)**)))	$hash(R^{10535}_{10538}, S2^{10531}_{10542}) = 45356449$ $windows[(R, 10443)] \uplus \{45356449\}$ ⋮ $windows[(R, 10531)] \uplus \{45356449\}$ $seeds[45356449] \uplus \{(R, 10443), \ldots, (R, 10531)\}$

Algorithm 6.2 Filter windows with the same seed content per sequence and for low complexity

```
1:  function FilterWindows(Windows, I, w, Seeds)
2:      for R ∈ I do                                   ▷ all sequences in the input set
3:          create table WindowsR from Windows with only windows from sequence R
4:          prev ← −w − 1
5:          for (R, k) ∈ WindowsR do                   ▷ assume table is ordered by position
6:              if filter(Windows[(R, k)]) < threshold then   ▷ optional filter for seed complexity
7:                  delete cell (R, k) from table Windows
8:              else if k − prev > w then              ▷ windows at position k and prev are distant
9:                  prev ← k
10:             else if Windows[(R, k)] ⊆ Windows[(R, prev)] then   ▷ seeds in k are part of k
11:                 delete cell (R, k) from table Windows
12:             else if Windows[(R, prev)] ⊂ Windows[(R, k)] then   ▷ seeds in prev are part of k
13:                 delete cell (R, prev) from table Windows
14:                 prev ← k
15:     if necessary regenerate Seeds
16:     return (Windows, Seeds)
```

subset of the seeds at position k, then the former one is deleted and the window at position k becomes the new reference window.

The complete filtering can be done in $\mathcal{O}(nw)$ time, because there are at most n windows with w seeds each if we assume that at most one local RNA structure starts per sequence position. The integer indices of the hash table *Windows* can be sorted in linear time using counting sort. The computation of the seed set intersection can be done in linear time $\mathcal{O}(w)$, because the seeds in the table cells are implemented as hash multisets.

After removing the unwanted windows, table *Seeds* has to be regenerated accordingly. This can also be done in $\mathcal{O}(nw)$ time.

6.2.4. Identification of common motifs in windows

The first step for the identification of common RNA motifs that are shared among multiple windows in different input sequences is the pairwise comparison of these windows regarding their seed content. For the identification, the windows act as smallest units that gather all structural RNA data within their neighborhood that expands w nucleotides. The objective for the use of pairwise comparisons is to circumvent combinatorial explosion; otherwise, the algorithm would need to consider 2^n different window combinations in order to find seeds that are shared among a certain number of them.

The procedure for the pairwise intersection of seeds that are grouped in windows is depicted in Algorithm 6.3. In order to compare only windows that share at least one seed, the algorithm only performs the multiset intersection between two windows if they occur together in one cell

Algorithm 6.3 Pairwise intersection of seed windows

```
1: function WindowIntersection(Seeds, Windows, w)
2:     for h ∈ Seeds do                          ▷ all different seeds represented by their hash values
3:         for (R, k) ∈ Seeds[h] do  ▷ Seeds[h] ordered ascending by sequence ID and position
4:             for (R′, k′) ∈ Seeds[h] such that R′ ≥ R and k′ > k do
5:                 if ((R, k), (R′, k′)) ∉ Intersec and (R ≠ R′ or k′ − k > w) then
6:                     Intersec[((R, k), (R′, k′))] ← Windows[(R, k)] ∩ Windows[(R′, k′)]
7:     return Intersec
```

of the *Seeds* table. We assume that the windows in the cells of table *Seeds* are ordered ascending by the sequence identifier and second by the position. In this way, the current window (R, k) is only compared to the other windows (R', k') that occur later in the sorted list. Furthermore, only windows that are from different sequences or whose distance within the same sequence is greater than w are compared. If two windows share multiple seeds, then the intersection will not be computed again. Instead of performing the intersection operation once per window pair, the algorithm could also add the seeds step by step to the intersection table. We have chosen the first solution, because the practical performance is much better.

The asymptotic runtime of the version presented in Algorithm 6.3 is $\mathcal{O}(n^2 w)$. In the worst-case, there are w seeds that are shared among all n windows if we again assume that at most one local RNA structure starts per sequence position. This still results in at most n^2 comparisons, because the intersection of two windows is never computed more than once. The comparison using the intersection operation can be done in linear time $\mathcal{O}(w)$, due to the implementation as hashed multiset.

Similar to the filtering of seeds in the previous section, the pairwise window intersections can also be filtered for proximal intersections with the same seed content and the complexity of the common seeds. This can be done similarly by searching for pairs of intersections $((R, i), (R', i'))$ and $((R, j), (R', j'))$ with $|i - j| \leq w$ and $|i' - j'| \leq w$. If $Intersec[((R, i), (R', i'))]$ is a subset of $Intersec[((R, j), (R', j'))]$, then it can be deleted and the same holds for the opposite.

6.2.5. Combining the common seeds among all windows

After creating pairwise intersections of windows, we need to combine the knowledge of all these intersections to groups of multiple windows that share common seeds. For this purpose, we use an undirected graph structure in which windows are represented by vertices. These vertices are connected by edges that are annotated with all seeds that are shared between these two regions. Since we use an undirected graph, assume that the edges (a, b) and (b, a) denote the same connection between windows a and b.

Algorithm 6.4 depicts the complete process for the identification of windows that share common RNA motifs. The function has the objective to transfer all pairwise intersections in

Algorithm 6.4 Find common seeds among all windows using graph structures

function CombineIntersections(*Intersec*, w)
  initialize graph G with vertices V and edges E
  for $((R, k), (R', k')) \in$ *Intersec* **do** ▷ all intersections after filtering
    if $((R, l), (R', l')) \in E$ with $k - \frac{w}{2} \leq l \leq k + \frac{w}{2}$ **and** $k' - \frac{w}{2} \leq l' \leq k' + \frac{w}{2}$ **then**
      add seeds in *Intersec*$[((R, k), (R', k'))]$ to edge $E[((R, l), (R', l'))]$
    else
      $m \leftarrow k$
      $max_m \leftarrow 0$
      for $(R, l) \in V$ with $k - \frac{w}{2} \leq l \leq k + \frac{w}{2}$ **do** ▷ find proximal position l
        $tmp \leftarrow \infty$
        for $((R, l), (R'', l')) \in E$ **do** ▷ all edges that include (R, l)
          $tmp \leftarrow \min(tmp, |Intersec[((R, k), (R', k'))] \cap E[((R, l), (R'', l'))]|)$
        if $tmp \geq thres$ **and** $tmp > max_m$ **then** ▷ check if position l has similar seeds
          $m \leftarrow l$
          $max_m \leftarrow tmp$
      $m' \leftarrow k'$
      $max_{m'} \leftarrow 0$
      for $(R', l') \in V$ with $k' - \frac{w}{2} \leq l' \leq k' + \frac{w}{2}$ **do** ▷ find proximal position l'
        $tmp \leftarrow \infty$
        for $((R'', l), (R', l')) \in E$ **do** ▷ all edges that include (R, l')
          $tmp \leftarrow \min(tmp, |Intersec[((R, k), (R', k'))] \cap E[((R'', l), (R', l'))]|)$
        if $tmp \geq thres$ **and** $tmp > max_{m'}$ **then** ▷ check if position l' has similar seeds
          $m' \leftarrow l'$
          $max_{m'} \leftarrow tmp$
      add seeds in *Intersec*$[((R, k), (R', k'))]$ to edge $E[((R, m), (R', m'))]$
  ConnComp $\leftarrow$ compute all connected components of G
  return *ConnComp* sorted by the structural complexity of the seeds in each component

the table *Intersec* into the graph G. While processing these pairwise intersections two cases can happen: first, there are already two windows connected by an edge in G that are in close proximity to (R, l) and (R', l'), respectively. In this case, the algorithm adds all seeds shared between the two windows to the annotation of the already existing edge. We chose this approach, because otherwise shared seeds among two regions might be scattered between different edges, even though both describe a connection between the same two regions.

Second, no such edge between two windows that are close to (R, l) and (R', l') exists. Then, the two windows will be handled separately. The algorithm looks for windows that already exist as vertices and are proximal to either (R, l) or (R', l'). Following, all edges that include these proximal windows are investigated whether their annotations share a number of seeds with the intersection of (R, l) and (R', l') that is greater than a threshold. If this holds, then these proximal windows might be candidates that are similar to the current windows. The

overall objective is to find proximal windows for (R, l) and (R', l') whose edge annotations show the closest resemblance to the seeds that are shared among (R, l) and (R', l'). The algorithm performs this search so that seeds shared between two regions are not scattered among different edges between proximal windows. But if there are multiple proximal windows, then the algorithm chooses the one whose seeds in the edge annotation are most similar to the current ones. On the other hand, if no such proximal windows exist, then new vertices (R, l) and (R', l') are introduced and an edge between them is formed.

Once the graph structure is completed, the search for connections between multiple windows can begin. In order to do that, the approach depicted in Algorithm 6.4 uses *connected components*. A connected component is a subgraph of a graph in which any two vertices are connected by a path. Every connected component denotes a group of windows of multiple sequences that are connected to each other due to similar seed content. But by using connected components, the algorithm does not expect all windows to have pairwise connections. We have chosen this relaxed view on window connections, because there might be some intersections that might have been filtered out in the previous processing steps. But one has to be careful using the results based on connected components, because by grouping all windows in a connected component, we are assuming transitivity. This means that if windows a and b as well as a and c are connected by an edge, then by using connected components, we assume that b and c are also related somehow, but we have not been able to find this relationship yet. If we expect pairwise edges between all windows that are grouped together, then we can use cliques instead of connected components. A *clique* is a subgraph of a graph in which any two vertices are connected by an edge. A clique is *maximal* if it can not be extended by an adjacent vertex. But we have to be aware that in contrast to connected components, one window might be part of multiple cliques, which increases the number of groups that will be given as output to the user. This is especially noteworthy, because there are at most n connected components but $3^{\frac{n}{3}}$ maximal cliques [78]. In the end, we have chosen to use connected components, also in order to reduce the output given to the user, but the algorithm is built in a modular way so that the search for connected components can be replaced by the search for cliques in the graph.

The asymptotic runtime of Algorithm 6.4 can be determined by discussing the algorithm step by step. In the worst-case, the algorithm needs to add n^2 different intersections to graph G if the filtering steps were not successful. Next, it needs to check w neighbor nodes that might have $n - 1$ outgoing edges if G is complete. The intersection between the set of seeds shared by (R, l) and (R', l') as well as the edge annotation set can be computed in linear time $\mathcal{O}(w)$, because they are implemented as hash sets and we assume again that at most one local RNA structure starts per sequence position. This leaves us with an asymptotic runtime of $\mathcal{O}(n^3w^2)$ until this point.

Last, we need to consider the computation of connected components, which can be done in $\mathcal{O}(|V| + |E|)$ by looping through all vertices and starting a depth-first search whenever a

Algorithm 6.5 The main algorithm that invokes all functions for the intermediate steps

1: **function** FINDGROUPSOFCOMMONMOTIFS(I, l, d, w)
2: ($Windows$, $Seeds$) ← GENERATESEEDS(I, l, d, w)
3: ($Windows$, $Seeds$) ← FILTERWINDOWS($Windows$, I, w, $Seeds$)
4: $Intersec$ ← WINDOWINTERSECTION($Seeds$, $Windows$, w)
5: $components$ ← COMBINEINTERSECTIONS($Intersec$, w)
6: **return** *components* sorted and annotated with all seeds present as annotations of all edges that are part of this component

new vertex that is not yet part of a connected component has been found [52]. On the other hand, if we want to identify maximal cliques instead of connected components, the runtime is exponential ($\mathcal{O}(3^{\frac{n}{3}})$) for this NP-complete problem [61, 78]. So, the overall asymptotic complexity of Algorithm 6.4 is $\mathcal{O}(n^3w^2)$ in case we use connected components and $\mathcal{O}(3^{\frac{n}{3}} + n^3w^2)$ if we decide to use maximal cliques instead.

6.2.6. Summary: combining all steps

In the previous sections, all steps for computing groups of segments from different input sequences have been discussed in detail. Here, we want to summarize the steps and show in which order they are performed. The overall procedure, which is depicted in Algorithm 6.5, expects four input parameters: the set of input sequences I, the length l of the structure part of the seeds, the number of nucleotides d that are excluded from the seed from the 5′ and 3′ direction, and the window length w.

The first step is the generation of seeds based on all sequences in I and all locally stable RNA structure folds using `RNALfold`. This step results in two data structures: *Windows* and *Seeds*. The former is a table of all windows of size w from all sequences in which each cell holds all hashed seed values that are included in this window. The latter, on the other hand, is a table of all hashed seed values in which each cell holds all windows that contain the corresponding seed. Next, table *Windows* might be optionally filtered for windows that are located close to each other and have similar seed content, and for windows whose seeds show low secondary structure complexity. After this filtering step, pairwise comparisons of windows based on their seed content are performed and the results of these intersections are saved in table *Intersec*. Following, also this table might be filtered based on similar criteria as before. Last, the data gathered in *Intersec* are combined using a graph structure that represents windows as vertices, and seeds that are shared among windows are represented by edge annotations. The connected components of this graph show related sequence segments from multiple sequences that share several seeds; therefore, their local secondary structure and primary sequence content is expected to be similar. The components will be ranked by the complexity of the secondary structure of the seeds that are part of a component. The number of base pairs can be one

measure for this such that a higher number of pairs results in a higher score.

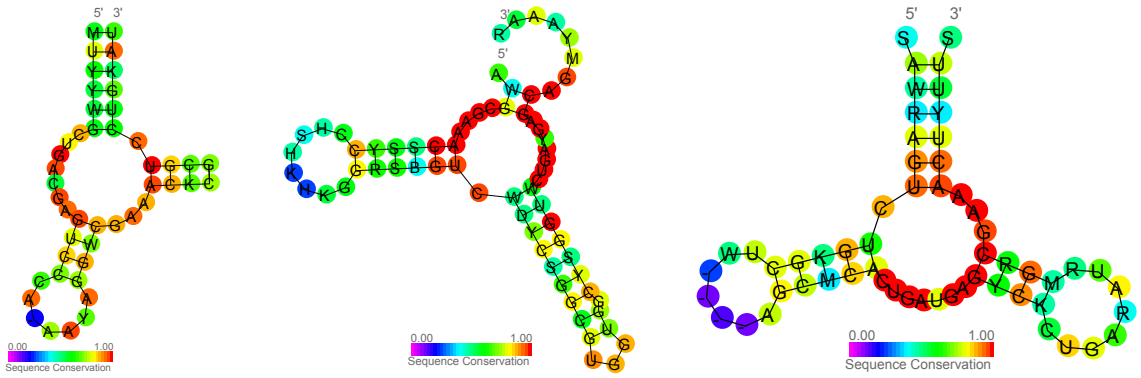
Following, a summary of the asymptotic runtimes of the intermediate steps: Algorithm 6.1 for generating the seeds needs $\mathcal{O}(nw^2)$, Algorithm 6.2 for filtering the windows of seeds runs in $\mathcal{O}(nw)$, Algorithm 6.3 for pairwise window intersections takes $\mathcal{O}(n^2w)$, and Algorithm 6.4 for grouping windows using a graph structure runs in $\mathcal{O}(n^3w^2)$ (connected components) or $\mathcal{O}(3^{\frac{n}{3}} + n^3w^2)$ (maximal cliques). This illustrates that the overall runtime of the combined steps, as shown in Algorithm 6.5, is determined by the last step that combines all windows into a graph structure.

6.3. Application of the algorithm

In this section, we apply the algorithm to two data sets that use different sequences from the hammerhead ribozyme I-III families as example. Usually, when the identification of new RNA family members is the task, we use specific tools that are tailor made for the molecule of question. But we want to answer the question whether our approach can identify these members without any *a priori* knowledge on their nature.

6.3.1. Seed sequences of the hammerhead ribozyme families

In 1986, the hammerhead ribozyme was the third reported catalytic RNA after the discovery of the Group I intron and the RNAse P [54, 88]. It catalyzes the scission of its own backbone without co-factors and has a length of 50 to 70 nucleotides. It was first identified in a group of plant viruses, virus satellites, and viroids. Since then, the hammerhead ribozyme was identified in a variety of organisms from bacterial to eukaryotic genomes [85], including humans [84]. The hammerhead ribozyme is often used as a model and prototype ribozyme, because it exists in three different forms that all share a catalytical center of 15 highly conserved nucleotides. This center is surrounded by three different hairpin loops that are absolutely required for catalysis and whose graphical representation resembles the shape of a hammerhead shark head [86]. The hammerhead ribozymes can be classified into three types based on which helix the $5'$ and $3'$ ends are located. If the $5'$ and $3'$ ends of the sequence lie in stem I, then it is a hammerhead ribozyme of type I. The classifications for the other two types are accordingly. Figure 6.1 shows the secondary structure representations of all three hammerhead types. The nucleotides in this graphical representation are based on the consensus of the seed alignments from the family models in *Rfam*. It is apparent that the most conserved nucleotides of the structures are in the multiloop region, which represents the catalytic center of the molecule. The one or two hairpin loops per structure are less conserved over all sequences of the seed alignment. *Rfam* seed alignments consist of manually selected sequences and manually curated family alignments. Therefore, we can be confident that a human expert agreed with the alignment and that it

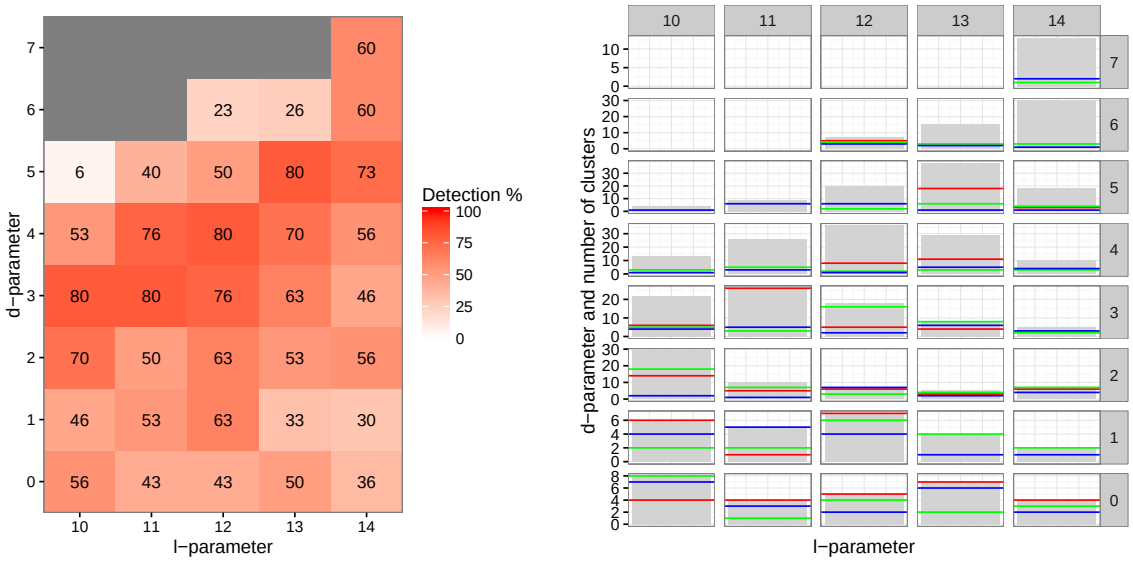


(a) Type I **(b)** Type II **(c)** Type III

Figure 6.1.: Secondary structure representation of all three hammerhead ribozyme types. The color code shows the sequence conservation in the alignment of all seed sequences.

shows the structure relevant for the biological function. The seed alignment for hammerhead ribozyme I consists of 29 sequences from viruses and eukaryotes; the alignment for type II consists of 24 sequences that are abundant in eubacteria; the alignment for type III consists of 82 sequences that can be found often in plant viroids and other satellite RNA species. The hammerhead ribozyme is a good testing choice, because we can find out whether the approach can discriminate between the three similar types. Furthermore, the molecule has a conserved primary sequence core that usually lies in the unpaired multiloop region, so we can test the benefit of using parts of the primary sequence to increase the specificity compared to pure secondary structure matching.

For our experiment, we sample a random genome of length 50,000 nucleotides, which is similar to most viral genomes, from a uniform distribution of the four nucleotides. In each genome, we insert ten hammerhead ribozymes of all three types from the seed data set at random positions. If we assume that the hammerhead ribozyme has an average length of 60 nucleotides, then we can say that $\approx 3.5\%$ of the random genome is covered by the hammerheads. For the identification of clusters, we use the approach that was described in the previous section. The pairwise intersections of the windows are added into a graph by connecting all windows, which are represented as nodes, with edges that are annotated with the seeds that are shared among them if they are within the intersection table. Subsequently, all connected components of the graph are identified; we will refer to them as clusters from now on. The clusters are scored by summing up all scores of the edges that are part of the connected component. Similarly, the edge score is the sum of the scores of all seeds that are shared among the two windows that are connected by this edge. For the seed score, we use a simple model that scores each paired nucleotide and all $l - 2d$ shared primary sequence nucleotides with one. For a more sophisticated scoring model of the two windows, we would need to perform a time consuming



(a) Detection rates all hammerhead types

(b) Number of clusters

Figure 6.2.: Results of the experiment for all three hammerhead types with various l and d parameters. (a) shows the detection rates in % for all hammerhead types combined. (b) shows the number of clusters that have been identified during each experiment. The red line shows the position of the hammerhead type I cluster; the green line shows the position of the hammerhead type II cluster; the blue line shows the position of the hammerhead type III cluster. Whenever a line of a specific color does not exist, it means that no cluster of this hammerhead type has been found for the specific parameter combination.

alignment, which is exactly what we wanted to avoid in the first place. The windows size for this experiment is 100 so that the approach is able to cover the full length of the molecule.

In order to obtain reproducible results, we repeated the experiment for all l and d parameter combinations ten times and display the median results. For each run, we created a new random genome and chose ten new sequences from the *Rfam* seed alignment. It is not displayed here, but the variance between the repeated experiments with the same parameter set is minimal and varies by at most two hammerheads that have been or not have been identified in the other experiments compared to the median result. Figure 6.2a illustrates the detection rates of this experiment for all three hammerhead types combined. The experiment was conducted with seed lengths from 10 to 14 nucleotides, because this length should cover most of the hairpin loops and the catalytic center of the molecule. Furthermore, we repeated the experiment for all possible d values, such that $0 \leq d \leq \frac{l}{2}$. Using a greater d value increases primary sequence specificity. We mark a hammerhead occurrence in the random genome as detected if a window, whose position significantly overlaps with the hammerhead in the random genome, is found within a cluster of hammerhead windows of the same type. The false positive rate of a cluster

of hammerheads is not allowed to be greater than 50%. Obviously, whenever we search for unknown RNA motifs, there is no way to infer the false positive rate beforehand. In this case, we use this restriction to infer the number of pairwise intersections that have to be considered and added into the graph to achieve an acceptable detection rate coupled with a low false positive rate. These findings can later be used in real experiments with completely unknown motifs.

In this experiment, we could observe that only hammerheads of the same type were grouped into one cluster. This shows that the approach is able to distinguish between the different secondary structures of the hammerhead types, as shown in Figure 6.1. In the best case, we are able to identify 80% of the 30 inserted hammerhead sequences. The best detection rates can be achieved by optimizing the d parameter such that it takes three to four nucleotides of the primary sequence into account. One might expect to see the highest detection rates with only small restrictions on the primary sequence, but we need to consider the relation between low sequence specificity and false positive rate. If we use a large d value, then we only look at the secondary structure. This might lead to a large number of random matches considering that $l \leq 14$. That is why the detection rate presented in Figure 6.2a considers only clusters with a maximal false positive rate of 50% so that the majority of cluster members are windows overlapping with the hammerhead occurrences.

Figure 6.2b shows the total number of clusters, i.e. connected components, that were identified using each parameter combination. We have marked the positions of the clusters for all three hammerhead types in the ordered result list. The clusters are ordered according to their score starting with the highest one at position one. As expected, with higher primary sequence specificity, the total number of identified clusters shows a decreasing tendency. Furthermore, we observe that for the parameter combinations that show the highest detection rates, which are (13, 5), (12, 4), (11, 3), (10, 3), the hammerhead clusters are among the top-ranked ones. But it is noticeable among these clusters that the hammerhead ribozymes of type I (red line) often ranks worse than the clusters of the other two types.

In Figure 6.3, we show an evaluation of the cluster size growth with increasing numbers of pairwise intersections that are added into the graph structure. Furthermore, this plot illustrates the relationship between increasing cluster size, detection rate, and false positive rate. For most parameter combinations, we can observe that the hammerhead types II and III contribute more to the overall detection rate than type I. This can be explained by the fact that the consensus secondary structure of type I only contains one hairpin loop (see Figure 6.1a), as compared to the two hairpin loops in types II and III (see Figures 6.1b and 6.1c). The additional hairpin loop enables the approach to find more shared seeds that increase the chance for a higher cluster score.

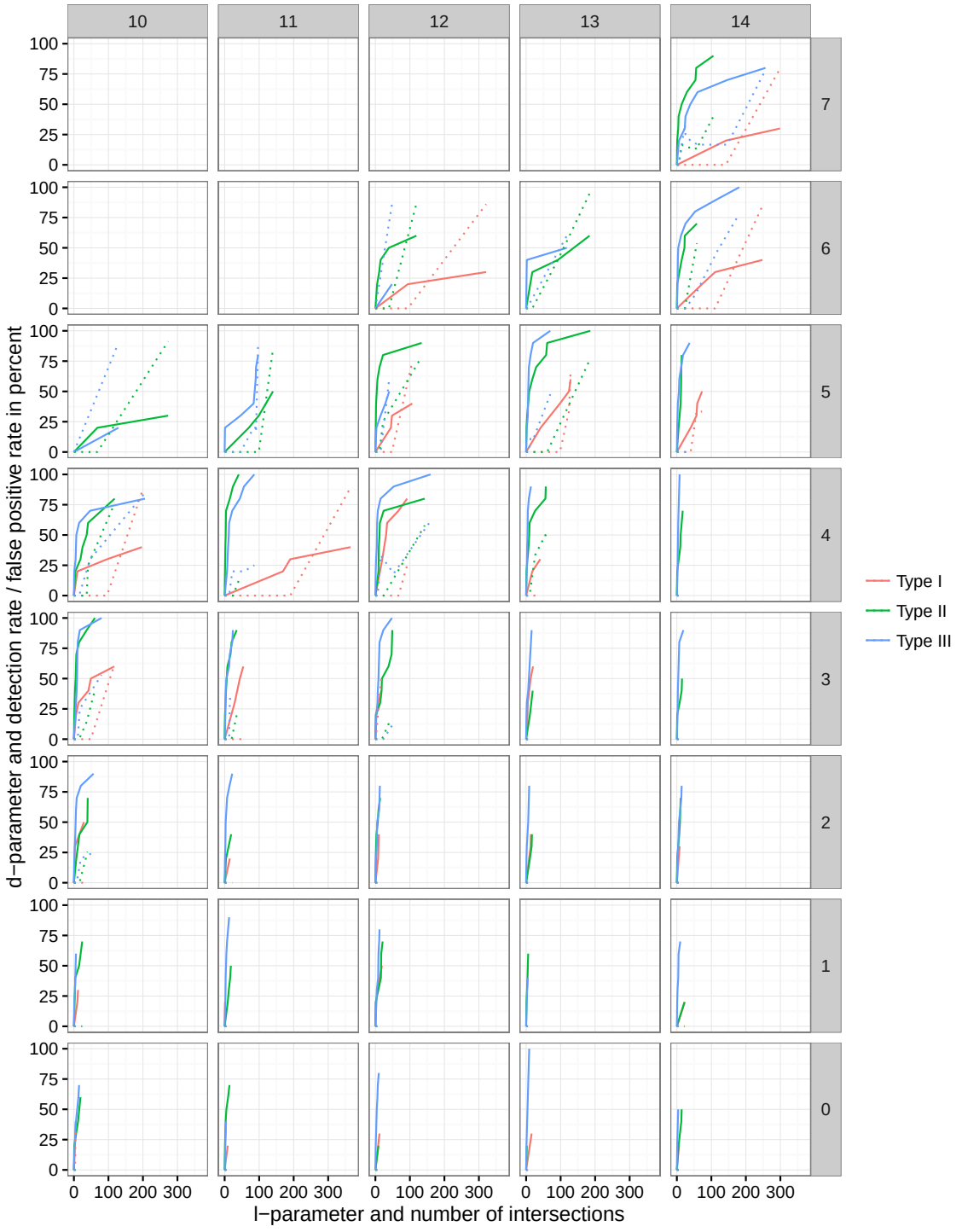
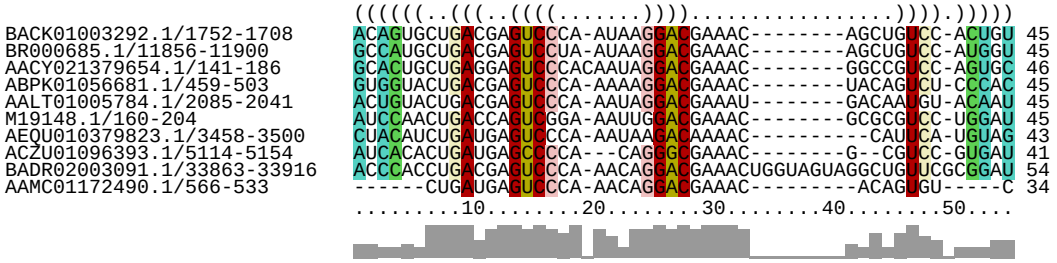
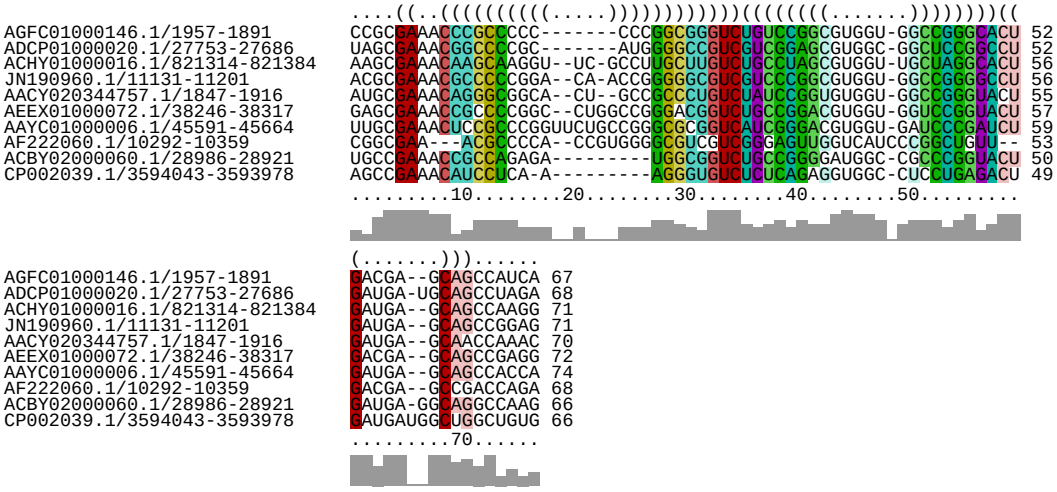


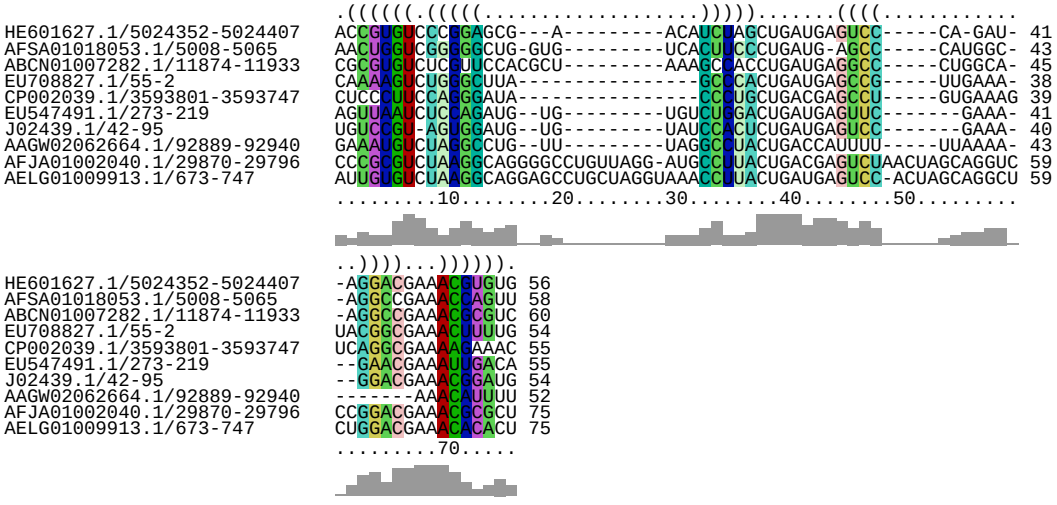
Figure 6.3.: This plot shows the experiments for all three hammerhead types with various l and d parameters. The x-axis on each plot shows the number of pairwise intersections that were added into the graph and the y-axis shows the detection rate for the solid lines and the false positive rate for the dotted lines.



(a) Hammerhead ribozyme type I



(b) Hammerhead ribozyme type II



(c) Hammerhead ribozyme type III

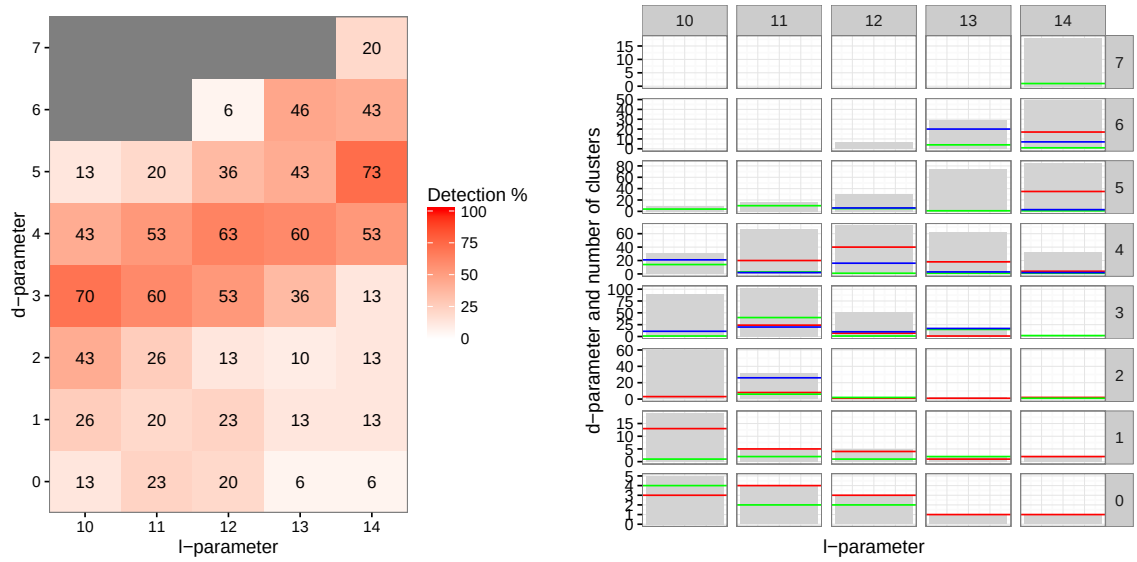
Figure 6.4.: The results of the sequence and structure alignments using LocARNA for the three hammerhead types. Each alignment consists of ten taxonomically distant sequences representing the same hammerhead type.

6.3.2. Taxonomically distant sequences of the hammerhead family

With the next step, we want to increase the difficulty by not using the sequences from the seed alignment of the families but rather by using hammerhead sequences from taxonomically distant organisms. For this, we considered all sequences in the *Rfam* database that were assigned to one of three hammerhead families: 55,175 sequences for type I, 285 for type II, and 1,590 sequences for type III. Next, we chose ten taxonomically distant sequences for each type that show less primary sequence conservation than the sequences from the seed alignment. Figure 6.4 shows the sequence and structure alignment for all ten sequences of a hammerhead type. The sequences of type I of taxonomically distant organisms still show high primary sequence conservation, but they only form one small hairpin loop with a conserved loop sequence. The sequences of type II, on the other hand, have a highly conserved primary sequence in the catalytic center, which seems to be in a stem junction region according to the structure prediction of `LocARNA`. The first hairpin loop seems to be present in only half of the sequences while the other two hairpin loops seem to be more conserved among all sequences. Last, the consensus structure of the sequences of type III contains two hairpin loops with large loop areas that are embedded into a multiloop structure. Here, it is apparent that both hairpin loop regions show low sequence conservation while the conserved catalytic center lies in the unpaired part of the stem junction.

We repeated the experiment performed in the previous section. This time, we only create a new random genome for every experimental run, but the chosen hammerhead sequences will stay the same. In addition to the fact that we have used less conserved primary sequences now, the size of the random genome increased to 100,000 nucleotides so that we can expect more random matches. The 100,000 nucleotides equal the size of four to five viral genomes, but in the end, it does not matter whether we analyze one large genome or multiple smaller ones. With an average size of 60 nucleotides per hammerhead sequence, $\approx 1.8\%$ of the random genome is covered by hammerhead sequences of types I to III. Figure 6.5a illustrates the detection rates of the new experiment for all three hammerhead types combined. In the new experiment, we are able to identify 73% of the 30 inserted hammerhead sequences in the best case. In general, the detection rate is lower compared to the previous experiment. Also, we can detect that setting d to a value that takes three to four nucleotides of the primary sequence into account promises the best values for all choices of l. Furthermore, using no primary sequence information in the seeds leads to a decreased detection rate due to a higher number of false positive results.

As before, Figure 6.5b illustrates the total number of connected components that were identified using each parameter combination. In comparison to the previous experiment, the total number of clusters has generally increased. But, on the other hand, one would expect that parameter combinations with a high d value, i.e. low primary sequence specificity, would also result in a greater number of clusters. When we compare the number of clusters of the



(a) Detection rates all hammerhead types **(b)** Number of clusters

Figure 6.5.: Results of the experiment for all three hammerhead types with various l and d parameters. (a) shows the detection rates in % for all hammerhead types combined. (b) shows the number of clusters that have been identified during each experiment. The red line shows the position of the hammerhead type I cluster; the green line shows the position of the hammerhead type II cluster; the blue line shows the position of the hammerhead type III cluster. Whenever a line of a specific color does not exist, it means that no cluster of this hammerhead type has been found for the specific parameter combination.

parameter combinations (14, 7) and (14, 5), we see that the latter results in a greater number of clusters. This can be explained by the idea that up until a certain point, the insertion of further pairwise intersection results leads to the creation of new nodes and edges in the graph, which increases the number of connected components. Once this point has been reached, new window intersections do not lead to the creation of new nodes in the graph anymore, because there already exists a node in the graph that represents a window that is in close vicinity to the window that should be inserted. Instead of creating a new node, the already existing node is used and becomes the end point of an additional edge that is inserted. But the insertion of an edge into an undirected graph with high connectedness, usually will not increase the number of connected components; it decreases or remains unchanged.

Figure 6.6 illustrates the detection and false positive rates of the clusters of the three hammerhead types with respect to the total number of pairwise window intersections that are added into the graph. One can see that for low d values, i.e. high primary sequence specificity, the hammerheads of type I are detected best. This can be explained by the fact that they have one very conserved hairpin loop structure while the other two types show more

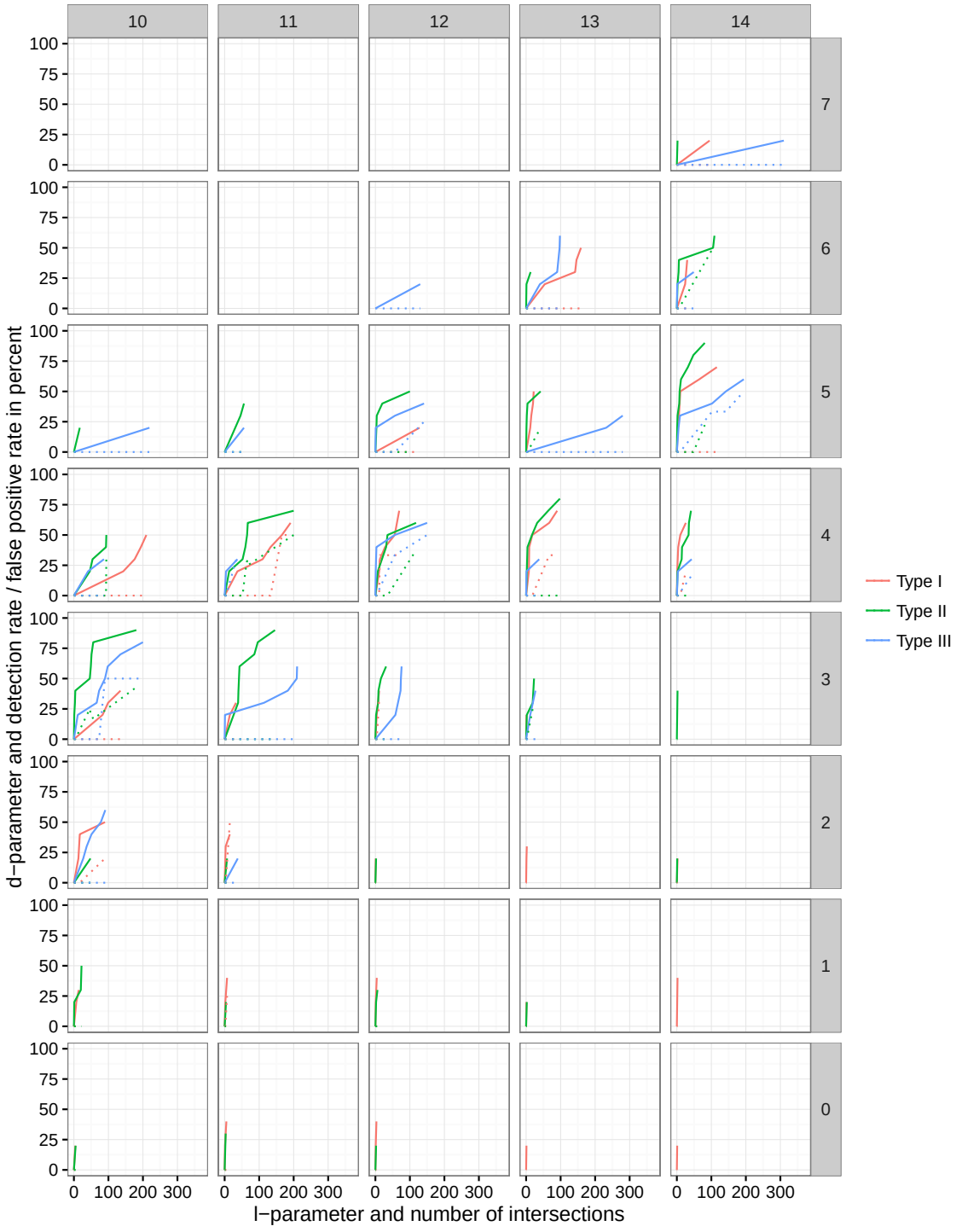


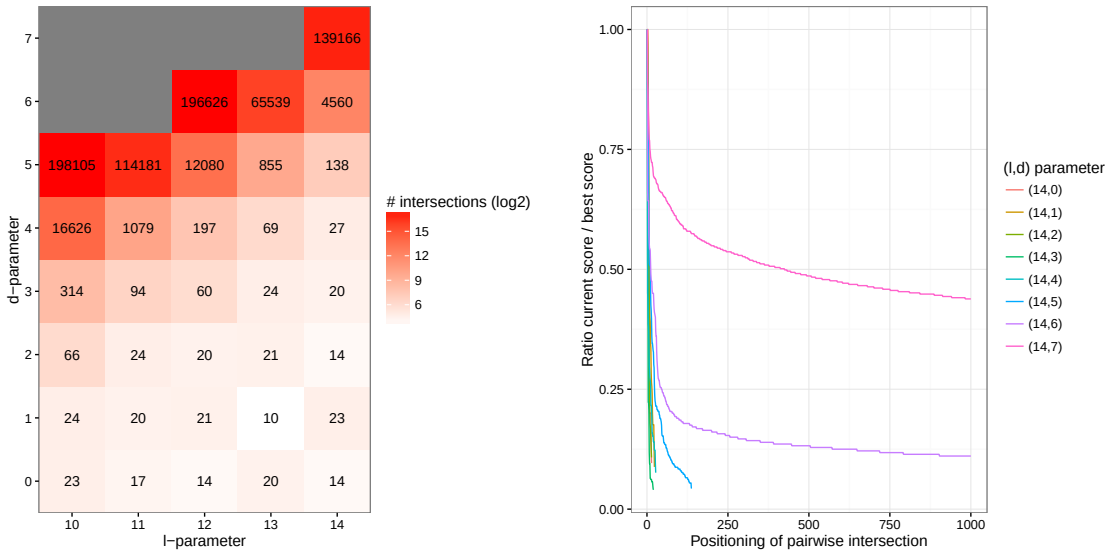
Figure 6.6.: This plot shows the experiments for all three hammerhead types with various l and d parameters. The x-axis on each plot shows the number of pairwise intersections that were added into the graph and the y-axis shows the detection rate for the solid lines and the false positive rate for the dotted lines.

secondary structure motifs that are less conserved. But with less primary sequence specificity, these matches become overshadowed by either multiple exact secondary structure matches of the other hammerhead types or other random matches.

6.4. Discussion

In this chapter, we have shown that our seed based matching approach is able to identify 70% to 80% of the hammerhead sequences correctly. The corresponding false positive rates were always kept below 50%. The questions that remain are: how can we decide which (l, d) parameters should be used? How many pairwise intersection results should be added into the graph without creating too many false positive results? The choice of the (l, d) parameter pair is tied to the expectation of the user concerning the primary sequence conservation of the motifs. Based on the experiments with the hammerhead sequences, we can suggest using a d value that takes three to four nucleotides into account. If no interesting motifs can be found using this parameterization, increasing the d value and thereby decreasing the primary sequence specificity is always an option. Our experiments with genome sizes of $100,000$ nucleotides were finished within a few minutes of CPU time; therefore, changing a parameter and rerunning the experiment is easily done. On the other hand, using pairwise intersections creates quadratic growth with increasing genome size. That is why we have implemented a filtering process for the computed windows in Algorithm 6.2 that may discard all windows whose summed up seed scores are below a user-defined threshold. In this way, the approach might lose potential motifs, because the simple scoring, which only scores base pairs and conserved primary sequence, cannot replace an exact structural alignment, but can significantly reduce the number of pairwise intersections that have to be performed. For the experiments in the previous section, we did not use any prefiltering process and were still able to finish the computations within a few minutes.

Figure 6.7a shows the number of pairwise intersections that were performed by the approach in Section 6.3.1, for each (l, d) parameter combination. With decreasing primary sequence specificity, the number of pairwise window combinations that share at least one seed grows exponentially. Before the graph structure is built, the pairwise window intersections are ordered descending based on their assigned score. Figure 6.7b shows the connection between the decreasing score and the position of the intersection in the ordered list for the first 1000 list positions. For the other list entries, however, the recognizable tendency of the curves continues. We have created this plot for all parameter combinations with $l = 14$, but the results can also be transferred to all other parameterizations. It is apparent that the score for low d values, i.e. more primary sequence conservation is needed for seeds to have the same hash value, decreases much faster than for seeds with higher d value. On the other hand, the curves of parameterizations like (14, 7) and (14, 6), which include zero or one nucleotide in the seed



(a) Number of pairwise intersections

(b) Scores of intersections

Figure 6.7.: (a) shows the number of all pairwise window combinations that have at least one seed in common, for each (l, d) parameter combination. The colors are based on the $\log_2$ values while the cell labels show the actual values. For plot (b), the pairwise intersections have been ordered according to their score. The curve shows the relative decrease in score compared to the best intersection score with increasing list positions up to 1000. Both plots show only the data for the first experiment from Section 6.3.1, but the data for the second experiment from Section 6.3.2 is similar.

calculation, approach a certain minimal value that is different for each parameter pair. And since these minimal ratios are different, we cannot simply suggest one ratio as cutoff that fits all parameterizations. However, what we can do is to specify a different criterion that is similar to the following: if the average of the score ratios of the previous ten entries is not at least x percent greater than the average of the score ratios of the next ten entries, then we will stop inserting the intersections into the graph. For our previous experiments, if we set x to 0.5%, then we would stop inserting intersections between list position 150 and 200. This is in agreement with Figures 6.3 and 6.6 that show that the added seed content of the intersections with a position greater than 200 often cause the false positive rates to exceed 50%.

Now, we can apply this knowledge to further experiments with other families from *Rfam*. The experiments carried out so far can only be an indication that our approach presented here can recognize a large part of the sequence and structure motifs reliably. For further tests, the unbiased and sampled data set from Chapter 5 could be used as long as there is no new benchmark data set with families of the most recent *Rfam* version.

CHAPTER

7

CONCLUSION

In this thesis, we have introduced two approaches for the identification of common RNA motifs in a sample set of not necessarily strongly related organisms. In the first approach, we have used exact matching of *in silico* predicted RNA secondary structures of multiple viral reference genomes to detect shared motifs, which might be essential for the infection of host cells and the subsequent replication of the viral genomes within the infected cells. One potential mechanism that might explain the common motifs in evolutionary distant viruses is convergent evolution, which describes the phenomenon that two different species that are originated from two ancestors share related or similar traits. Whenever we identify common RNA motifs, there are two options how they developed: based on a common ancestor followed by divergent evolution, or by convergent evolution based on similar selection criteria in common host organisms. We tried to maximize the chance of finding common motifs produced by the latter phenomenon by retaining only those secondary structure matches whose underlying nucleotide sequences show low similarities and cannot be found in a protein database. Since our data set consisted of 4,667 viral reference genomes and a total of 31,346,596 secondary structure strings to compare, it is a possibility that some of these matches occur randomly. By repeating the experiment with a random genome of the same size, we were able to choose only those secondary structure matches that did not occur in the random set. In the end, we were able to identify 69 common RNA motifs that share the same secondary structure with a length of at least 50 nucleotides in at least three taxonomically distant viruses. Furthermore, none of the sequences that fold into those secondary structures could be assigned to an existing RNA family in the *Rfam* database. While these are strong indicators for structural convergence, computational prediction does not enable us to find out whether those structures have a real function in the interplay between virus and host. Now, it is up to experimental biologists to synthesize the sequences and test their effect on the infection of a host organism using several viruses.

During this experiment, we had to overcome several obstacles: first, the exact matching using suffix arrays left us with maximal repeats of secondary structures in multiple viral genomes that do not necessarily represent well-formed RNA structure strings. This led to an additional time-

consuming and I/O intensive post-processing step that was used to find the longest substring of the maximal repeat that is a well-formed RNA structure. Second, while trying to assign the identified common motifs to already existing RNA families in *Rfam*, we found that the majority of the known sequences were assigned to the tRNA family; the other families were underrepresented. Last, known RNA families often have conserved sequences in the unpaired loop regions of the structure. Therefore, just matching the predicted secondary structures misses useful information that might help to increase the specificity of the approach. Creating a combined string consisting of dot-bracket characters in the paired regions and primary sequence nucleotides in the unpaired regions was too restrictive. Whenever we found a cluster where all loop sequences were identical, all other segments of the sequences were nearly identical, too.

The first problem led to the creation of a new data structure with the name viable suffix tree. It takes the constraints on the RNA secondary structure into account and can be used to output only well-formed structures as maximal repeats. In general, this data structure does not only work for RNA structures but can also be used for any other problem domain for which a set of allowed words can be defined. We have shown that Ukkonen's algorithm for the suffix tree construction can be modified in order to construct a viable suffix tree for RNA sequences directly from the string. The modified algorithm is not guaranteed to work for other problems, but it is possible to construct the classical suffix tree first and later prune the tree so that only allowed words are represented as leaves in the tree. The time complexity for constructing viable suffix trees for well-formed RNA sequences remains linear while for languages that can be described using a CFG, the time complexity of the CYK algorithm poses an upper bound. However, for other languages in the Chomsky hierarchy, this is still an open research challenge. Additionally, we have shown that the viable suffix tree can reduce the memory footprint of classical suffix trees if were only interested in suffixes that are valid words. In order to identify maximal repeats of RNA structures that are valid words of the language, we have extended the approach used for classical suffix trees. Thereby, we have reduced the quadratic time, which is needed due to the post-processing of the suffix array results, to linear time with an additional factor that describes the nesting depth of the RNA structure. In comparison to previous work on constrained suffix trees, the viable suffix tree is one of the most versatile structures that does not restrict the grammar that describes the language of allowed words. On the other hand, in comparison to suffix arrays, the space requirements of viable suffix trees are still high. For the future, it might be worth investigating how we can restrict the suffixes in the (enhanced) suffix array in a similar way.

The overrepresentation of the tRNAs among all assigned sequences in the first experiment made us wonder whether other RNA data sets might be balanced with regard to the RNA family composition. When looking for a suitable benchmark data set to test RNA motif detection and alignment algorithms, we ran into an open problem in RNA bioinformatics. We found that BRali-Base is the most widely used benchmark in the field for assessing the accuracy of RNA secondary

structure alignment methods. In most case studies based on the BRaliBase benchmark, one can observe a puzzling drop in accuracy in the 40% to 60% sequence identity range, the so-called "BRaliBase dent". We showed that this dent is due to a bias in the composition of the BRaliBase benchmark, namely the inclusion of a disproportionate number of tRNAs, which exhibit a very conserved secondary structure. Our analysis, aside of its interest regarding the specific case of the BRaliBase benchmark, also raises important questions regarding the design and use of benchmarks in computational biology. Furthermore, we have shown that a sampling approach that restricts the presence of the most abundant RNA families can prevent such artifacts and that a careful choice of a benchmark set is necessary to gain meaningful results that enable us to judge the performance of a tool. The *Rfam* version that was used to generate the current BRaliBase data set is already a decade old. Immediately after one of the co-authors presented the results of this study at a conference in the beginning of 2017, a group of volunteers has formed to tackle the nontrivial task of creating an unbiased and up to date benchmark data set based on the current version of the *Rfam* database.

The limitation of exact structure matching is the need for long common stretches of secondary structures that are not allowed to have a mismatch at any position. Current state of the art *de novo* screening methods use time consuming structural alignment methods that we want to avoid. Therefore, we use RNA sequence and structure seeds that can be encoded into an integer hash value. The approach collects the seeds for user-defined window lengths and subsequently performs a pairwise comparison of the seed content of multiple windows. The pairwise relations between the seed contents of multiple windows are represented using an undirected graph, where an edge represents two windows sharing multiple seeds. We used various taxonomically distant sequences that are assigned to the hammerhead ribozyme families to evaluate the performance of the approach. Depending on the sequence and structure similarities of all sequences, the approach is able to detect 60% to 80% of the hammerhead motifs with a false positive rate of less than 50%. Compared to the other existing methods, our approach takes only a few minutes of CPU time instead of days or even years. But while the other methods directly identify the shared structures that are potentially unidentified functional RNAs, our approach returns potential windows in which multiple common seed sequences have been found. Therefore, to identify the complete and shared RNA structure, an additional RNA sequence and structure alignment should be performed. But we now have the advantage that the set of sequences that need to be aligned is already prefiltered and small. In the end, there is still need for an extensive benchmark of the approach using multiple RNA families. This could be based on our unbiased and sampled version of the BRaliBase data set as long as there is no new benchmark data set with families of the most recent *Rfam* version.

Appendices

APPENDIX

A

COMPLETE PRUNING OF THE SUFFIX TREE

Let S be a string whose viable suffix tree should be generated. Furthermore, assume that the suffix tree of $S\$$ is given and consists of vertices and edges in the form of a quadruple (source node, destination node, first position of edge label in $S\$$, last position of edge label in $S\$$). We will describe and proof the correctness of the pruning algorithm that creates a complete and compact viable suffix tree from a given suffix tree of an input string S plus the sentinel. The simplified version of this algorithm is depicted in Section 4.2.

The pruning algorithm that creates a compact viable suffix tree traverses the classical suffix tree as basis in a depth-first order. The pruning starts at the root and recursively visits every internal node as well as every leaf exactly once. The removal of leaves without a lvp and internal nodes that are not needed anymore is executed in a bottom-up fashion. Thereby, the algorithm differentiates between leaves and internal nodes. For leaves, we distinguish three different cases: first, if the lvp of the current leaf/suffix is longer or equal to the depth of its parent internal node, then the edge from the parent to the leaf is kept. Furthermore, the edge may be shortened if the lvp does not include the whole suffix. Note, edges with empty labels might occur if the length of the lvp is equal to the depth of the parent node (case 1). Second, if the lvp has length zero, then the edge and the leaf attached to it are removed from the tree (case 3). Third, if the lvp is shorter than the depth of its parent internal node but greater than zero, the edge to the parent node is removed and the leaf is marked for insertion at the internal node with the greatest depth that is shorter than the lvp (case 2).

Internal nodes, on the other hand, are handled differently. First, all children nodes are processed recursively until the bottom-up processing reaches the root. The recursive call of the child node returns three different parameters: first, the number of edges between the current internal node and the child. Usually, the number is either one if the child node still exists, zero if the child node was deleted, or the special case greater than one, which will be explained later. Second, a set of leaves whose lvp is shorter than the depth of the current node; and third, the

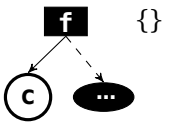
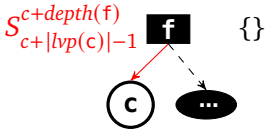
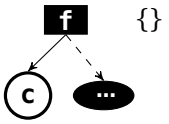
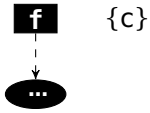
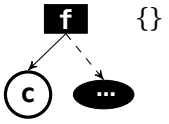
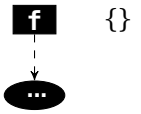
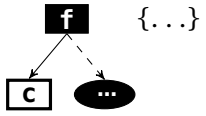
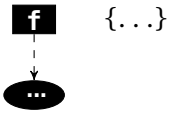
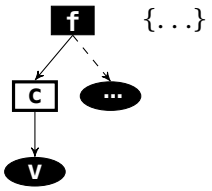
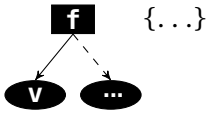
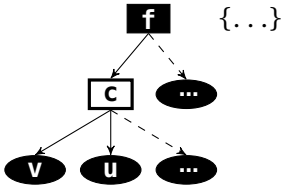
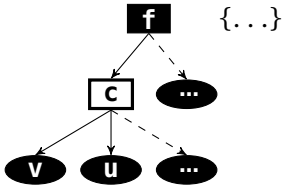
Algorithm A.1 Complete pruning of the suffix tree to generate a viable suffix tree

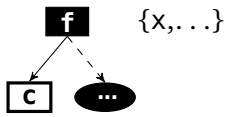
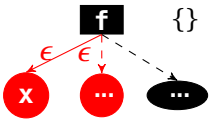
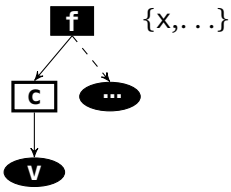
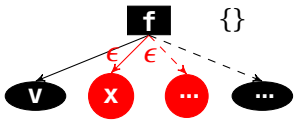
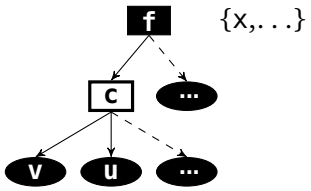
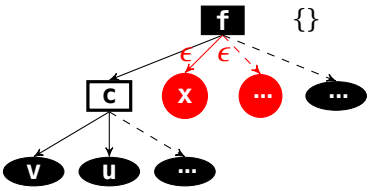
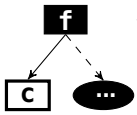
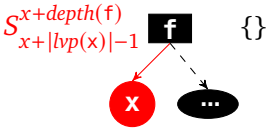
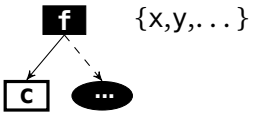
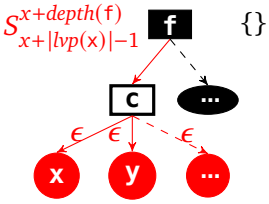
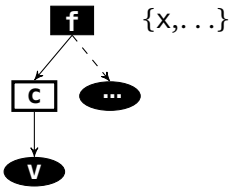
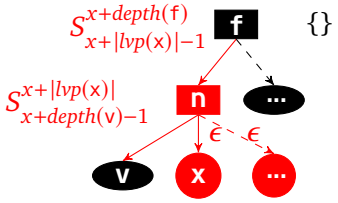
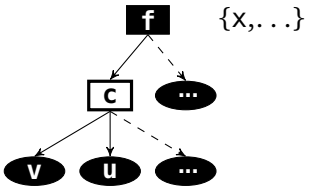
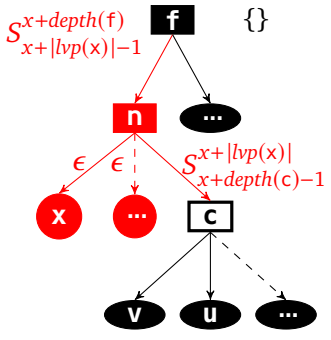
```
1: function PRUNE
2:     DFS(root, root, −1, −1, 0)
3: function DFS(parent, child, i, j, len)
4:     if child is leaf then                                              ▷ Cases 1-3
5:         edge_len ← |lvp(child)| − depth(parent)
6:         if edge_len ≥ 0 and lvp(child) ≠ ϵ then                        ▷ Case 1
7:             replace edge (parent, child, i, j) with (parent, child, i, i + edge_len − 1)
8:             return (1, {}, −1)
9:         else if lvp(child) ≠ ϵ then                                    ▷ Case 2
10:            remove edge (parent, child, i, j)
11:            return (0, {child}, |lvp(child)|)
12:        else                                                           ▷ Case 3
13:            remove leaf child and edge (parent, child, i, j)
14:            return (0, {}, −1)
15:    else                                  ▷ (child is internal node or root.) Cases 4-13
16:        num_childern ← 0
17:        Leaves ← {}
18:        lvp_len ← −1
19:        for edges of the form (child, v, k, l) do          ▷ Recursion over every child
20:            (x, Y, z) ← DFS(child, v, k, l, len + l − k + 1)
21:            num_childern ← num_childern + x
22:            if Y ≠ {} then
23:                Leaves ← Leaves ∪ Y
24:                lvp_len ← z                ▷ lvp_len of all children in set Leaves is the same
25:        if child = root then                                  ▷ Termination of recursion
26:            return
27:        if lvp_len < depth(parent) then                                ▷ Cases 4-6
28:            if num_childern = 0 then                                   ▷ Case 4
29:                remove node child and edge (parent, child, i, j)
30:                return (0, Leaves, lvp_len)
31:            else if num_childern = 1 then                              ▷ Case 5
32:                let (child, v, k, l) be the only outgoing edge of node child
33:                replace edges (parent, child, i, j) and (child, v, k, l) by (parent, v, i, l)
34:                remove node child
35:            return (1, Leaves, lvp_len)
```

Algorithm A.1 Complete pruning of the suffix tree to generate a viable suffix tree (continued)

```
36:     else if lvp_len = depth(parent) then                                  ▷ Cases 7-9
37:         if num_childern = 0 then                                          ▷ Case 7
38:             remove node child and edge (parent, child, i, j)
39:         else
40:             if num_childern = 1 then                                      ▷ Case 8
41:                 let (child, v, k, l) be the only outgoing edge of node child
42:                 replace edges (parent, child, i, j) and (child, v, k, l) by (parent, v, i, l)
43:                 remove node child
44:             num_childern ← 1
45:         for l ∈ Leaves do
46:             add edge (parent, l, 0, −1) with empty label
47:         return (num_childern + |Leaves|, {}, −1)
48:     else                                                                  ▷ Cases 10-13
49:         if num_childern = 0 then                                          ▷ Cases 10-11
50:             replace (parent, child, i, j) with
                (parent, child, i, i + lvp_len − depth(parent))
51:             if |Leaves| = 1 then                                          ▷ Case 10
52:                 replace child by l ∈ Leaves in edge
                    (parent, child, i, i + lvp_len − depth(parent))
53:             else                                                          ▷ Case 11
54:                 for l ∈ leaves do
55:                     add edge (child, l, 0, −1) with empty label
56:         else                                                              ▷ Cases 12-13
57:             create novel internal branching node new
                replace edge (parent, child, i, j) by
58:             (parent, new, i, i + lvp_len − depth(parent)) and
                (new, child, i + lvp_len − depth(parent) + 1, j)
59:             if num_children = 1 then                                      ▷ Case 12
60:                 let (child, v, k, l) be the only outgoing edge of node child
61:                 replace edges (new, child, i + lvp_len − depth(parent) + 1, j) and
                    (child, v, k, l) by (new, v, i + lvp_len − depth(parent) + 1, l)
62:                 remove node child
63:             for l ∈ Leaves do
64:                 add edge (new, l, 0, −1) with empty label
65:         return (1, {}, −1)
```

Table A.1.: The table lists all cases that can occur while pruning the nodes of the suffix tree. Note that the tree graphics only show a subtree, i.e. `f` is not necessarily the root node. Internal nodes (including the root) are represented by rectangles; leaves are drawn as circles; ellipses represent nodes that can be either one of these options. The current node is shown in white with the label `c`; the parent node has the label `f`; children of `c` are labeled with `v` and `u`. Newly inserted internal nodes are labeled `n` and leaves are either called `x` or `y`. Leaf characters that are used within formulas represent the starting position of the suffix in the input string. All newly inserted nodes are shown in red. Also shortened or newly inserted edges are drawn in red. Furthermore, all changed edges are labeled with the new string that they are representing. The set of leaves that have to be reinserted are drawn next to the parent node. This set is cleared as soon as these leaves are inserted into the tree. The label `...` in either a node or the set represent 0 or more nodes while the labels `x` and `y` represent exactly one node in the tree or one leaf in the set. The edges leading to these putative nodes are drawn dashed.

Case	Before	After
1 c is leaf $lvp(c) \neq \epsilon$ $\lvert lvp(c)\rvert \geq depth(f)$		
2 c is leaf $lvp(c) \neq \epsilon$ $\lvert lvp(c)\rvert < depth(f)$		
3 c is leaf $lvp(c) = \epsilon$		
4 c is internal node $\lvert lvp(...)\rvert < depth(f)$		
5 c is internal node $\lvert lvp(...)\rvert < depth(f)$		
6 c is internal node $\lvert lvp(...)\rvert < depth(f)$		

7 c is internal node $\|lvp(x)\| = depth(f)$		
8 c is internal node $\|lvp(x)\| = depth(f)$		
9 c is internal node $\|lvp(x)\| = depth(f)$		
10 c is internal node $\|lvp(x)\| > depth(f)$		
11 c is internal node $\|lvp(x)\| > depth(f)$		
12 c is internal node $\|lvp(x)\| > depth(f)$		
13 c is internal node $\|lvp(x)\| > depth(f)$		

length of said lvp.

For the processing of the internal node and the aggregated return values of the children, we can first distinguish between three different cases: first, the lvp length is shorter than the depth of the parent node (cases 4-6). Then, if the current node does not have any children, it and the corresponding edge are deleted (case 4). But, if it has exactly one outgoing edge, then this edge is replaced by an edge from its parent to its only child and the current node gets deleted (case 5). In any other case, no further action is needed (case 6).

Second, the lvp length is equal to the depth of the parent node (cases 7-9). The first two cases, the current node does not have any children or it has exactly one outgoing edge, are handled as before (cases 7-8). But additionally, all leaves in our set are added to the parent node by edges with empty labels, because their lvps equal the depth of the parent node (case 9). Furthermore, the return value for the number of outgoing edges is adjusted according to the number of empty edges that were added. This value can now be greater than one.

In the last four cases, the lvp length is greater than the depth of the parent node (cases 10-13). If the current node does not have any children, the edge from the parent node is shortened such that the depth of the current node will be equal the length of the lvp (cases 10-11). Furthermore, if the set of leaves has only one element, then the current node is replaced by this element and deleted after that (case 10). But, if the set has more than one element, then all of them are added to the current node by edges with empty labels, because now the depth of the current node equals the lengths of their lvps (case 11).

On the other hand, if the current node has one or more children, we introduce a new internal node and split the edge between the current node and the parent node by inserting the new node such that its depth equals the length of the lvps. Subsequently, all leaves in our set are added to the new node by edges with empty labels (cases 12-13). Furthermore, if the current node only has one outgoing edge, then this edge is replaced by an edge from the new node to its only child and the current node gets deleted (case 12).

Lemma A.1. *The PRUNE function correctly transforms the suffix tree of S into a compact viable suffix tree.*

Proof. The PRUNE function distinguishes between internal nodes, leaves, and the root node. The given suffix tree of S of length n is compact and has exactly n leaves for all n suffixes of S. Therefore, the pruning algorithm only needs to take care of three tasks: first, remove all leaves that do not have a lvp, relocate/shorten leaves whose lvp does not cover the full length of the suffix, and remove all internal nodes that only have ≤ 1 children such that the tree stays compact. In the following, we will only talk about removing or relocating the leaves of the tree, but this also includes the removal and insertion of the edges that connect the leaves and the obsolete internal nodes to their parent nodes.

In the cases 1 to 3, all leaves either are correctly shortened according to the length of the lvp if

it is longer than the depth of the parent node (case 1), are removed if the lvp is the empty word (case 3), or are stored such that they can be inserted at the correct internal node later (case 2). Case 3 takes care that all suffixes without a lvp are removed. Case 2 records all suffixes whose lvps are shorter than the suffix and that have to be relocated. The sets in which those leaves will be stored are passed on from the children nodes to the parent nodes as return values of the recursion. Since the results of this post-order traversal are collected in a bottom-up strategy, it helps to bring to mind that the lvps of all suffixes that are collected at a parent node are equal. Therefore, the position of reinsertion for all elements is the same and has to be tested just once for each parent node that the set is passed through. Using these three steps, all cases that can occur at leaf nodes are covered. Note, leaves whose lvps cover the full length of the suffix are also included in case 1. Furthermore, the correct number of children is returned to the parent node, so either one if the leaf was shortened or stayed the same, or zero in case the leaf got deleted or will be reinserted later.

For the internal nodes, we have to differentiate between different situations in order to fulfill the two last tasks. In general, cases 4, 7, and 10 are taking care of the compactness of the tree. If the current internal node does not have any children, then it must be removed in order to keep the three compact. On the other hand, cases 5, 8, and 12 account for the possibility that the current node has exactly one child. This child is directly linked to the parent node and the current node can be deleted subsequently. Using these rules, the compactness of the resulting tree can be guaranteed.

In order to make sure that all shortened leaves are relocated to their correct position, the algorithm distinguishes three situations. In the first situation, the set for reinsertion is empty or the length of the lvp of the elements of this set is smaller than the depth of the parent node (cases 4-6). No further action has to be taken here, because the correct relocation position for all leaves in the set is still further up in the tree (higher up than the current parent node). Also, the set of leaves that have to be reinserted is passed through to the parent node.

Second, the length of the lvp of the set elements is equal to the depth of the parent node (cases 7-9). As the lvp lengths of the set elements equal the depth of the parent node, all leaves are relocated by adding them as empty edges directly to the parent node. Also, an empty set is now passed to the parent, since all children of this branch have been reinserted.

Last, the length of the lvp of the set elements is greater than the depth of the parent node (cases 10-13). If the current internal node does not have any children and the set only holds one leaf, then this leaf can replace the internal node by shortening the edge such that the depth equals the length of the lvp. On the other hand, if there are multiple leaves in the set, then the current node can be shortened such that the depth equals the length of the lvp. Subsequently, all leaves can be added via edges with empty labels. Otherwise, a new internal node, whose depth equals the length of the lvp, is needed and all leaves are added via edges with empty labels to this node. If the current internal node has more than one child, it is added to the new

branching node instead of the parent node. And if not, its child is directly connected to the new branching node. In the end, one child is reported as return value, because there was no new edge introduced or an existing edge deleted, and an empty set is given to the parent node.

Using these case distinctions that cover all possibilities, the algorithm can fulfill the last two tasks, as it correctly removes all internal nodes with less than two children by directly attaching the child to the parent. Furthermore, it reinserts all leaves at the correct position in the tree according to their lvp. □

Lemma A.2. *The suffix tree can be pruned in* $\mathcal{O}(n)$ *using the* PRUNE *function.*

Proof. Basically, the PRUNE function uses an adapted depth-first search approach. During the recursion every node will be visited exactly once. This includes the at most n leaves and $n-1$ internal nodes. Additionally, the length of the lvp of some suffix might be smaller than the depth of the parent node. In this case, these nodes are removed at the current position and are added to a set that is passed on to the next parent node during the bottom-up processing. The leaf is kept in this set until it gets reinserted at the correct position later. Note that only leaves are inserted into this set and this also only once.

Furthermore, once the leaves in the set are reinserted into a higher position in the tree, an empty list is passed to the parent node as result of the function call. This guarantees that each leaf is inserted at most once. All other operations can be performed in constant time. These are: deleting the node and the corresponding edge that are currently processed, replacing a node and the corresponding edge by an edge to its only child or any other node from the set of leaves that have to be reinserted, changing the end position of an edge label, and splitting an edge by inserting a new internal node.

All in all, this results in a runtime of $\mathcal{O}(n)$, which is mainly determined by the cost of the depth-first search, because the processing of a node can be done in constant time. □

So, the total cost of creating a viable suffix tree for string S and given language L is $\mathcal{O}(|S|)$. This can be achieved by first building the classical suffix tree in $\mathcal{O}(|S|)$ (assuming a constant alphabet size) and the subsequent pruning of the tree given the precomputed *End* table for S.

APPENDIX

B

DETAILS ABOUT BRALIBASE BENCHMARKS

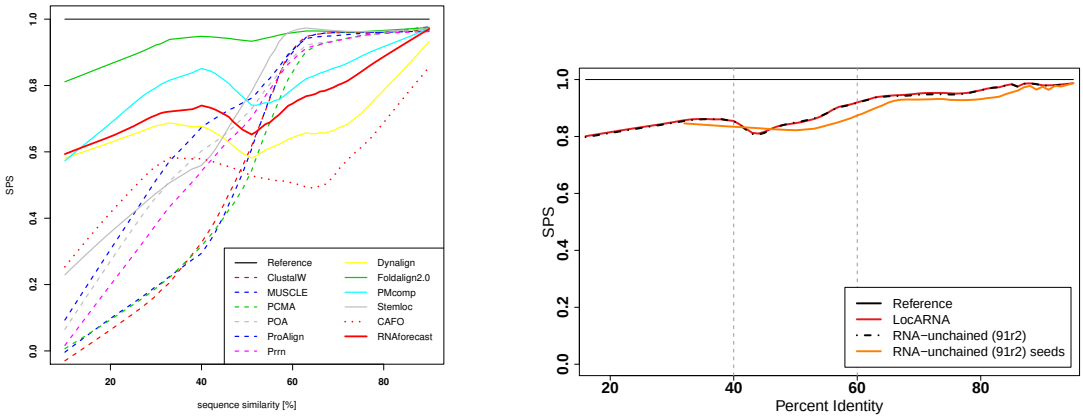
Here, we provide more details on the programs that were applied to the BRaliBase data set, an unpublished Plan C approach, and the comparison of different `RNA-unchained` modes.

B.1. Details about programs used for benchmark re-computation

Table B.1.: Summary of programs, parameters, and errors during the re-evaluation of the 2006 version of BRaliBase used for Figures 5.2 to 5.4 and Figures B.1b, B.2 and B.3.

Program	Version	Parameters	Errors
`Dynalign`	RNAstructure 5.8	default	/
`Foldalign`	2.1.1[15]	`-no_pruning -global`	/
`Lara`	1.3.2a	default parameter file	/
`LocARNA`	1.8.7	default	/
`PMcomp`	11.09.2004	default	392 missing alignments, because of failed backtracks and memory consumption > 60 GB.
`RAF`	1.00	`predict`	/
`RNA-unchained`	1.0	`-c -a -f -r`	/
`SPARSE`	`LocARNA` 1.8.7	`-sparse`	/
`Stemloc`	`DART` Library 0.2	`-slow -na 1000`	99 missing alignments, because alignment score is $-\infty$. `-na` was raised to 1000 as countermeasure.

[15] We also tested `Foldalign2.5` (preRelease3), but `Foldalign` 2.1.1 performed better with SPS as measure.



(a) **(b)**

Figure B.1.: (a) shows another evaluation of the 2005 BRaliBase data that includes the Plan C approach `RNAforecast` in red. (b) shows smoothed curves based on the 2006 BRaliBase data for `LocARNA`, `RNA-unchained` for the complete data set, and `RNA-unchained` only for data points for which seeds have been found.

Figure 5.1 was generated using the original BRaliBase data[16] from [33]. The scoring scheme for Figure 5.4b was implemented following the original Perl script[17] [47].

B.2. `RNAforecast`: a Plan C approach

The plot in Figure B.1a was extracted from an unpublished manuscript (from 2007) that introduces the tool `RNAforecast` for RNA consensus structure prediction and alignment. We do not have access to the source code of the tool or the SPS measurement data anymore. Therefore, no further details concerning the evaluation in Figure B.1a can be given.

B.3. Comparison of `RNA-unchained` modes

`RNA-unchained`, as a heuristic that computes seeds for `LocARNA`, can be used in two modes. The black solid-dotted line of Figure B.1b shows the mode that uses `LocARNA` even if no seeds were found. This curve agrees with the one of `LocARNA`. On the other hand, the second mode only calls `LocARNA` if seeds were found; therefore, not all data points of the 2006 BRaliBase data set are integrated. Nevertheless, a small hint of a dent can be seen in the 60% to 40% identity region of the orange curve. For a detailed analysis of more parameters see [15].

[16] Data is accessible through `http://projects.binf.ku.dk/pgardner/bralibase/bralibase2.html`.
[17] Perl script is available at `http://www.tbi.univie.ac.at/RNA/PMcomp/pmcomp.pl`.

B.4. Detailed plots for all RNA families individually

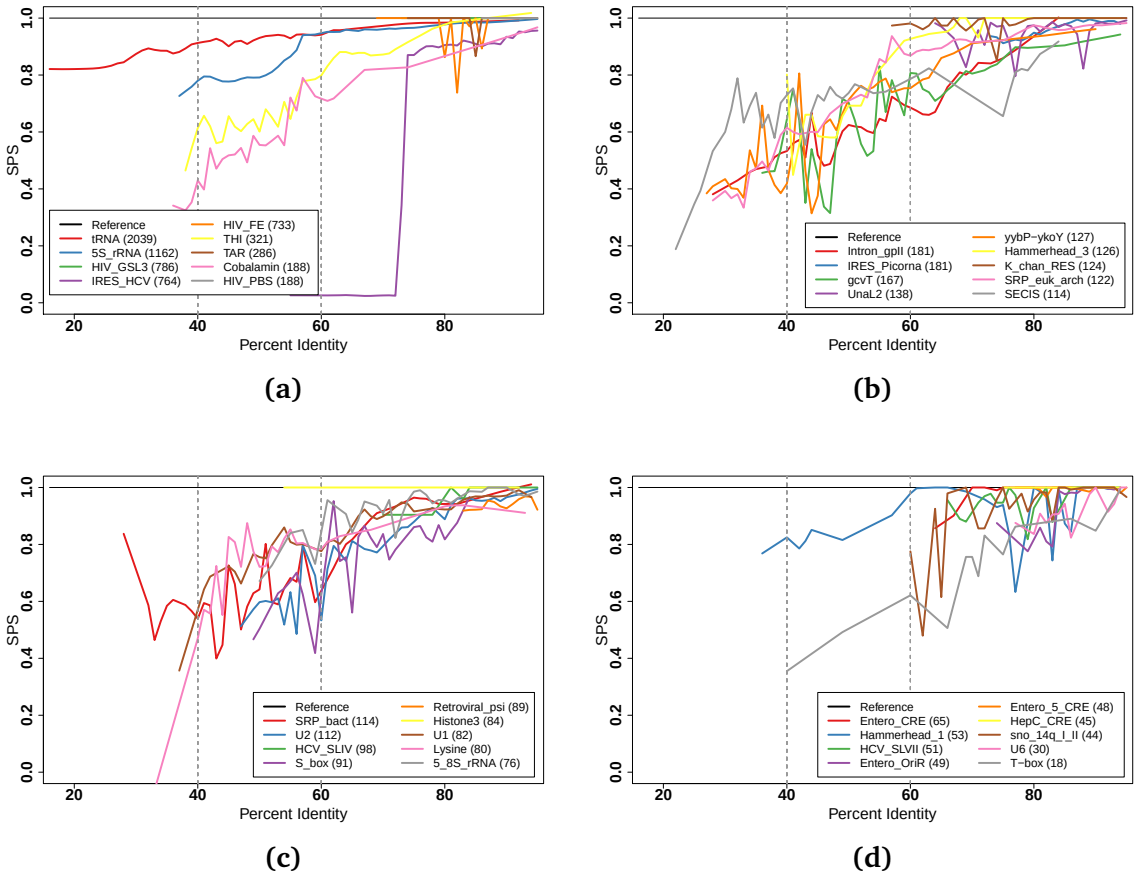


Figure B.2.: Plots (a) to (d) show the performance of LocARNA for all 36 RNA families individually. The family names are extended by the agglomerated number of alignments per family in square brackets.

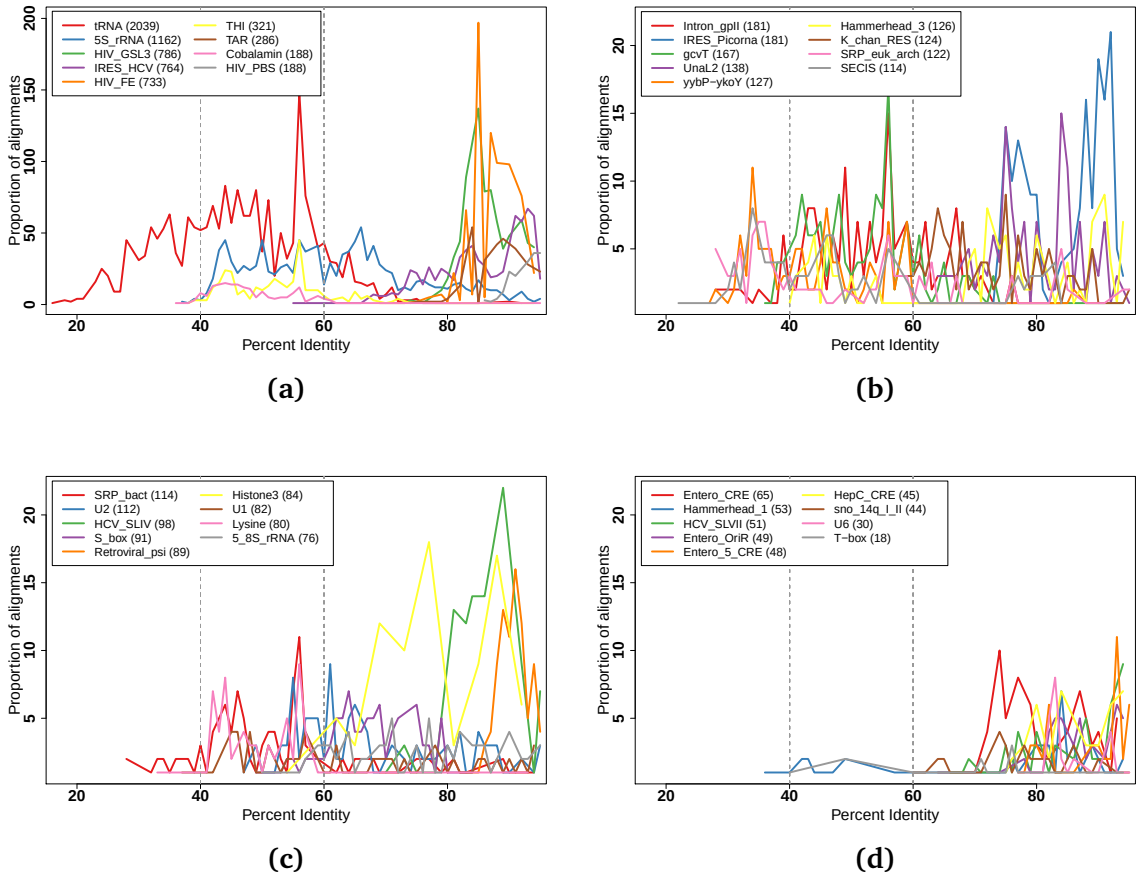


Figure B.3.: Plots (a) to (d) show the distribution of the number of pairwise alignments for all 36 RNA families individually over the range of different sequence identities. The family names are extended by the agglomerated number of alignments per family in square brackets. Note, plot (a) has a different y-axis range than plots (b) to (d).

BIBLIOGRAPHY

[1] M. I. Abouelhoda and E. Ohlebusch. "Chaining algorithms for multiple genome comparison." In: *Journal of Discrete Algorithms* 3.2-4 (2005), pp. 321–341.

[2] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. "Replacing suffix trees with enhanced suffix arrays." In: *Journal of Discrete Algorithms* 2.1 (2004), pp. 53–86.

[3] S. F. Altschul, W. Gish, W. Miller, *et al.* "Basic local alignment search tool." In: *Journal of Molecular Biology* 215.3 (1990), pp. 403–410.

[4] A. Amir, E. Chencinski, C. S. Iliopoulos, *et al.* "Property matching and weighted matching." In: *Theoretical Computer Science* 395.2-3 (2008), pp. 298–310.

[5] A. Andersson, N. J. Larsson, and K. Swanson. "Suffix Trees on Words." In: *Algorithmica* 23.3 (1999), pp. 246–260.

[6] T. Babak, B. J. Blencowe, and T. R. Hughes. "Considerations in the identification of functional RNA structural elements in genomic alignments." In: *BMC Bioinformatics* 8 (2007), p. 33.

[7] R. A. Baeza-Yates, E. Chávez, and M. Crochemore, eds. *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*. Vol. 2676. Lecture Notes in Computer Science. Springer, 2003.

[8] M. Bailly-Bechet, M. Vergassola, and E. Rocha. "Causes for the intriguing presence of tRNAs in phages." In: *Genome Research* 17.10 (2007), pp. 1486–1495.

[9] D. Baltimore. "Expression of animal virus genomes." In: *Bacteriological Reviews* 35.3 (1971), p. 235.

[10] M. Bauer, G. W. Klau, and K. Reinert. "Accurate multiple sequence-structure alignment of RNA sequences using combinatorial optimization." In: *BMC Bioinformatics* 8 (2007), p. 271.

[11] D. A. Benson, M. Cavanaugh, K. Clark, *et al.* "GenBank." In: *Nucleic Acids Research* 45 (2017), pp. D37–D42.

[12] H. S. Bernhardt. "The RNA world hypothesis: the worst theory of the early evolution of life (except for all the others)[a]." In: *Biology Direct* 7.1 (2012), pp. 23+.

[13] P. Bille, J. Fischer, I. L. Gørtz, *et al.* "Sparse Suffix Tree Construction in Small Space." In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*. Ed. by F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, *et al.* Vol. 7965. Lecture Notes in Computer Science. Springer, 2013, pp. 148–159.

[14] J. Błażewicz, P. Formanowicz, and P. Wojciechowski. "Some remarks on evaluating the quality of the multiple sequence alignment based on the BAliBASE benchmark." In: *International Journal of Applied Mathematics and Computer Science* 19.4 (2009), pp. 675–678.

[15] L. Bourgeade, C. Chauve, and J. Allali. "Chaining Sequence/Structure Seeds for Computing RNA Similarity." In: *Journal of Computational Biology* 22.3 (2015), pp. 205–217.

[16] R. K. Bradley, L. Pachter, and I. Holmes. "Specific alignment of structured RNA: stochastic grammars and sequence annealing." In: *Bioinformatics* 24.23 (2008), pp. 2677–2683.

[17] A. Bremges, S. Schirmer, and R. Giegerich. "Fine-tuning structural RNA alignments in the twilight zone." In: *BMC Bioinformatics* 11.1 (2010), p. 222.

[18] J. R. Brister, D. Ako-adjei, Y. Bao, *et al.* "NCBI Viral Genomes Resource." In: *Nucleic Acids Research* 43.D1 (2015), p. D571.

[19] J. R. Brister, Y. Bao, C. Kuiken, *et al.* "Towards Viral Genome Annotation Standards, Report from the 2010 NCBI Annotation Workshop." In: *Viruses* 2 (10 2010), pp. 2258–2268.

[20] M. Burrows and D. J. Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Report 124. Palo Alto, CA, USA: Digital Systems Research Center, 1994.

[21] A. Carbone. "Codon Bias is a Major Factor Explaining Phage Evolution in Translationally Biased Hosts." In: *Journal of Molecular Evolution* 66.3 (2008), pp. 210–223.

[22] W. I. Chang and E. L. Lawler. "Sublinear Approximate String Matching and Biological Applications." In: *Algorithmica* 12.4/5 (1994), pp. 327–344.

[23] M. Chatzou, C. Magis, J.-M. Chang, *et al.* "Multiple sequence alignment modeling: methods and applications." In: *Briefings in Bioinformatics* 17.6 (2016), pp. 1009–1023.

[24] N. Chirico, A. Vianelli, and R. Belshaw. "Why genes overlap in viruses." In: *Proceedings of the Royal Society of London B: Biological Sciences* 277.1701 (2010), pp. 3809–3817.

[25] K. Darty, A. Denise, and Y. Ponty. "VARNA: Interactive drawing and editing of the RNA secondary structure." In: *Bioinformatics* 25.15 (2009), pp. 1974–1975.

[26] S. Djebali, C. A. Davis, A. Merkel, *et al.* "Landscape of transcription in human cells." In: *Nature* 489.7414 (2012), pp. 101–108.

[27] C. B. Do, C.-S. S. Foo, and S. Batzoglou. "A max-margin model for efficient simultaneous alignment and folding of RNA sequences." In: *Bioinformatics* 24.13 (2008), pp. i68–i76.

[28] R. F. Doolittle. "Convergent evolution: the need to be explicit." In: *Trends in Biochemical Sciences* 19.1 (1994), pp. 15–18.

[29] R. D. Dowell and S. R. Eddy. "Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction." In: *BMC Bioinformatics* 5 (2004), p. 71.

[30] S. R. Eddy and R. Durbin. "RNA sequence analysis using covariance models." In: *Nucleic Acids Research* 22.11 (1994), p. 2079.

[31] A. Fire, S. Xu, M. K. Montgomery, *et al.* "Potent and specific genetic interference by double-stranded RNA in *Caenorhabditis elegans*." In: *Nature* 391.6669 (1998), pp. 806–811.

[32] P. P. Gardner and R. Giegerich. "A comprehensive comparison of comparative RNA structure prediction approaches." In: *BMC Bioinformatics* 5.1 (2004), p. 140.

[33] P. P. Gardner, A. Wilm, and S. Washietl. "A benchmark of multiple sequence alignment programs upon structural RNAs." In: *Nucleic Acids Research* 33.8 (2005), pp. 2433–2439.

[34] R. Giegerich. "Introduction to Stochastic Context Free Grammars." In: *RNA Sequence, Structure, and Function: Computational and Bioinformatic Methods*. Ed. by J. Gorodkin and W. L. Ruzzo. Humana Press, 2014, pp. 85–106.

[35] R. Giegerich and S. Kurtz. "A Comparison of Imperative and Purely Functional Suffix Tree Constructions." In: *Science of Computer Programming* 25.2-3 (1995), pp. 187–218.

[36] R. Giegerich and S. Kurtz. "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction." In: *Algorithmica* 19.3 (1997), pp. 331–353.

[37] R. Giegerich, S. Kurtz, and J. Stoye. "Efficient implementation of lazy suffix trees." In: *Software: Practice and Experience* 33.11 (2003), pp. 1035–1049.

[38] R. Giegerich, B. Voß, and M. Rehmsmeier. "Abstract shapes of RNA." In: *Nucleic Acids Research* 32.16 (2004), p. 4843.

[39] W. Gilbert. "Origin of life: The RNA world." In: *Nature* 319.6055 (1986), p. 618.

[40] J. Gorodkin and I. L. Hofacker. "From structure prediction to genomic screens for novel non-coding RNAs." In: *PLOS Computational Biology* 7.8 (2011), e1002100+.

[41] S. Griffiths-Jones, A. Bateman, M. Marshall, *et al.* "Rfam: an RNA family database." In: *Nucleic Acids Research* 31.1 (2003), pp. 439–41.

[42] A. R. Gruber, S. Findeiß, S. Washietl, *et al.* "RNAz 2.0: Improved Noncoding RNA Detection." In: *Biocomputing 2010: Proceedings of the Pacific Symposium, Kamuela, Hawaii, USA, 4-8 January 2010*. Ed. by R. B. Altman, A. K. Dunker, L. Hunter, *et al.* World Scientific Publishing, 2010, pp. 69–79.

[43] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1999.

[44] C. Hammann, A. Luptak, J. Perreault, *et al.* "The ubiquitous hammerhead ribozyme." In: *RNA* 18.5 (2012), pp. 871–885.

[45] J. H. Havgaard, E. Torarinsson, and J. Gorodkin. "Fast Pairwise Structural RNA Alignments by Pruning of the Dynamical Programming Matrix." In: *PLoS Computational Biology* 3.10 (2007), e193.

[46] M. Höchsmann, B. Voß, and R. Giegerich. "Pure Multiple RNA Secondary Structure Alignments: A Progressive Profile Approach." In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 1.1 (2004), pp. 53–62.

[47] I. L. Hofacker, S. H. F. Bernhart, and P. F. Stadler. "Alignment of RNA base pairing probability matrices." In: *Bioinformatics* 20.14 (2004), pp. 2222–2227.

[48] I. L. Hofacker, M. Fekete, and P. F. Stadler. "Secondary structure prediction for aligned RNA sequences." In: *Journal of Molecular Biology* 319.5 (2002), pp. 1059–1066.

[49] I. L. Hofacker, W. Fontana, P. F. Stadler, *et al.* "Fast folding and comparison of RNA secondary structures." In: *Monatshefte für Chemie/Chemical Monthly* 125.2 (1994), pp. 167–188.

[50] I. L. Hofacker, B. Priwitzer, and P. F. Stadler. "Prediction of locally stable RNA secondary structures for genome-wide surveys." In: *Bioinformatics* 20.2 (2004), pp. 186–190.

[51] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. 3. ed., international ed. Addison-Wesley, 2007.

[52] J. Hopcroft and R. Tarjan. "Algorithm 447: Efficient Algorithms for Graph Manipulation." In: *Communications of the ACM* 16.6 (1973), pp. 372–378.

[53] G. Hotz and G. Pitsch. "On Parsing Coupled-Context-Free Languages." In: *Theoretical Computer Science* 161.1&2 (1996), pp. 205–233.

[54] C. J. Hutchins, P. D. Rathjen, A. C. Forster, *et al.* "Self-cleavage of plus and minus RNA transcripts of avocado sunblotch viroid." In: *Nucleic Acids Research* 14.9 (1986), pp. 3627–3640.

[55] S. Inenaga and M. Takeda. "On-Line Linear-Time Construction of Word Suffix Trees." In: *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*. Ed. by M. Lewenstein and G. Valiente. Vol. 4009. Lecture Notes in Computer Science. Springer, 2006, pp. 60–71.

[56] S. Janssen and R. Giegerich. "The RNA shapes studio." In: *Bioinformatics* 31.3 (2015), p. 423.

[57] D. Joseph, J. Meidanis, and P. Tiwari. "Determining DNA Sequence Similarity Using Maximum Independent Set Algorithms for Interval Graphs." In: *Algorithm Theory - SWAT '92, Third Scandinavian Workshop on Algorithm Theory, Helsinki, Finland, July 8-10, 1992, Proceedings*. Ed. by O. Nurmi and E. Ukkonen. Vol. 621. Lecture Notes in Computer Science. Springer, 1992, pp. 326–337.

[58] I. T. Jr and C. Bustamante. "How RNA folds." In: *Journal of Molecular Biology* 293.2 (1999), pp. 271–281.

[59] J. Kärkkäinen and P. Sanders. "Simple Linear Work Suffix Array Construction." In: *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*. Ed. by J. C. M. Baeten, J. K. Lenstra, J. Parrow, *et al.* Vol. 2719. Lecture Notes in Computer Science. Springer, 2003, pp. 943–955.

[60] J. Kärkkäinen and E. Ukkonen. "Sparse Suffix Trees." In: *Computing and Combinatorics, Second Annual International Conference, COCOON '96, Hong Kong, June 17-19, 1996, Proceedings*. Ed. by J. Cai and C. K. Wong. Vol. 1090. Lecture Notes in Computer Science. Springer, 1996, pp. 219–230.

[61] R. M. Karp. "Reducibility Among Combinatorial Problems." In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York*. Ed. by R. E. Miller and J. W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103.

[62] D. K. Kim, J. S. Sim, H. Park, *et al.* "Linear-Time Construction of Suffix Arrays." In: *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*. Ed. by R. A. Baeza-Yates, E. Chávez, and M. Crochemore. Vol. 2676. Lecture Notes in Computer Science. Springer, 2003, pp. 186–199.

[63] A. King, E. Lefkowitz, M. Adams, *et al. Virus Taxonomy: Ninth Report of the International Committee on Taxonomy of Viruses*. Immunology and Microbiology 2011. Elsevier Science, 2011.

[64] P. Ko and S. Aluru. "Space Efficient Linear Time Construction of Suffix Arrays." In: *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*. Ed. by R. A. Baeza-Yates, E. Chávez, and M. Crochemore. Vol. 2676. Lecture Notes in Computer Science. Springer, 2003, pp. 200–210.

[65] S. Kurtz. "Reducing the space requirement of suffix trees." In: *Software: Practice and Experience* 29.13 (1999), pp. 1149–1171.

[66] S. Kurtz. *The Vmatch large scale sequence analysis software*. Last visit: 16.03.2017 22:09 CET. URL: `http://www.vmatch.de/`.

[67] D. Lipman and W. Pearson. "Rapid and sensitive protein similarity searches." In: *Science* 227.4693 (1985), pp. 1435–1441.

[68] R. Lorenz, S. H. Bernhart, C. Höner zu Siederdissen, *et al.* "ViennaRNA Package 2.0." In: *Algorithms for Molecular Biology* 6.1 (2011), p. 26.

[69] B. Löwes, C. Chauve, Y. Ponty, *et al.* "The BRaliBase dent—a tale of benchmark design and interpretation." In: *Briefings in Bioinformatics* 18.2 (2017), p. 306.

[70] T. J. Macke, D. J. Ecker, R. R. Gutell, *et al.* "RNAMotif, an RNA secondary structure definition and search algorithm." In: *Nucleic Acids Research* 29.22 (2001), pp. 4724–4735.

[71] U. Manber and E. W. Myers. "Suffix Arrays: A New Method for On-Line String Searches." In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California.* Ed. by D. S. Johnson. SIAM, 1990, pp. 319–327.

[72] U. Manber and E. W. Myers. "Suffix Arrays: A New Method for On-Line String Searches." In: *SIAM Journal on Computing* 22.5 (1993), pp. 935–948.

[73] D. H. Mathews, M. D. Disney, J. L. Childs, *et al.* "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure." In: *Proceedings of the National Academy of Sciences of the United States of America* 101.19 (2004), pp. 7287–7292.

[74] J. S. McCaskill. "The equilibrium partition function and base pair binding probabilities for RNA secondary structure." In: *Biopolymers* 29.6-7 (1990), pp. 1105–1119.

[75] E. M. McCreight. "A space-economical suffix tree construction algorithm." In: *Journal of the ACM* 23.2 (1976), pp. 262–272.

[76] F. Meyer, S. Kurtz, R. Backofen, *et al.* "Structator: fast index-based search for RNA sequence-structure patterns." In: *BMC Bioinformatics* 12.1 (2011), p. 214.

[77] F. Meyer, S. Kurtz, and M. Beckstette. "Fast online and index-based algorithms for approximate search of RNA sequence-structure patterns." In: *BMC Bioinformatics* 14.1 (2013), p. 226.

[78] J. W. Moon and L. Moser. "On cliques in graphs." In: *Israel Journal of Mathematics* 3.1 (1965), pp. 23–28.

[79] E. P. Nawrocki, S. W. Burge, A. Bateman, *et al.* "Rfam 12.0: updates to the RNA families database." In: *Nucleic Acids Research* 43.D1 (2015), p. D130.

[80] E. P. Nawrocki and S. R. Eddy. "Infernal 1.1: 100-fold faster RNA homology searches." In: *Bioinformatics* 29.22 (2013), pp. 2933–2935.

[81] C. Notredame, D. G. Higgins, and J. Heringa. "T-Coffee: A novel method for fast and accurate multiple sequence alignment." In: *Journal of Molecular Biology* 302.1 (2000), pp. 205–217.

[82] R. Nussinov, G. Pieczenik, J. R. Griggs, *et al.* "Algorithms for Loop Matchings." In: *SIAM Journal on Applied Mathematics* 35.1 (1978), pp. 68–82.

[83] C. Otto, M. Möhl, S. Heyne, *et al.* "ExpaRNA-P: simultaneous exact pattern matching and folding of RNAs." In: *BMC Bioinformatics* 15 (2014), p. 404.

[84] M. de la Peña and I. García-Robles. "Intronic hammerhead ribozymes are ultraconserved in the human genome." In: *EMBO Reports* 11.9 (2010), pp. 711–716.

[85] M. de la Peña and I. García-Robles. "Ubiquitous presence of the hammerhead ribozyme motif along the tree of life." In: *RNA* 16.10 (2010), pp. 1943–1950.

[86] M. de la Peña, I. García-Robles, and A. Cervera. "The Hammerhead Ribozyme: A Long History for a Short RNA." In: *Molecules* 22.1 (2017), p. 78.

[87] N. Philippe, M. Legendre, G. Doutre, *et al.* "Pandoraviruses: Amoeba Viruses with Genomes Up to 2.5 Mb Reaching That of Parasitic Eukaryotes." In: *Science* 341.6143 (2013), pp. 281–286.

[88] G. A. Prody, J. T. Bakos, J. M. Buzayan, *et al.* "Autolytic processing of dimeric plant virus satellite RNA." In: *Science* 231 (1986), pp. 1577–1581.

[89] K. D. Pruitt, T. Tatusova, G. R. Brown, *et al.* "NCBI Reference Sequences (RefSeq): current status, new features and genome annotation policy." In: *Nucleic Acids Research* 40 (2012), pp. D130–D135.

[90] J. Reeder, J. Reeder, and R. Giegerich. "Locomotif: from graphical motif description to RNA motif search." In: *Bioinformatics* 23.13 (2007), pp. i392–i400.

[91] J. Reeder and R. Giegerich. "Consensus shapes: an alternative to the Sankoff algorithm for RNA consensus structure prediction." In: *Bioinformatics* 21.17 (2005), pp. 3516–3523.

[92] J. Reeder and R. Giegerich. "Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics." In: *BMC Bioinformatics* 5.1 (2004), p. 104.

[93] W. L. Ruzzo and J. Gorodkin. "De Novo Discovery of Structured ncRNA Motifs in Genomic Sequences." In: *RNA Sequence, Structure, and Function: Computational and Bioinformatic Methods*. Ed. by J. Gorodkin and W. L. Ruzzo. Humana Press, 2014, pp. 303–318.

[94] R. Sanjuan, M. R. Nebot, N. Chirico, *et al.* "Viral Mutation Rates." In: *Journal of Virology* 84.19 (2010), pp. 9733–9748.

[95] D. Sankoff. "Simultaneous Solution of the RNA Folding, Alignment and Protosequence Problems." In: *SIAM Journal on Applied Mathematics* 45.5 (1985), pp. 810–825.

[96] K. Sato, Y. Kato, T. Akutsu, *et al.* "DAFS: simultaneous aligning and folding of RNA sequences via dual decomposition." In: *Bioinformatics* 28.24 (2012), pp. 3218–3224.

[97] J.-P. Schlüter, J. Reinkensmeier, S. Daschkey, *et al.* "A genome-wide survey of sRNAs in the symbiotic nitrogen-fixing alpha-proteobacterium *Sinorhizobium meliloti*." In: *BMC Genomics* 11.1 (2010).

[98] P. Steffen, B. Voß, M. Rehmsmeier, *et al.* "RNAshapes: an integrated RNA analysis package based on abstract shapes." In: *Bioinformatics* 22.4 (2006), p. 500.

[99] J. D. Thompson, P. Koehl, R. Ripp, *et al.* "BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark." In: *Proteins: Structure, Function, and Bioinformatics* 61.1 (2005), pp. 127–136.

[100] J. D. Thompson, F. Plewniak, and O. Poch. "BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs." In: *Bioinformatics* 15.1 (1999), pp. 87–88.

[101] E. Torarinsson, Z. Yao, E. D. Wiklund, *et al.* "Comparative genomics beyond sequence-based alignments: RNA structures in the ENCODE regions." In: *Genome Research* 18.2 (2008), pp. 242–251.

[102] D. H. Turner and D. H. Mathews. "NNDB: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure." In: *Nucleic Acids Research* 38 (2009), pp. D280–282.

[103] T. Uemura and H. Arimura. "Sparse and Truncated Suffix Trees on Variable-Length Codes." In: *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*. Ed. by R. Giancarlo and G. Manzini. Vol. 6661. Lecture Notes in Computer Science. Springer, 2011, pp. 246–260.

[104] E. Ukkonen. "On-Line Construction of Suffix Trees." In: *Algorithmica* 14.3 (1995), pp. 249–260.

[105] B. Voß, R. Giegerich, and M. Rehmsmeier. "Complete probabilistic analysis of RNA shapes." In: *BMC Biology* 4.1 (2006), p. 5.

[106] S. Washietl, I. L. Hofacker, and P. F. Stadler. "Fast and reliable prediction of noncoding RNAs." In: *Proceedings of the National Academy of Sciences of the United States of America* 102.7 (2005), pp. 2454–2459.

[107] Z. Weinberg, J. E. Barrick, Z. Yao, *et al.* "Identification of 22 candidate structured RNAs in bacteria using the CMfinder comparative genomics pipeline." In: *Nucleic Acids Research* 35.14 (2007), pp. 4809–4819.

[108] P. Weiner. "Linear Pattern Matching Algorithms." In: *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*. IEEE Computer Society, 1973, pp. 1–11.

[109] E. Westhof and P. Auffinger. "RNA Tertiary Structure." In: *Encyclopedia of Analytical Chemistry*. John Wiley & Sons, Ltd, 2006.

[110] S. Will, C. Otto, M. Miladi, *et al.* "SPARSE: Quadratic Time Simultaneous Alignment and Folding of RNAs Without Sequence-Based Heuristics." In: *Bioinformatics* 31.15 (2015), pp. 2489–2496.

[111] S. Will, K. Reiche, I. L. Hofacker, *et al.* "Inferring Noncoding RNA Families and Classes by Means of Genome-Scale Structure-Based Clustering." In: *PLoS Computational Biology* 3.4 (2007), e65.

[112] A. Wilm, I. Mainz, and G. Steger. "An enhanced RNA alignment benchmark for sequence alignment programs." In: *Algorithms for Molecular Biology* 1 (2006), p. 19.

[113] Z. Yao, J. Barrick, Z. Weinberg, *et al.* "A Computational Pipeline for High- Throughput Discovery of cis-Regulatory Noncoding RNA in Prokaryotes." In: *PLOS Computational Biology* 3.7 (2007), e126+.

[114] Z. Yao, J. Barrick, Z. Weinberg, *et al.* "A computational pipeline for high-throughput discovery of cis-regulatory noncoding RNA in prokaryotes." In: *PLoS Computational Biology* 3.7 (2007), e126.

[115] Z. Yao, Z. Weinberg, and W. L. Ruzzo. "CMfinder—a covariance model based RNA motif finding algorithm." In: *Bioinformatics* 22.4 (2006), p. 445.

[116] D. H. Younger. "Recognition and Parsing of Context-Free Languages in Time n^3." In: *Information and Control* 10.2 (1967), pp. 189–208.

[117] M. Zuker and P. Stiegler. "Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information." In: *Nucleic Acids Research* 9.1 (1981), pp. 133–148.