



# Grammar-Based Compression for RNA

Thesis submitted in accordance with the requirements of the University of Liverpool for  
the degree of Doctor in Philosophy by

**Evarista Onokpasa**

April 2025



# Contents

<b>Contents</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Outline . . . . .	2
1.3 Contributions . . . . .	3
<b>2 Preliminaries and Literature</b>	<b>4</b>
2.1 Data Compression . . . . .	4
2.1.1 Kolmogorov Complexity . . . . .	4
2.1.2 Universal Compression . . . . .	5
2.1.3 Arithmetic Coding . . . . .	5
2.2 RNA Representation . . . . .	7
2.2.1 RNA Secondary structures . . . . .	8
2.2.2 Probabilistic Context-Free Grammars . . . . .	9
2.2.3 RNA, Parse Trees and Compression with PCFG . . . . .	9
2.2.4 CYK Parser . . . . .	14
2.2.5 Earley Parser . . . . .	16
2.3 RNA Secondary Structure Prediction Using PCFGs . . . . .	17
2.3.1 PCFG as Probabilistic Models . . . . .	18
2.4 Related Work . . . . .	19
2.4.1 On RNA Compression . . . . .	19
2.4.2 On RNA secondary structure prediction . . . . .	20

2.4.3	On Generating Refined Grammars . . . . .	22
<b>3</b>	<b>Joint Compression of RNA Primary and Secondary structure</b>	<b>23</b>
3.1	Background . . . . .	23
3.2	Using Refined Grammars and Arithmetic Coding . . . . .	24
3.3	Compression Experiments . . . . .	24
3.3.1	Huffman coding vs. Arithmetic coding . . . . .	24
3.3.2	Nullable Grammar vs. Non-Nullable Grammar . . . . .	27
3.3.3	Joint compression of Primary and Secondary Structure using Various Refined Grammars . . . . .	27
3.4	Summary . . . . .	29
<b>4</b>	<b>RNA Secondary Structure Prediction and Compression</b>	<b>30</b>
4.1	Background . . . . .	30
4.1.1	Nussinov Algorithm . . . . .	31
4.2	Secondary Structure Prediction Algorithm Inspired by the Viterbi Algorithm	36
4.3	Compression Ratio Vs. Prediction Quality . . . . .	41
4.4	Summary . . . . .	41
<b>5</b>	<b>Search Algorithms for Grammars</b>	<b>44</b>
5.1	Introduction . . . . .	44
5.2	Stochastic RNA Normal Form for Grammars . . . . .	45
5.3	Exhaustive Search for Grammars . . . . .	46
5.3.1	Distribution of compression ability . . . . .	48
5.3.2	Comparison with expert-curated grammars . . . . .	48
5.4	Conclusion . . . . .	50
<b>6</b>	<b>Heuristics for Autogenerating Grammars</b>	<b>53</b>
6.1	Introduction . . . . .	53
6.2	Random Search . . . . .	54
6.2.1	General Random search . . . . .	54
6.2.2	Nonterminal Based Search . . . . .	56
6.2.3	Rule Type Based Search . . . . .	57
6.3	Local Neighbour Search . . . . .	59
6.4	Random Local Search . . . . .	60
6.5	Snap shots of Results obtained . . . . .	62
6.5.1	Results for Nonterminal Based search . . . . .	62
6.5.2	Results for Rule Type Based Search . . . . .	64
6.5.3	Local Search Results . . . . .	64
6.5.4	Radom Local Search Results . . . . .	64
6.6	Summary of Results . . . . .	66

<b>7</b>	<b>The Software Development Process</b>	<b>69</b>
7.1	Introduction . . . . .	69
7.2	Concept Exploration and Requirements . . . . .	69
7.3	Design and Implementaion . . . . .	70
7.3.1	Design and Implementation to Reproduce RNACompress . . . . .	70
7.3.2	Initial Design for AC-Compressor . . . . .	70
7.3.3	Modularising the Program and the Use of Earley Parser . . . . .	72
7.3.4	RNA Secondary Structure Prediction . . . . .	73
7.3.5	Data Package for Conformity of Dataset . . . . .	73
7.3.6	An Earley Style Implementation for Prediction . . . . .	74
7.3.7	Replacing Earley with SRF parser . . . . .	74
7.4	Verification, Validation and Documentation . . . . .	74
7.5	Automatically Generated Grammars and Future Development . . . . .	76
<b>8</b>	<b>Conclusion and Future Work</b>	<b>77</b>
8.1	Summary of Findings . . . . .	77
8.1.1	Joint Compression of RNA Primary and Secondary Structure . . . . .	77
8.1.2	RNA Secondary Structure Prediction and Compression . . . . .	77
8.1.3	Automatically generated Grammars and Heuristics . . . . .	78
8.2	Lessons Learnt . . . . .	78
8.3	Future Work . . . . .	78
	<b>References</b>	<b>80</b>
A	Human Expert Grammars . . . . .	86
B	9 Best Grammars from Local Search Heuristics . . . . .	90

# Acknowledgement

I am thankful to God, my loving family, my brilliant supervisors and my supportive friends.

# Abstract

In bioinformatics, the benefits of compression is twofold. It (a) brings practical improvements for storage requirements and sometimes computational efficiency at the same time, it is also (b) used as a direct means to generate biological insight. In this research, we improve on the joint-RNA (ribonucleic acid) compression in [32] using arithmetic coding and refined Probabilistic Context Free Grammars(PCFG).

Given an RNA primary sequence and a refined PCFG grammar a secondary structure can be predicted and its accuracy calculated, with this we establish the co-relation between prediction quality and compression quality of a grammar. Observing this remarkable property of RNA grammars, we launched an automatic search for better grammars in terms of compression and prediction quality and obtained interesting results.

Finally we fine-tune the automatic grammar search with the use of heuristics to obtain grammars with better compression and prediction.



# List of Figures

2.1	Sample RNA . . . . .	8
2.2	Sample RNA, which contains loops, helices and pseudoknots (dotted lines link bases with bonds which form pseudoknots) . . . . .	9
2.3	Parse Tree representation of Example 2.2.1 using Grammar G. . . . .	11
2.4	Earley Parser . . . . .	16
3.1	Comparing RNACompress and AcCompressor . . . . .	25
3.2	Comparing RNACompress and AC-Compressor with a Scatter Plot . . . . .	26
3.3	Compression using Various Grammars . . . . .	28
4.1	Example on Nussinov Algorithm . . . . .	35
4.2	Region shaded light grey in matrix shows the traceback cells, on the right the folding over of the RNA is shown . . . . .	36
4.3	Compression vs. Prediction Scatter plot . . . . .	42
4.4	Compression vs. Prediction, with G6 on Benchmark . . . . .	43
5.1	Plot for compression ratio of Static-model vs. Adaptive-model for Autogenerated Grammars . . . . .	51
5.2	Histogram of the normalized average compressed size (in bits per base) for all grammars from Figure 5.1 on the 10% subsample of the benchmark dataset from Dowell and Eddy [8] using the adaptive (left) resp. static (right) rule-probability model. . . . .	52
5.3	Newly identified grammars; $G_{k,r}^*$ is the best grammar with $k$ NTs (Nonterminals) and $r$ rules from exhaustive search. . . . .	52
6.1	Results for Nonterminal based heuristic experiments . . . . .	63
6.2	Results for Rule Type based heuristic experiments. . . . .	65
6.3	Results for Experiments using local search. . . . .	67
6.4	Results for Random Local search . . . . .	68

---

7.1	Flowchart for the design of Liu et al's RNAcompress. Dotted lines link associated classes with process . . . . .	71
7.2	Flowchart for the encoding process of AC-Compressor. Dotted lines link associated classes with process . . . . .	72
7.3	Package Diagram shows the AC-Compressor Package, with subpackages. Dotted arrows show dependency . . . . .	75

# List of Tables

2.1	Probability distribution for Grammar G. . . . .	12
5.1	Exhaustively Generated Grammars . . . . .	47
5.2	Human expert grammars and Autogenerated grammars . . . . .	49
6.1	Grammar $G_{6,10}^\dagger$ . Without the grey rules (dead end rules) gives the best grammar found so far $G_{6,10}'$ . . . . .	55
6.2	Comparing Human Expert Grammars and Autogenerated Grammars . . . . .	55



# Chapter 1

## Introduction

In this research, we studied the *joint compression* of RNA (ribonucleic acid) primary and secondary structure data using Probabilistic Context-Free Grammars and improved on the Liu et al.'s work on joint compression of RNA sequence and structure data [32] (details in Chapter 3). RNA molecules, short for ribonucleic acid, are a single-stranded relative of DNA. They consist of a primary structure, i.e. a chain (backbone) of bases (adenine (A), cytosine (C), guanine (G), or uracil (U)), modeled as a string in  $\{A, C, G, U\}^*$  and a secondary structure, which shows the secondary bonds which may exist in the RNA string, and which cause the (linear) RNA molecule to fold in space into a specific shape. This folding over is referred to as the secondary structure (see Section 2.2.1).

Furthermore, we focus on predicting the secondary structure of RNAs when the primary sequence is known, using Probabilistic Context-Free Grammars. The secondary structure plays a crucial role in the biological function of non-coding RNA molecules and is of significant interest to biologists. We also investigated the correlation between secondary structure prediction and compression using various human-expert curated grammars. Upon discovering that grammars with better compression also produce better predictions, we were motivated to search for improved grammars that could further enhance both compression and prediction. This exploration ultimately led to the discovery of grammars with remarkable compression capabilities!

## 1.1 Motivation

With the size of data sets ever growing, memory usage of computations is becoming a key bottleneck that is only partly mitigated by improvements in computer hardware. Approaches that economize on the used amount of memory, both for storage (classic compression) as well as during computation (space-efficient data structures, computation over compressed data), are therefore much sought after, thus we produced the joint-rna compressor (also known as Arithmetic Coding compressor (AC-Compressor)) for RNA primary and secondary structure data compression.

Determining RNA secondary structure requires expensive, techniques like X-ray crystallography [48]. This has driven interest in the prediction of secondary structures in RNA. We investigated the correlation between secondary structure prediction and RNA data compression using various refined grammars and identified a direct relationship between compression efficiency and predictive accuracy. Discovering that more refined grammars improve compression and better predict the secondary structures of RNA sequences naturally led to the next step: developing an automatic generator for improved grammars.

## 1.2 Outline

This thesis begins with the Introduction in Chapter 1. Chapter 2 presents a review of the literature on RNA secondary structure compression and prediction, along with definitions of Kolmogorov complexity, universal compression, arithmetic coding, parsing and RNA representation. Chapter 3 details the work undertaken to improve compression through the use of arithmetic coding and refined grammars. This is followed in Chapter 4, by the establishment of a correlation between compression and the prediction of secondary structure of RNA. Chapter 5 focuses on the systematic search for more refined grammars and the discovery of newly developed grammars. Chapter 6 gives details of the heuristics designed to enhance the systematic search for better grammars, particularly in terms of RNA secondary structure prediction. Chapter 7 describes the process of developing the software that implements the concepts outlined in Chapters 3 to 6. Chapter 8 provides a summary of the entire work.

## 1.3 Contributions

In this research, we successfully improved the compression of RNA data using arithmetic coding and refined PCFG grammars. We established a clear correlation between RNA secondary structure prediction and compression. Additionally, this work introduced a new normal form, the Stochastic RNA Normal Form (SRF), which is analogous to the Chomsky Normal Form. The SRF simplifies parsing and serves as the foundation for the systematic generation of grammars in the search for improved grammars. Furthermore, we explored heuristic approaches to enhance the search for better grammars. The results of this research work have been published. The “List of Publications” below, shows the list of papers which publish our work.

### List of Publications

- E. Onokpasa, S. Wild, and P. W. H. Wong. “RNA secondary structures: from ab initio prediction to better compression, and back”. In Data Compression Conference (DCC), pages 278–287, 2023.[39].

The results presented in this publication, which demonstrate the correlation between compression and prediction, are discussed in detail in Section 4.3 of this thesis.

- E. Onokpasa, S. Wild and P. W. H. Wong, “Towards Optimal Grammars for RNA Structures”. 2024 Data Compression Conference (DCC), Snowbird, UT, USA, 2024, pp. 332-341.[40].

Sections 5.2 and 5.3 of this Thesis have been directly adapted from this publication.

- E. Onokpasa, S. Wild and P. W.H. Wong, “RNA Secondary Structures: Inherent Information Content, Compression, and Prediction via Probabilistic Context-Free Grammar”[submitted to BMC Bioinformatics].

The compression experiments, its results in Section 3.3 and the SRF parser definition in Section 5.2 with its elaborate example are taken from this submitted publication.

## Chapter 2

# Preliminaries and Literature

To achieve the objectives of this research, a background study was conducted on compression techniques, theories in compression, grammars (both context-free ambiguous and unambiguous), and parsers. An understanding of tree structures and graphs was also essential, as RNA secondary structures can be represented as trees. Additionally, arithmetic encoding, a key compression technique, was applied in this research to the compression of RNA data.

### 2.1 Data Compression

This research applies several fundamental concepts, including Kolmogorov complexity, universal compression, arithmetic coding, parsing and RNA representation. An efficient parser library [7] was incorporated into the implementation.

#### 2.1.1 Kolmogorov Complexity

The Kolmogorov complexity [31] of an object is the length of a shortest computer program (in a predetermined programming language) that produces the object as output. It is a measure of the information content of the object and is also known as the descriptive complexity of a string. It provides a lower bound for compression.

### 2.1.2 Universal Compression

Universal compression seeks to produce compression schemes that do not require any prior knowledge of the distribution of the data source, e.g. the Lempel-Ziv compression. The Lempel-Ziv compression requires “a rule for parsing strings of symbols from a finite alphabet  $A$  into substrings, or words, whose lengths do not exceed a prescribed integer  $L$ , and a coding scheme which maps these substrings sequentially into uniquely decipherable codewords of fixed length  $L$ , over the same alphabet  $A$ ” [54].

### 2.1.3 Arithmetic Coding

Arithmetic coding [51] “assigns one codeword to each possible data set. . . with the codeword drawn from the interval  $[0, 1)$ ”. Each codeword uniquely identifies the subinterval in which the probabilities of occurrence of the given data set can be found. Unlike the Huffman encoding which spits out codewords per event in a dataset, arithmetic encoding produces one codeword at the end of all the events in a given data set. An arithmetic encoder “must work in conjunction with a modeler that estimates the probability of each possible event at each point in the coding [. . .]. The models can be **adaptive** (dynamically estimating the probability of each event based on all events that precede it), semi-adaptive (using a preliminary pass of the input file to gather statistics), or **nonadaptive** (using fixed or **static** probabilities for all files)” [24]. In arithmetic coding, every event  $A$  from a given data set is assigned a probability  $p$  and an order or position  $r$  relative to other events in the given set of events. A sequence of events is encoded in arithmetic coding by subdividing the half open interval  $[0, 1)$  successively in proportion to the probability  $p$  of the event, in position  $r$  relative to the other events. The final subinterval say  $[a, b)$  is encoded in arithmetic coding by determining the smallest  $m$  such that  $\exists x \in \mathbb{N}_0$  with

$$\left[ \frac{x}{2^m}, \frac{x+1}{2^m} \right) \subseteq [a, b)$$

we encode  $x$  using at least

$$m = \left\lceil \log_2 \left( \frac{1}{0.02} \right) \right\rceil + 2$$

The Example 2.1.1 illustrates the arithmetic encoding using a static model:

**Example 2.1.1.** Assume we have a set of exercising activities, comprising of events {jump, squat} and event {end} (which indicates the end of the exercising activity). Suppose the

events {jump, squat, end} occur with probabilities 0.4, 0.5, 0.1 respectively. This order is followed in the subdivision selection (i.e. for every half open interval  $[a, b)$ ,  $[a, b)$  is subdivided in proportion to the probabilities of the events in the order {jump, squat, end} to give  $[a, a + 0.4 * (b - a))$ ,  $[a + 0.4 * (b - a), a + 0.9 * (b - a))$ ,  $[a + 0.9 * (b - a), b)$  respectively). If we have a SIMPLE exercise routine with event sequence

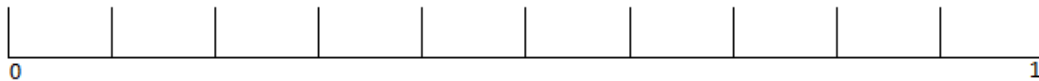
$$R = \text{squat, jump, end,}$$

this will be coded in AC as follows:

1. Store the probabilities for each event:  $p(\text{jump}) = 0.4$ ,  $p(\text{squat}) = 0.5$ ,  $p(\text{end}) = 0.1$ 
  - (a) Assign an order for subinterval selection.
    - For every subinterval  $[a, b)$ ,  $p(\text{jump})$  is assigned the first subdivision of the interval  $[a, b)$ , with  $\text{length} = 0.4 * (b - a)$ . This subdivision equals  $[a, a + 0.4 * (b - a))$
    - $p(\text{squat})$  takes up the 2<sup>nd</sup> subdivision of  $[a, b)$  with  $\text{length} = 0.5 * (b - a)$  and we obtain the interval  $[a + 0.4 * (b - a), a + 0.9 * (b - a))$
    - $p(\text{end})$  having  $\text{length} = 0.1 * (b - a)$  takes the last subdivision  $[a + 0.9 * (b - a), b)$
  - (b) Read each event from left to right and start subdividing

**Initial Interval**  $[0, 1)$

length: 1



First Event:  $\rightarrow$  squat,  $p(\text{squat}) = 0.5$  Index: 2<sup>nd</sup>

**New Subinterval is**  $[0 + 0.4 * (1 - 0), 0 + 0.9 * (1 - 0)) = [0.4, 0.9)$  length: 0.5



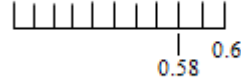
Next Event:  $\rightarrow$  jump,  $p(\text{jump}) = 0.4$  Index: 1<sup>st</sup>

**New Subinterval is**  $[0.4, 0.4 + 0.4 * (0.9 - 0.4)) = [0.4, 0.6)$  length: 0.2



Final Event:  $\rightarrow$  end,  $p(\text{end}) = 0.1$  Index: 3<sup>rd</sup>

**Final Subinterval**  $[0.4 + 0.9 * (0.6 - 0.4), 0.6) = [0.58, 0.6)$  length: 0.02



- The final interval  $[0.58, 0.6)$  is represented with a single codeword, by determining the smallest  $m$  such that  $\exists x \in \mathbb{N}_0$  with

$$\left[ \frac{x}{2^m}, \frac{x+1}{2^m} \right) \subseteq [0.58, 0.6)$$

we obtain the interval

$$\left[ \frac{150}{2^8}, \frac{151}{2^8} \right) \subseteq [0.58, 0.6)$$

using at least

$$\left\lceil \log_2 \left( \frac{1}{0.02} \right) \right\rceil + 2$$

digits of precision (from Shannon's Entropy [47]) the lower bound  $\frac{150}{2^8}$  is converted to binary to give 0.10001100. Thus the code word representation for  $[0.58, 0.6)$  is 0.10001100.

## 2.2 RNA Representation

RNAs have a primary structure and secondary structure. The primary structure simply shows the sequence, chain (or order) of nucleotides (adenine (A), cytosine (C), guanine (G), and uracil (U)) in the RNA string, e.g., *GGUCCCACC*. The secondary structure shows the chain of nucleotides as well as the folding or secondary bonding which may exist as shown in Figure 2.1.

The dot bracket notation (introduced by [23]) is widely used to represent RNA secondary structure. Brackets for bonds or base pairs, dots for unpaired bases. The dot bracket notation for the example in Figure 2.1 is written as:

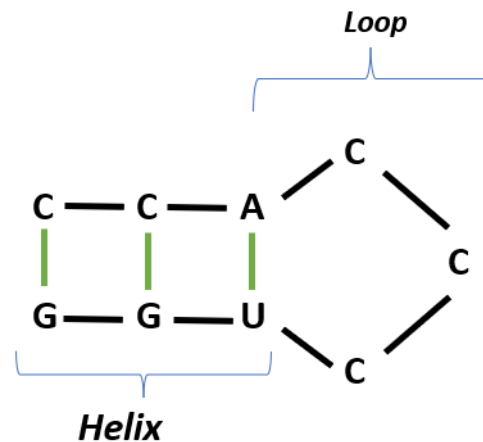


Figure 2.1: Sample RNA, secondary bonds (indicated by green lines), helical and loop regions.

```

GGUCCCACC
(((...)))

```

### 2.2.1 RNA Secondary structures

Based on secondary structure formation, RNAs can be classed into 2: pseudoknotted and non-pseudoknotted RNAs. A **pseudoknot** is defined as a structure containing base-pairs which violate the nesting convention. Pseudoknots are often found in functionally important regions of RNA, where “they play a key role in regulating gene expression” [1]. Non-pseudoknotted RNAs obey a nesting convention: If the bases of an RNA primary sequence are indexed from 1 (called the 5' terminus) to  $n$  (called the 3' terminus), such that any RNA sequence  $r$  of length  $n$  can be written as  $r_1...r_n$ . For any two base-pairs  $i, j$  and  $k, l$  (where  $i < j, k < l$  and  $i < k$ ), either  $i < k < l < j$  or  $i < j < k < l$ . Other physical features in RNA secondary structure are loops, helixes and bulges. **Loops** are sequence of unpaired bases which are nested within paired bases. Hairpin loops are loops with 1 unpaired base and **bulges** are loops consisting of more than one unpaired base. A **helix** is a group of two or more consecutive base pairs. Figure 2.2 illustrates these features.

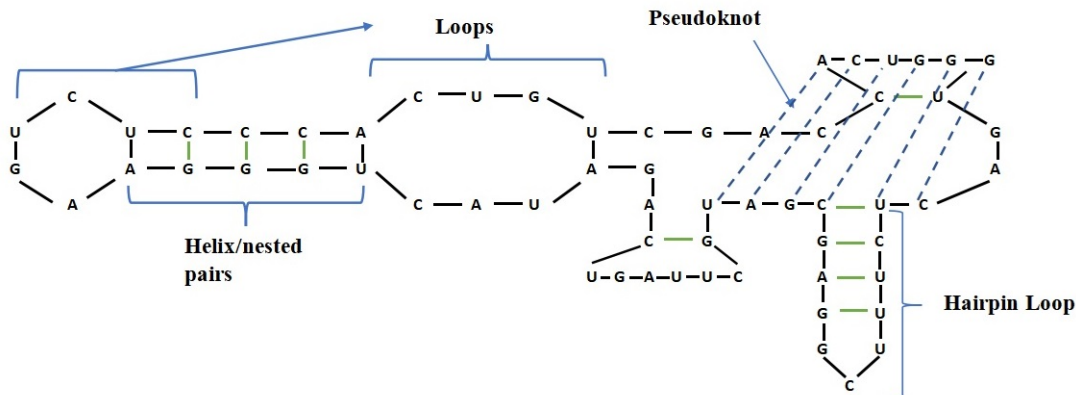


Figure 2.2: Sample RNA, which contains loops, helices and pseudoknots (dotted lines link bases with bonds which form pseudoknots)

### 2.2.2 Probabilistic Context-Free Grammars

A **Context-Free Grammar** is a 4-tuple  $(I, T, R, S)$ , where  $I$  is a finite set of nonterminals,  $T$  is a finite set of terminals,  $R$  is a finite set of rules  $R \subseteq I \times (I \cup T)$  and  $S \in I$  is the start symbol. Following the definitions from [13], a weighted context-free grammar (WCFG) is a 5-tuple  $G = (I, T, R, S, W)$ , where  $(I, T, R, S)$  is a Context-Free grammar (CFG) and  $W : R \rightarrow \mathbb{R}^+$  is a mapping such that each rule  $f \in R$  is equipped with a weight  $w_f := W(f)$ .

If  $G$  is a WCFG, then  $G$  is a **Probabilistic Context-Free Grammar** (PCFG) iff the following additional restrictions hold:

1. For all  $f \in R$  we have  $W(f) \in (0, 1]$  which means the weights are probabilities.
2. The probabilities are chosen in such a way that for all  $A \in I$ , we have

$$\sum_{f \in R, Q(f)=A} w_f = 1,$$

where  $Q(f)$  denotes the premise of the production.

Thus, for any PCFG,  $G = (I, T, R, S, W)$ , the mapping  $W : R \rightarrow [0, 1]$  provides a probability distribution on the production rules.

### 2.2.3 RNA, Parse Trees and Compression with PCFG

In our RNA (primary and secondary structure) compression, we use a rule for parsing the RNA string, this rule is defined in the chosen grammar. We illustrate this using the unique

grammar presented in Liu et al's 2008 work [32] with some slight modifications (here, the terminals are expressed as pairs of characters).

$G = (I, T, R, S)$  with

$I = \{S, L\}$

$T = \{[\overset{A}{\underset{C}{\downarrow}}], [\overset{C}{\underset{G}{\downarrow}}], [\overset{G}{\underset{U}{\downarrow}}], [\overset{U}{\underset{A}{\downarrow}}], [\overset{A}{\underset{C}{\downarrow}}], [\overset{C}{\underset{G}{\downarrow}}], [\overset{G}{\underset{U}{\downarrow}}], [\overset{A}{\underset{\bullet}{\downarrow}}], [\overset{C}{\underset{\bullet}{\downarrow}}], [\overset{G}{\underset{\bullet}{\downarrow}}], [\overset{U}{\underset{\bullet}{\downarrow}}]\}$

$R = \left\{ S \rightarrow LS, S \rightarrow \varepsilon, \right.$   
 $L \rightarrow [\overset{A}{\underset{C}{\downarrow}}]S[\overset{U}{\underset{G}{\downarrow}}], L \rightarrow [\overset{U}{\underset{G}{\downarrow}}]S[\overset{A}{\underset{C}{\downarrow}}], L \rightarrow [\overset{C}{\underset{G}{\downarrow}}]S[\overset{G}{\underset{U}{\downarrow}}], L \rightarrow [\overset{G}{\underset{U}{\downarrow}}]S[\overset{C}{\underset{G}{\downarrow}}], L \rightarrow [\overset{U}{\underset{G}{\downarrow}}]S[\overset{G}{\underset{U}{\downarrow}}], L \rightarrow [\overset{G}{\underset{U}{\downarrow}}]S[\overset{U}{\underset{G}{\downarrow}}],$   
 $L \rightarrow [\overset{A}{\underset{\bullet}{\downarrow}}], L \rightarrow [\overset{U}{\underset{\bullet}{\downarrow}}], L \rightarrow [\overset{C}{\underset{\bullet}{\downarrow}}], L \rightarrow [\overset{G}{\underset{\bullet}{\downarrow}}] \left. \right\}$

Here  $[\overset{A}{\underset{C}{\downarrow}}]$  etc. are terminal symbols consisting of pairs of characters. The first character is a symbol from the primary sequence which consists of 4 different bases (A, C, G, U). The second character is a symbol from the secondary structure.  $\varepsilon$  is the empty string.

**Example 2.2.1.** In this example we use this simple made up RNA shown below

GAC  
(.)

This RNA can be written as pairs of characters as follows:

$[\overset{G}{\underset{C}{\downarrow}}] [\overset{A}{\underset{\bullet}{\downarrow}}] [\overset{C}{\underset{G}{\downarrow}}]$

Using the rules of  $G$  the derivation for the sample RNA is:

$S \Rightarrow LS \Rightarrow [\overset{G}{\underset{C}{\downarrow}}]S[\overset{C}{\underset{G}{\downarrow}}]S \Rightarrow [\overset{G}{\underset{C}{\downarrow}}]LS[\overset{C}{\underset{G}{\downarrow}}]S \Rightarrow [\overset{G}{\underset{C}{\downarrow}}] [\overset{A}{\underset{\bullet}{\downarrow}}]S[\overset{C}{\underset{G}{\downarrow}}]S \Rightarrow [\overset{G}{\underset{C}{\downarrow}}] [\overset{A}{\underset{\bullet}{\downarrow}}] \varepsilon [\overset{C}{\underset{G}{\downarrow}}]S \Rightarrow [\overset{G}{\underset{C}{\downarrow}}] [\overset{A}{\underset{\bullet}{\downarrow}}] [\overset{C}{\underset{G}{\downarrow}}] \varepsilon = [\overset{G}{\underset{C}{\downarrow}}] [\overset{A}{\underset{\bullet}{\downarrow}}] [\overset{C}{\underset{G}{\downarrow}}]$

The parse tree representation of Example 2.2.1 is shown in Figure 2.3.

Next, we show how to compress RNA primary and secondary structure data using a PCFG. For illustrative purposes, we assign the probabilities to the production rules in  $G$  as shown in Table 2.1.

The rules are applied in the order:

$S \rightarrow LS, \quad L \rightarrow [\overset{G}{\underset{C}{\downarrow}}]S[\overset{C}{\underset{G}{\downarrow}}], \quad S \rightarrow LS, \quad L \rightarrow [\overset{A}{\underset{\bullet}{\downarrow}}], \quad S \rightarrow \varepsilon, \quad S \rightarrow \varepsilon$

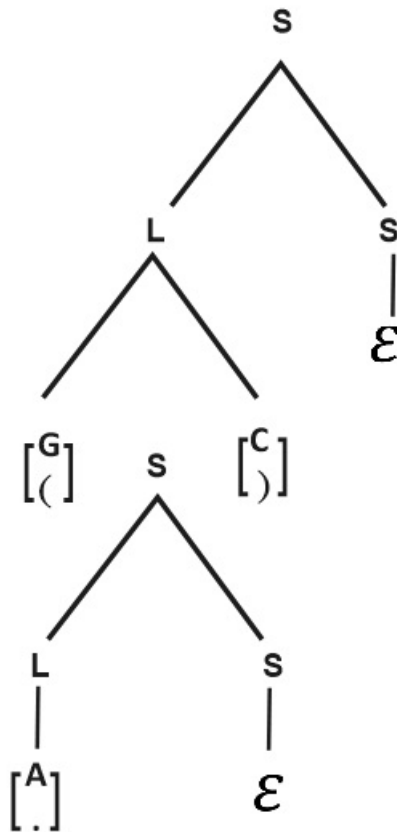


Figure 2.3: Parse Tree representation of Example 2.2.1 using Grammar G.

Rule	Probability
$S \rightarrow LS$	0.65
$S \rightarrow \varepsilon$	0.35
$L \rightarrow \begin{bmatrix} A \\ \downarrow \end{bmatrix} S \begin{bmatrix} U \\ \downarrow \end{bmatrix}$	0.05
$L \rightarrow \begin{bmatrix} U \\ \downarrow \end{bmatrix} S \begin{bmatrix} A \\ \downarrow \end{bmatrix}$	0.15
$L \rightarrow \begin{bmatrix} C \\ \downarrow \end{bmatrix} S \begin{bmatrix} G \\ \downarrow \end{bmatrix}$	0.10
$L \rightarrow \begin{bmatrix} G \\ \downarrow \end{bmatrix} S \begin{bmatrix} C \\ \downarrow \end{bmatrix}$	0.05
$L \rightarrow \begin{bmatrix} U \\ \downarrow \end{bmatrix} S \begin{bmatrix} G \\ \downarrow \end{bmatrix}$	0.05
$L \rightarrow \begin{bmatrix} G \\ \downarrow \end{bmatrix} S \begin{bmatrix} U \\ \downarrow \end{bmatrix}$	0.10
$L \rightarrow \begin{bmatrix} A \\ \bullet \end{bmatrix}$	0.10
$L \rightarrow \begin{bmatrix} U \\ \bullet \end{bmatrix}$	0.15
$L \rightarrow \begin{bmatrix} C \\ \bullet \end{bmatrix}$	0.10
$L \rightarrow \begin{bmatrix} G \\ \bullet \end{bmatrix}$	0.15

Table 2.1: Probability distribution for Grammar G.

where the right hand side for each rule, replaces the nonterminal on the left. Looking up the probabilities from Table 2.1, the probabilities obtained are:

0.65, 0.05, 0.65, 0.10, 0.35 and 0.35.

The next step is the arithmetic coding step.

- The first rule  $S \rightarrow LS$ , with probability 0.65 takes the first division of the initial interval  $[0, 1)$  which is  $[0, 0.65 * (1 - 0)) = [0, 0.65)$ .
- Next comes  $L \rightarrow \begin{bmatrix} G \\ C \end{bmatrix} S \begin{bmatrix} C \\ S \end{bmatrix}$  with probability 0.05 and index position 4. So it takes up the 4th subinterval of  $[0, 0.65)$ . To get the start point of this subinterval, a sum is taken of the probabilities of rules with same left handside which precede index 4 i.e.  $p(L \rightarrow \begin{bmatrix} A \\ C \end{bmatrix} S \begin{bmatrix} U \\ S \end{bmatrix}) + p(L \rightarrow \begin{bmatrix} U \\ C \end{bmatrix} S \begin{bmatrix} A \\ S \end{bmatrix}) + p(L \rightarrow \begin{bmatrix} C \\ C \end{bmatrix} S \begin{bmatrix} G \\ S \end{bmatrix}) = 0.05 + 0.15 + 0.10 = 0.3$ . The product of this sum (0.3) and the length of the current subinterval ( $[0, 0.65)$ ) gives the start point ( $[0.3 * 0.65, \dots) = [0.195, \dots)$  of the new subinterval for the rule  $L \rightarrow \begin{bmatrix} G \\ C \end{bmatrix} S \begin{bmatrix} C \\ S \end{bmatrix}$ . To get the end point, we simply add the product of  $p(L \rightarrow \begin{bmatrix} G \\ C \end{bmatrix} S \begin{bmatrix} C \\ S \end{bmatrix})$  and the length (0.65) (i.e.  $0.05 * 0.65 = 0.0325$ ) of the current subinterval  $[0, 0.65)$  to the start point of the new subinterval  $[0.195, \dots)$ , to arrive at  $[0.1975, 0.2275)$
- Again  $S \rightarrow LS$  follows next with probability of 0.65.  $[0.1975, 0.2275)$  is subdivided in the ratio 65:35 and the first subdivision,  $[0.1975, 0.218625)$  with length 0.021125 is obtained.
- $L \rightarrow \begin{bmatrix} A \\ \bullet \end{bmatrix}$  is the next rule in the derivation with probability 0.10. From Table 2.1,  $L \rightarrow \begin{bmatrix} A \\ \bullet \end{bmatrix}$  is placed at the 7th rule, in the group of rules with left handside  $L$ . Probabilities of all rules preceding  $L \rightarrow \begin{bmatrix} A \\ \bullet \end{bmatrix}$  are summed up to give 0.5. The current subinterval is subdivided into parts: 0.5, 0.1 and 0.4, with the 0.1 corresponding to the probability of rule  $L \rightarrow \begin{bmatrix} A \\ \bullet \end{bmatrix}$ . This gives the subinterval  $[0.2080625, 0.210175)$  with length 0.0021125.
- The last two rules in the derivation  $S \rightarrow \varepsilon$  and  $S \rightarrow \varepsilon$  with probabilities 0.35 and 0.35 further subdivide  $[0.2080625, 0.210175)$  to obtain  $[0.209435625, 0.210175)$  with length 0.000739375 and the final subinterval  $[0.20991621875, 0.210175)$  with length 0.00025878125.
- To encode the final subinterval  $[0.20991621875, 0.210175)$ ,  $m$  is determined

$$m = \left\lceil \log_2 \left( \frac{1}{0.00025878125} \right) \right\rceil + 2 = 14$$

next task is to find  $x$  such that

$$\left[ \frac{x}{2^{14}}, \frac{x+1}{2^{14}} \right) \subseteq [0.20991621875, 0.210175)$$

we obtain the interval

$$\left[ \frac{3440}{2^{14}}, \frac{3441}{2^{14}} \right) \subseteq [0.20991621875, 0.210175)$$

coded in binary as 0.00110101110000

### 2.2.4 CYK Parser

The **CYK** parser is attributed to J. Cocke, D.H. Younger, and T. Kasami, interestingly they independently produced variations of the parsing method [19]; The original work by Younger can be found in [53] and Sakai's description in [27]. The standard form of the CYK algorithm works with a context free grammar rendered in **Chomsky Normal Form**. A context-free grammar  $G = (I, T, R, S)$  is in Chomsky Normal Form, if all its rules are of the form:

- **Binary rule:**  $A \rightarrow BC$ , where  $A, B, C \in I$  (nonterminal symbols) and neither  $B$  nor  $C$  is the start symbol.
- **Terminal rule:**  $A \rightarrow a$ , where  $A \in V$  and  $a \in T$  (a terminal symbol).

The algorithm in Listing 1 shows the steps involved in parsing a string using a CYK parser. To parse a string  $L[1, n]$  using the CYK parser, the algorithm processes spans of increasing lengths and partitions them recursively to build the parse table. Lines 8 -12 show how the parse table is built for unit productions using the rules of type  $R[v] \rightarrow a[s]$ . In the array of booleans,  $P[][][]$  (line 6) an entry  $P[1][s][a]$  is set to true (line 22), if there is a rule  $R[a] \rightarrow R[b] R[c]$  and a split of its length 1 into  $p$  and  $1-p$ , for which the boolean arrays  $(P[p][s][b], P[1-p][s+p][c])$  for the splits have been evaluated to be true. The recursion ends at  $P[n][1][1]$ . If  $P[n][1][1]$  is true the parse tree is returned by taking a backtrack on the array  $back[][][]$ .

The CYK algorithm has a runtime complexity of  $O(n^3 \cdot |G|)$ , where:

- $n$ : Length of the input string.

- $|G|$ : Number of grammar rules in the grammar  $G$ .

## Listing 1: The CYK Parsing Algorithm

```

1 // Assume the input string I consists of n characters: a1 a2 ... an
2 // Assume the grammar contains r nonterminal symbols R1, R2, ..., Rr, with R1 ←
   as the start symbol

5 bool P[n][n][r]; // Boolean array initialized to false
6 list Back[n][n][r]; // Array of lists to store back-pointers, initially empty

8 // Step 1: Initialize P for unit productions
9 for (int s = 1; s <= n; s++) {
10     for each unit production (R[v] -> a[s]) {
11         P[1][s][v] = true;
12     }
13 }

15 // Step 2: Fill the DP table
16 for (int l = 2; l <= n; l++) { // Length of the span
17     for (int s = 1; s <= n - l + 1; s++) { // Start of the span
18         for (int p = 1; p <= l - 1; p++) { // Partition of the span
19             for each production (R[a] -> R[b] R[c]) {
20                 if (P[p][s][b] && P[l - p][s + p][c]) {
21                     P[l][s][a] = true;
22                     Back[l][s][a].append(<p, b, c>);
23                 }
24             }
25         }
26     }
27 }

29 // Step 3: Check if the input string is in the language
30 if (P[n][1][1]) { // If start symbol can generate the full string
31     return back; // The parse tree can be reconstructed using back-pointers
32 } else {
33     return "Not a member of the language";
34 }

```

### 2.2.5 Earley Parser

The Earley Parsing algorithm [10] has the ability to process any Probabilistic Context Free Grammars (PCFG) and determine whether a string belongs to the language of the grammar or not. To describe the steps in the Earley parser we use the description from [15] shown here in Figure 2.4. Earley parser uses dotted rules to indicate progress in recognizing the

#### Item Form

- **Normal items:**  $(i \ j, A \rightarrow \alpha \bullet \beta)$  satisfying:
  - $1 \leq i \leq j \leq n + 1$ ,  $\alpha$  and  $\beta$  are possibly empty strings of terminals and nonterminals
- **Rule existence items:** Rule  $[A \rightarrow \alpha\beta]$ :
  - $A \rightarrow \alpha\beta$  is a rule of the PCFG  $G$ , These items exist from the start and have as their value the probability of the rule  $P(A \rightarrow \alpha\beta)$ .
- **Goal Item**
  - $(1 \ n + 1, S' \rightarrow \lambda \bullet)$   $S$  is the start nonterminal in  $G$ ,  $\lambda \bullet$  is the fully recognized right hand side.

#### Inference Rules

- **Start item:**

$$\frac{}{1 \ 1, \ S' \rightarrow \bullet S}$$

- **Prediction**

$$\frac{\text{Rule}[B \rightarrow \gamma]}{(j \ j, \ B \rightarrow \bullet \gamma)} [i, A \rightarrow \alpha \bullet B \beta, j]$$

- **Scanning:**

$$\frac{(i \ j - 1, \ A \rightarrow \alpha \bullet w_{j-1} \beta)}{(i \ j, \ A \rightarrow \alpha w_{j-1} \bullet \beta)}$$

- **Completion:**

$$\frac{(i \ r, \ A \rightarrow \alpha \bullet B \beta) \ (r \ j, \ B \rightarrow \beta \bullet)}{(i \ j, \ A \rightarrow \alpha B \bullet \beta)}$$

Figure 2.4: Earley Parser

input e.g.  $A \rightarrow \alpha \bullet \beta$ , the position of the  $\bullet$  implies that tokens before the dot,  $\alpha$  have been seen by the parser, tokens after the dot  $\beta$  are yet to be seen,  $\alpha$  and  $\beta$  are either terminals or nonterminals. Item forms and inference rules are also defined in Figure 2.4. The normal item form

$$(i \ j, A \rightarrow \alpha \bullet \beta)$$

satisfying:

$1 \leq i \leq j \leq n + 1$ ,  $\alpha$  and  $\beta$  are possibly empty strings of terminals and nonterminals.  $i$  is the starting point of the rule in the input and  $j$  is the current position in the input. The inference rule, *prediction*, expands nonterminals after the dot, *scanning* matches terminals and makes progress, *completion* verifies a nonterminal has been fully recognized. Back-tracing rebuilds the parse tree by working backward from completed rules. For a PCFG  $G = (I, T, R, S, P)$  where  $I$  is the finite set of nonterminals,  $T$  is the finite set of terminals,  $R$  is the set of production rules and  $S$  is the start symbol (we add a new start symbol for the Earley parser,  $S'$ , and a single rule  $S' \rightarrow S$ ),  $P : R \rightarrow [0, 1]$  provides a probability distribution on the production rules, a word  $w \in T^n$  can be found in the language of  $G$ ,  $L(G)$ , with the use of the Earley parser, if the goal  $\{1 \ n + 1, S' \rightarrow \lambda \bullet\}$  can be attained i.e all the symbols of  $w$  are fully recognized.

### 2.3 RNA Secondary Structure Prediction Using PCFGs

A probabilistic context-free grammar (PCFG) can be used for RNA secondary structure prediction, where the terminals correspond to the nucleotide bases, and the leftmost derivation of an RNA sequence encodes its secondary structure.

The prediction quality of such a model can be evaluated using the metrics *sensitivity* and *positive predictive value* (PPV) [36]. These metrics measure the agreement between the predicted structure and a reference structure:

- **Sensitivity** is defined as the fraction of correctly predicted base pairs among all base pairs in the reference structure. Higher sensitivity implies a higher prediction accuracy.
- **Positive Predictive Value (PPV)** is defined as the fraction of correctly predicted base pairs among all base pairs in the predicted structure. The higher the PPV the more reliable the prediction quality

Formally, let the number of base pairs be classified as follows:

- *TP*: True Positives (correctly predicted base pairs),
- *TN*: True Negatives (correctly predicted non-base pairs),
- *FP*: False Positives (incorrectly predicted base pairs),
- *FN*: False Negatives (missed base pairs in the reference structure).

Using these definitions, the metrics can be expressed mathematically as:

$$\text{Sensitivity} = \frac{TP}{TP + FN},$$

$$\text{PPV} = \frac{TP}{TP + FP}.$$

### 2.3.1 PCFG as Probabilistic Models

**Derivations as representations.** If a grammar  $G$  is known (by convention or because it has been stored explicitly), a **leftmost derivation** ( $\text{lmd}_d(S)$ ) is a derivation  $d = (r_1, \dots, r_t)$  of a word  $w$ .  $w = \text{lmd}_d(S)$  is an encoding for  $w$ : the original word can always be reconstructed from  $d$  by (leftmost) application of the rules starting with  $S$ . This is the basis for our RNA encoding [39]. Note that the grammar is not required to be unambiguous for that, although it seems plausible that unambiguous grammars would yield more effective compression.

If  $G$  is a PCFG, the probability of a derivation  $r_1, \dots, r_t$  in the grammar  $G$ , is the product of the probabilities of all used rules:  $\mathbb{P}[r_1, \dots, r_t] = \prod_{i=1}^t P(r_i)$ . This corresponds to the probability of obtaining this derivation in the random process, where starting with  $S$ , in each time step, we choose a random replacement for the leftmost nonterminal  $A$  in the current sentential form. For that, we sample one of the rules  $A \rightarrow \gamma$  with probabilities according to  $P$  and, conditionally on having left-hand side  $A$ , independent of the past choices.

We define the probability  $\mathbb{P}[w]$  of a word  $w$  as the sum of the probabilities of its derivations.

$$\mathbb{P}[w] = \sum_{r_1, \dots, r_t: \text{lmd}_{r_1, \dots, r_t}(S) = w} \mathbb{P}[r_1, \dots, r_t].$$

The sum is understood to range over all **leftmost derivations**,  $\text{lmd}_{r_1, \dots, r_t}(S) = w$  (of arbitrary length).

We also define the *Viterbi value*  $V(w)$  of  $w$ , the probability of the most likely derivation of  $w$ :

$$V(w) = \max_{r_1, \dots, r_t: \text{lmd}_{r_1, \dots, r_t}(S) = w} \mathbb{P}[r_1, \dots, r_t].$$

If  $G$  is unambiguous, there is only one derivation and we have  $\mathbb{P}[w] = V(w)$ .

## Probabilistic Parsing

Given a PCFG,  $G = (I, T, R, S, W)$  and a word  $w \in T^*$  in the language of  $G$ , a probabilistic parser determines a *Viterbi derivation*, i.e., a most likely derivation for  $w$ :  $\arg \max\{d : \text{lmd}_d(S) = w\}$ .

The theory of such parsers is well established and does not require  $G$  to have any specific normal form [16, 15]. However the resulting general algorithms are rather intricate; for example, chain rules can require to (symbolically) solve infinite summations for correct probabilistic parsing [15]. But such grammars immediately allow an infinite number of leftmost derivations for one word; since our compression methods specify a single derivation, such ambiguity is counterproductive for compression.

A normal form such as *Chomsky normal form (CNF)* (all rules of the type  $A \rightarrow c$  or  $A \rightarrow BC$ ) can simplify parsing dramatically; here even a probabilistic parser remains a simple (bottom-up) dynamic-programming algorithm (probabilistic CYK) [16]. Unfortunately, CNF is inconvenient for expressing the complementarity of paired bases in an RNA. We therefore start by proposing a new normal form for our grammars in Section 5.2.

## 2.4 Related Work

### 2.4.1 On RNA Compression

Liu et al. [32] produced a compressor they termed RNACompress by creating a simple RNA grammar and applying an LL(1) parser (Left to Right, Leftmost derivation with one token look ahead per parsing step) and Huffman codes. Interestingly, this is the first known specialised software designed for compression of RNA primary sequence and structure data [32]. Other compression softwares like GenCompress [4] (derives some of its concepts from Lempel-Ziv's algorithm for sequential data compression [54]), DNACom-

press [5] (uses an improvement of PatternHunter [33] to search for approximate repeats in DNA sequence to improve compression), Biocompress [18] (lossless compression algorithm inspired by the Lempel-ziv compression algorithm) and Cfact [41] (searches the longest exact matching repeat using suffix tree data structure in an entire sequence) are designed for compressing the primary sequences of DNA. So the RNACompress by Liu et al. was compared with general text compression software - Winrar and g.zip and the GenCompress software which can be applied to compress texts in general. RNACompress showed its superiority in terms of compression ratio over these 3 compression methods existing at the time [32].

RNAs can be represented as labelled trees with each node representing a nucleotide and the branches representing the bonds. Friemel [12] explored this treelike property of RNA sequences for compression. His algorithm represents RNAs as strict unary-binary trees, then contracts tree nodes (in a method inspired by Karger's Contraction Algorithm [29]) formed from sequences of multiple dots in the secondary structure or on a sequence of multiple nested brackets in the dot-bracket notation. After the node contraction the algorithm encodes the contracted node tree using Huffman coding [25]. RNAContract, is the term he used for his compression method. In his work, the results of RNAContract outperformed RNACompress in terms of compression ratio.

## 2.4.2 On RNA secondary structure prediction

The functions of RNA are related to its structural characteristics [11]. Obtaining RNA secondary structure information experimentally is expensive with the use of methods like X-ray crystallography. This has driven a growing interest in computational prediction of RNA secondary structure [22]. Predicting RNA secondary structure is highly challenging, as it requires determining correct base pairings and a 3D-helical structure from only a linear sequence of nucleotides. From [21] it is estimated that for 16S rRNA, a molecule 1500 nucleotides in length, there are approximately  $4.3 \times 10^{393}$  possible structure models!

Considerable amount of effort has been put into the prediction of RNA secondary structure. Gutell et al [21] used **covariation analysis** of approximately 7000 16S and 1050 23S rRNA sequences (from [20]) to predict all of the standard secondary structure base pairings and helices in the 16S and 23S rRNA crystal structures. This comparative analysis of RNA structure is based on the principle that different RNA sequences can fold into the same secondary and tertiary structures. It utilizes covariation i.e. similar patterns of variation,

in a set of sequences aligned for maximum sequence identity.

In 1873, Gibbs published his work on free energy [14]. He formulated the Gibbs free energy ( $G$ ), which is foundational for chemical thermodynamics. Gibbs's energy is the thermodynamic potential that is minimized when a system reaches chemical equilibrium at constant pressure and temperature when it is not driven by an applied electrolytic voltage. With the concept of Gibbs's free energy, we conclude that an RNA molecule is most stable when its free energy is minimal. Nussinov et al [38] sought to solve this optimization problem - given an RNA sequence: What is the optimal secondary structure (i.e. one with maximum stability and minimum free energy)? To answer this question they produced an efficient Dynamic Programming algorithm to maximise the number of **complementary base pairs** (as this produces maximum stability). Zuker and Stiegler [55] took the other route to obtain a Dynamic Programming algorithm with the use of energy rules which depend on loops to compute the free energy of the RNA conformation. Further, biochemists have studied the secondary bonds of RNA molecules, which exhibit maximum stability and minimum free energy, and observed that a bond's stability depends on the identity of its nearest neighbor. This insight led to the development of the Individual Nearest Neighbor (INN) and Individual Nearest Neighbor-Hydrogen Bond (INN-HB) models [52, 35], which are employed in RNA secondary structure prediction.

Another path which has been taken to predict RNA secondary structures is the use of **Probabilistic Context Free Grammars(PCFG)** in the prediction of RNA secondary structures. Sakakibara et al applied PCFG to a class of RNAs known as tRNA and showed that they provide a flexible and highly effective method for prediction of secondary structures [44]. Knudsen and Hein produced the KH99-algorithm for secondary structure prediction. The KH99-algorithm uses a probabilistic context-free grammar (PCFG) to produce a prior probability distribution of RNA structures [30]. Dowell and Eddy [8] explore the impact of different PCFG designs on single-sequence RNA secondary structure prediction accuracy. PCFGs can be used to simulate energy models like the Turner energy model and used to carry out analytical analysis of the free energy of RNA secondary structures as shown in [37]. This inspired the design of Schulz's [46] intricate PCFG that mirrors current thermodynamic models applied in modern physics-based RNA secondary structure prediction methods. This grammar produced impressive prediction results.

### 2.4.3 On Generating Refined Grammars

Prior to our work on systematic search of Grammars for better compression (Chapter 6), the refined PCFG grammars used for compression were hand crafted by human experts [32, 8, 46]. To compress RNA structured data, Liu et al. produced a simple RNA grammar (see table 2.1) containing 12 rules and 2 nonterminals, this simple grammar modelled class of RNAs which had only canonical bonds. Dowel and Eddy in [8] produced 2 ambiguous grammars, 4 unambiguous grammars and 3 non-stacking grammars, which were applied to secondary structure prediction. Sakakibara et al [45] produced 4 PCFGs, used an existing alignment of tRNA sequences, with the aid of their **Tree-Grammar EM**, to train the grammars to obtain probabilities which could model tRNA sequences. These grammars named MixedTRNA500, ZeroTrain, MT100, MT10CY10 and RandomTRNA618, produced remarkable Secondary prediction results. Also, in the direction of accurate prediction of secondary structure of RNA grammar, Schulz [46] produced an intricate, unambiguous, epsilon free grammar, consisting of 108 nonterminals and 244 rules!

Considering the great effort involved in producing grammars for joint RNA compression and structure prediction made us think: Can we automate the process of obtaining refined grammars? We answer this question in Chapter 6.

## Chapter 3

# Joint Compression of RNA Primary and Secondary structure

### 3.1 Background

The first known specialised software for RNA primary and secondary structure data compression is the 2008, Liu et al. work [32]. They explored the joint compression of RNA primary and secondary structures and introduced a tool called RNACompress. Their method involved constructing a simple RNA grammar and implementing an LL(1) parser, which processes input from left to right using a leftmost derivation with a one-token lookahead per parsing step, combined with Huffman coding. The results showed that RNACompress achieved a higher compression ratio compared to three existing methods at the time [32].

Given the tree-like structure of RNA molecules, Friemel explored various tree representations for RNA compression, including unranked trees, unary-binary trees, and strict unary-binary trees. Ultimately, he focused on strict unary-binary trees, as they require fewer bits to encode each node. His algorithm simplifies tree structures by merging consecutive dots in the RNA secondary structure or collapsing multiple nested brackets in dot-bracket notation. After this transformation, the resulting compressed tree is further encoded using Huffman coding. He named this approach RNAContract, which in his study, achieved a higher compression ratio than RNACompress.

## 3.2 Using Refined Grammars and Arithmetic Coding

In this part of the research we applied and implemented arithmetic coding to the combined RNA primary and secondary structure compression. We integrated the Earley Parser library into our compression implementation, which has proven highly effective in handling various PCFGs. By experimenting with both ambiguous and unambiguous grammars [8] and combining them with arithmetic encoding (using semi-adaptive and adaptive models), we achieved a significant improvement in compression performance compared to RNACompress and RNAContract.

We showed in Example 2.1.1 the steps in encoding an RNA using arithmetic coding. In the example a *static rule-probability model*, was used. The static rule-probability model is usually obtained from a training dataset with known structures by counting how often each rule is used in the dataset derivations. With arithmetic coding, we can easily replace this by an *adaptive rule-probability model*, where rule probabilities are computed as relative frequencies in the prefix encoded so far (starting with some initial value for counters, typically 1). This entirely avoids the need for a second pass or a training dataset, as well as storing the rule probabilities. For long input, the adaptive model converges to the sequence-specific relative rule frequencies; we hence also include the *semi-adaptive model* where rule counts are determined for the given sequence in a first pass. Unless one also stores the rule counts, this model does not allow decoding, but can indicate the limiting behavior of the adaptive model.

## 3.3 Compression Experiments

### 3.3.1 Huffman coding vs. Arithmetic coding

We here compare the influence of the coding step on compression ratio in isolation. For that, we modify Liu et al.’s RNACompress [32] to use arithmetic coding instead of a Huffman code, leaving everything else unchanged, and compare the results.

We were not able to obtain the original implementation of RNACompress and the datasets from Liu et al. [32]. We hence re-implemented RNACompress, and used the Friemel-modified dataset of 17 000 RNA samples originally taken from [3] instead of the dataset from [32]. Some of the RNAs in Friemel’s dataset have non-canonical bonds (these are less stable secondary bonds). Since Liu et al. do not allow non-canonical bonds in their tool, we also removed these from Friemel’s dataset, i.e., we replaced the open ( and

close ) parenthesis for non-canonical bonds with unpaired bases • in the positions were non-canonical bonds appeared. Afterwards only the stable bonds (Watson-Crick and G–U wobble bonds) were left in all samples in the dataset, which we call Friemel-modified.

Unsurprisingly, the arithmetic coding produced better compression results than Huffman coding, but the difference between the means is only 2.7%. Figure 3.1 shows the distribution of compressed size over the RNAs; while arithmetic coding has moderate impact on the mean compressed size, it helps a lot to bring down the right tail. The scatterplot in Figure 3.2 further shows that indeed, arithmetic coding (with this fixed static model) is doing better on almost all RNAs, and the effect is bigger for those RNAs that are compressed worse.

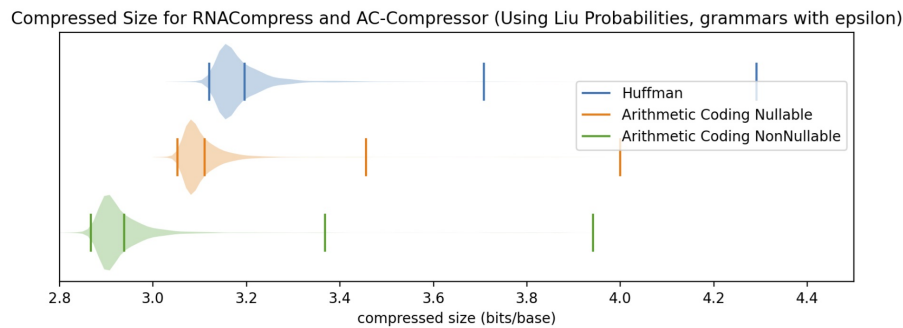


Figure 3.1: Compressed size in bits per base for RNACompress (original with Huffman coding) and RNACompress with arithmetic coding, and the RNACompress variant with the  $\epsilon$ -rule-free grammar. The vertical bars show from left to right, the 1% quantile, mean, 99% quantile, and maximum. The means obtained are 3.195 and 3.110 bits per base for the Huffman and non-nullable (AC-Compressor) compression methods respectively.

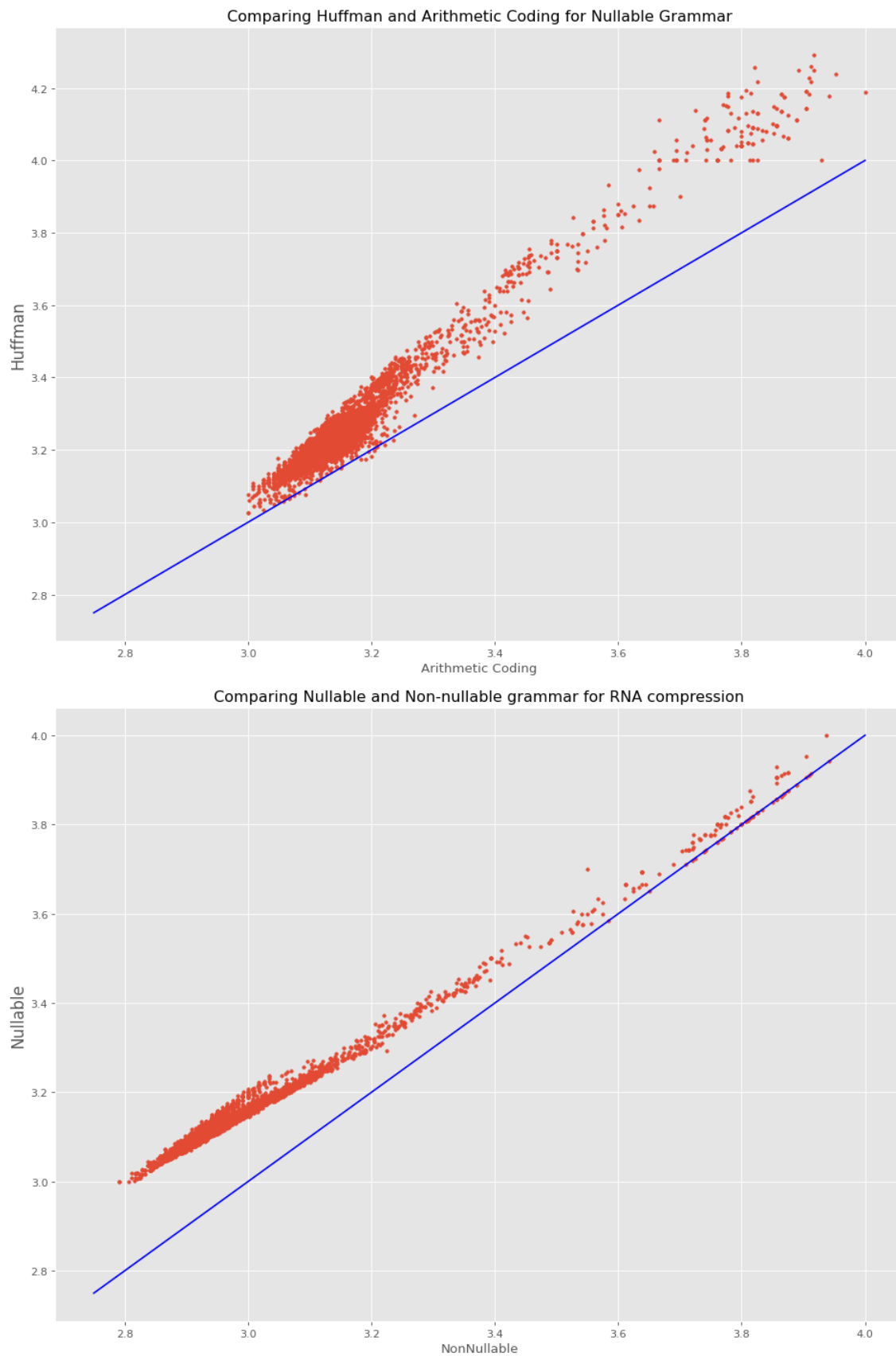


Figure 3.2: The same data as in Figure 3.1, but as scatter plots with one point per RNA.

### 3.3.2 Nullable Grammar vs. Non-Nullable Grammar

Liu et al. [32] originally use the following grammar

$$G_L$$

$$L \rightarrow aS\hat{a} \mid a$$

$$S \rightarrow LS \mid \epsilon$$

For general parsers,  $\epsilon$ -rules are often inconvenient; we therefore modified this grammar to

$$G_{L'}$$

$$T \rightarrow a$$

$$T \rightarrow aS\hat{a}$$

$$S \rightarrow T \mid TS \text{ (where } a\dots\hat{a} \text{ are canonical base pairs)}$$

This transformation makes the probabilistic model slightly richer and so will help compression, but it does not change the nature of the grammar; the structure of leftmost derivations of strings remain (almost) the same. (We here ignore the fact that the empty string is no longer in the language of grammar  $G_{L'}$ , while it was derivable in  $G_L$ . For RNA compression, this is not relevant.) We manually implemented a parser for the original  $G_L$  grammar and compared the compression outcome. As Figure 3.1 shows, this very moderate enrichment of the probabilistic model has a larger impact than moving from Huffman to arithmetic coding. The scatter plot in Figure 3.2 (bottom) shows that again, we never do worse in  $G_{L'}$  compared to  $G_L$ , but that this time, the biggest savings are happening for the (much larger number of) RNAs that are compressed *well*.

### 3.3.3 Joint compression of Primary and Secondary Structure using Various Refined Grammars

We went further to apply the arithmetic coding compressor with 8 different grammars from [8] and [46] to Friemel's dataset (see Appendix A). Figure 3.3 shows the compression quality of different grammars, normalized to the (average) number of bits per base in the RNA. Interestingly, the method from the literature, Liu et al.'s RNACompress [32], performs much worse than all the structure-prediction grammars (for all rule-probability models), indicating that these grammars indeed incorporate effective domain knowledge about RNA structures. Also note that a simplistic encoding of the RNA sequence alone would use 2 bits/base; the most sophisticated grammars come very close to that for the joint encoding of sequence *and* structure: 2.11 bits/base on average for the grammar of Nebel and

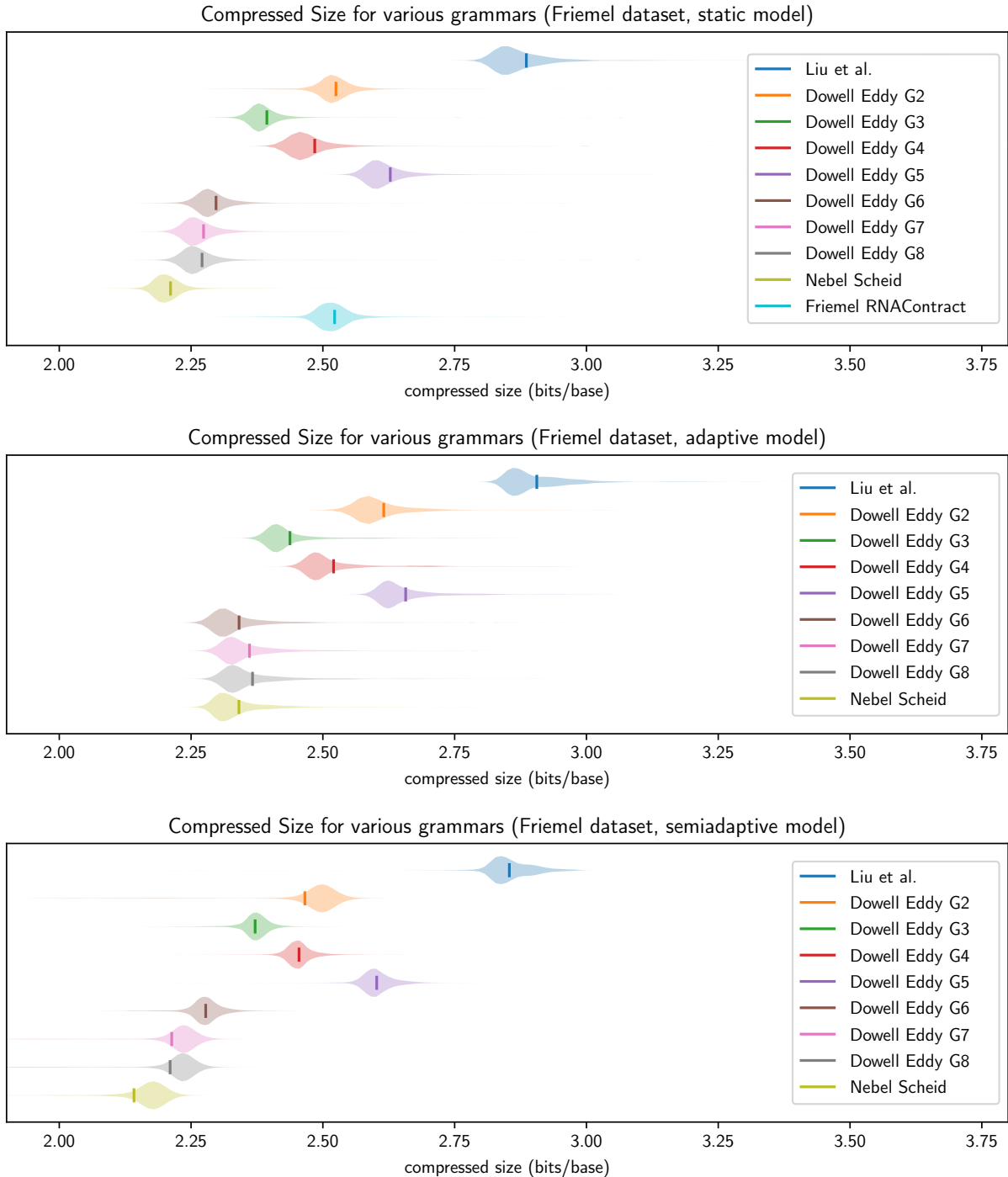


Figure 3.3: The means (are shown with vertical bars) and distributions (are the shaded violin plot) of the normalized compressed size using various grammars on Friemel’s RNA dataset. All compressed sizes are shown as bits per base. **Top:** results using static rule probabilities, determined from the entire dataset. **Middle:** results using adaptive rule-probabilities model (Laplace model). **Bottom:** semi-adaptive rule probabilities (ignoring space for storing rule probabilities).

Scheid [34]. The large grammars  $G_2$ ,  $G_7$ , and  $G_8$  [8] (those with “stacking parameters”) and the huge grammar by Nebel and Scheid [34](see Appendix A) perform overall best. But some much smaller grammars like  $G_6$  come very close, despite having a factor 10 fewer parameters. This shows that it is the structure of the grammar, not merely the number of parameters of the model, that improve compression of RNA secondary structures.

### 3.4 Summary

The results obtained from the implementation of arithmetic coding along with refined grammars for the compression of RNA structured data shows clearly that the AC-Compressor produces better compression than RNACompress [32].

## Chapter 4

# RNA Secondary Structure Prediction and Compression

### 4.1 Background

Determining RNA secondary structure requires **expensive**, techniques like X-ray crystallography [48]. This has driven the interest in RNA secondary structure prediction. Extensive research has been conducted on RNA secondary structure prediction. As discussed in Subsection 2.4.2, RNA secondary structure prediction relies on various techniques, including thermodynamic models, dynamic programming, stochastic and probabilistic approaches, comparative sequence analysis, machine learning methods, and hybrid strategies. Hybrid methods are particularly interesting as they combine multiple modeling approaches. For instance, Zuker and Stiegler [55] developed a dynamic programming algorithm, similar to Nussinov's [38], that not only predicts the structure with the maximum number of Watson-Crick base pairs but also identifies the most thermodynamically stable structure by minimizing free energy. This was achieved by integrating additional parameters from enzymatic studies to optimize the predicted structure. Xia et al., Andronescu et al. [17], and Turner et al. further refined this approach by incorporating additional nearest-neighbor parameters to improve the accuracy of RNA secondary structure prediction.

Probabilistic Context-Free Grammars (PCFGs) have also been applied to RNA secondary structure prediction, where terminals represent bases, and the most probable left-most derivation of an RNA sequence corresponds to its secondary structure. Dowell and Eddy [8] used simple PCFGs to predict RNA secondary structure and compared their

prediction quality. Similarly, Schulz [46] used probabilistic models and PCFGs to predict RNA secondary structures. In this chapter we consider the Nussinov algorithm for secondary structure prediction and the SRF-Viterbi Predictor.

### 4.1.1 Nussinov Algorithm

The Nussinov Algorithm pioneered the use of dynamic programming for efficiently identifying stable base pairs and predicting RNA secondary structure. It searches for the maximum number of canonical base pairs (i.e., Watson-Crick pairs: A–U, G–C, and sometimes G–U) that can form in a given RNA sequence by using a dynamic programming approach. As defined in [9], given a sequence  $x$  of length  $L$  with symbols  $x_1, \dots, x_L$ . Let  $\delta(i, j) = 1$ , if  $x_i$  and  $x_j$  are a complementary/canonical base pair, otherwise  $\delta(i, j) = 0$ . To obtain the maximum number of base pairs that can be formed for subsequence  $x_i, \dots, x_j$ , we recursively calculate the score  $\gamma(i, j)$  as follows:

**Fill stage:** Initialisation:

$$\begin{aligned} \gamma(i, i-1) &= 0 && \text{for } i = 2 \text{ to } L; \\ \gamma(i, i) &= 0 && \text{for } i = 1 \text{ to } L. \end{aligned}$$

Recursion:

for subsequence of length 2 to L

$$\gamma(i, j) = \max \begin{cases} \gamma(i+1, j), \\ \gamma(i, j-1), \\ \gamma(i+1, j-1) + \delta(i, j), \\ \max_{i < k < j} [\gamma(i, k) + \gamma(k+1, j)] \end{cases}$$

The first part of Nussinov’s algorithm: Initialisation and recursion, is the dynamic programming matrix fill stage. The application is shown in Figure 4.2 using the sequence AAAGGUCC (made up for the illustration). On completion, the value of  $\gamma(1, L)$  gives the maximum number of base pairs for the RNA sequence. It is important to state that there could be multiple folded structures for the given sequence, which have the **maximum number** of base pairs. The traceback algorithm is used to obtain one of these structures. We trace back from  $\gamma(1, L)$  on the dynamic programming matrix.

**Traceback:****Initialisation:** Push  $(1, L)$  onto a stack**Recursion:**

```

while (stack.size!=0){
    pop(i,j);
    if (i>=j)
        continue;
    else if ( $\gamma(i+1,j) == \gamma(i,j)$ ) push ( $i+1,j$ );
        else if ( $\gamma(i,j-1) == \gamma(i,j)$ ) push ( $i,j-1$ );
            else if ( $\gamma(i+1,j-1) + \delta(i,j) == \gamma(i,j)$ )
                record  $i,j$  base pair;
                push ( $i+1,j-1$ );
    else
    for ( $k = i+1$  to  $j-1$ )
        if ( $\gamma(i,k) + \gamma(k+1,j) == \gamma(i,j)$ )
            push ( $k+1,j$ );
            push ( $i,k$ );
        exit loop
}

```

In Listing 2 the algorithm is rendered in C style. The matrix  $DP[N][N]$  is the dynamic programming table which stores the base pairing scores for the RNA sequence. After initialising the diagonals (lines 4-8), each cell  $DP[i][j]$  is filled by selecting the **highest** score from the cell to its left ( $DP[i][j-1]$  ( $j$  is unpaired)), below ( $DP[i+1][j]$  ( $i$  is unpaired)) or the value from the diagonal cell ( $DP[i+1][j-1]$ ) plus a base-pairing score, which accounts for the possibility that bases at positions  $i$  and  $j$  form a valid pair (lines 14-18). This obtains maximum base pair at a single helical substructure. Lines 20 - 22 obtains maximum base pairing, by searching for the bifurcation (2 helices joined by unpaired bases) with maximum base pairs. Lines 37-38 traces back on  $DP$  to obtain a/the structure with maximum base pair.

Listing 2: The Nussinov Algorithm

```

1  // Nussinov Algorithm for RNA Secondary Structure Prediction

3  function Nussinov(char RNA[], int N):
4      int DP[N][N] // Initialize DP table
5      for i= 1 to N
6          DP[i][i] = 0 // Set diagonal to 0
7          for i= 2 to N
8              DP[i][i-1]= 0//sets subdiagonal to zero

10     // Fill the DP table
11     for length = 1 to N-1:
12         for i = 0 to N - length - 1:
13             j = i + length
14             DP[i][j] = max(
15                 DP[i+1][j],          // Case 1: i is unpaired
16                 DP[i][j-1],          // Case 2: j is unpaired
17                 DP[i+1][j-1] + match(RNA[i], RNA[j]) // Case 3: (i, j) form a ↵
18                 pair
19             )

20             // Case 4: Check all possible bifurcations
21             for k = i to j-1:
22                 DP[i][j] = max(DP[i][j], DP[i][k] + DP[k+1][j])

24     return DP // Return filled DP table

26 // Function to check if two bases can pair
27 function match(char a, char b):
28     return (a == 'A' && b == 'U') (a == 'U' && b == 'A')
29         (a == 'C' && b == 'G') (a == 'G' && b == 'C')

31 // Function to initialise DP table
32 function initialize(int DP[][], int N):
33     for i = 0 to N-1:
34         for j = 0 to N-1:
35             DP[i][j] = 0 // Base case: single nucleotides have no pairs

```

```
37 // Function to reconstruct structure from DP table
38 function traceback(int DP[][], char RNA[], int i, int j):
39     if i >= j:
40         return
41     if DP[i][j] == DP[i+1][j]:
42         traceback(DP, RNA, i+1, j) // i is unpaired
43     else if DP[i][j] == DP[i][j-1]:
44         traceback(DP, RNA, i, j-1) // j is unpaired
45     else if DP[i][j] == DP[i+1][j-1] + match(RNA[i], RNA[j]):
46         print("(" + i + ", " + j + ")") // Pair (i, j)
47         traceback(DP, RNA, i+1, j-1)
48     else:
49         for k = i to j-1:
50             if DP[i][j] == DP[i][k] + DP[k+1][j]:
51                 traceback(DP, RNA, i, k)
52                 traceback(DP, RNA, k+1, j)
53                 break
```

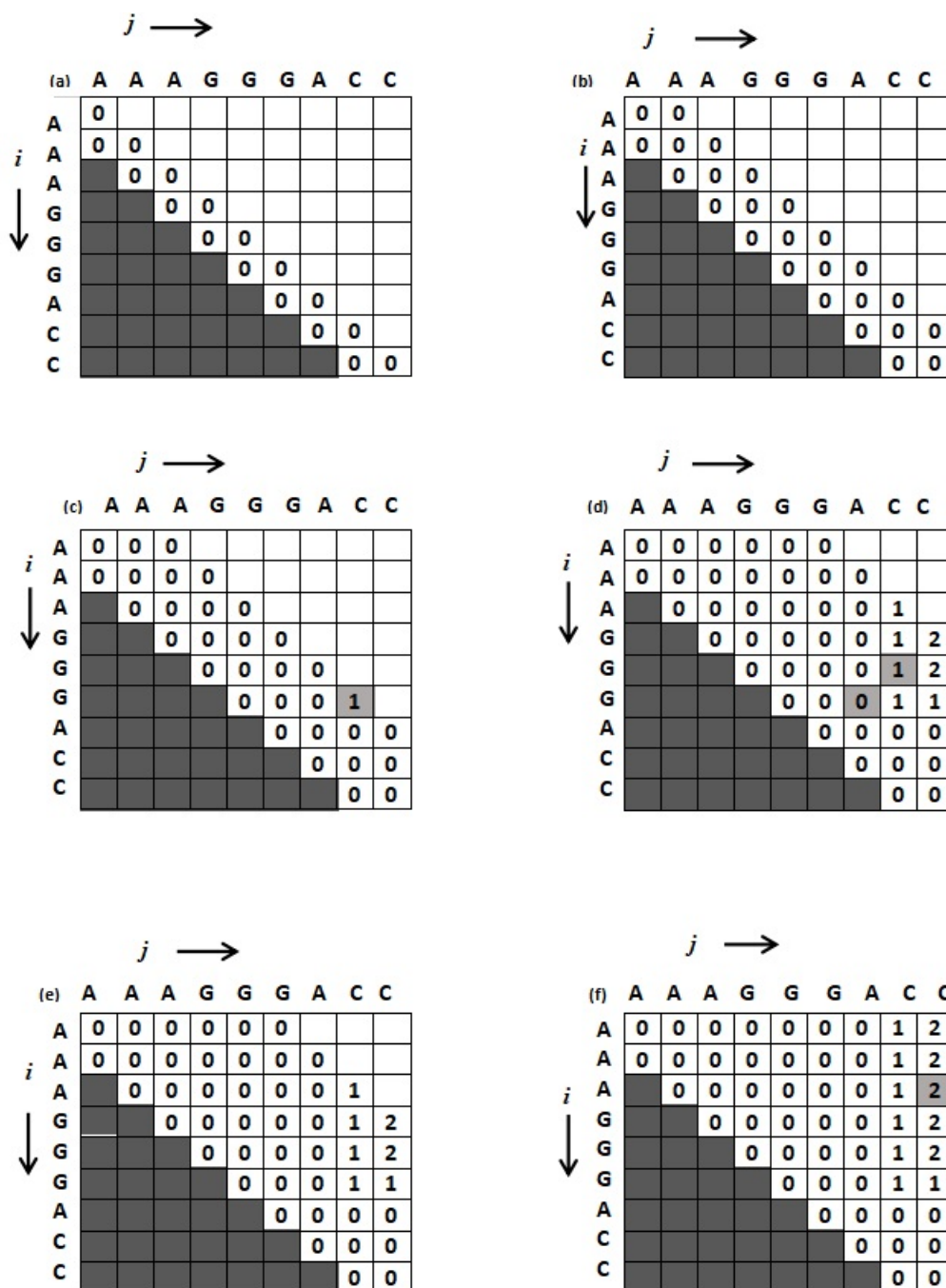


Figure 4.1: This figure shows some steps in applying the Nussinov algorithm to the RNA sequence AAAGGGACC (a) the initialisation stage: the dark grey shaded region is excluded. All cells on diagonal path beneath the major diagonal and the major diagonal itself are initialised to zero (b) All cells above the leading diagonal here evaluates to zero as this checks for bonds between adjacent bases and there are no two adjacent bases in this example that form canonical bonds. (c) The light grey shaded cell has value 1, because the bases at index  $i=6$  and  $j=8$  form a canonical base pair GC,  $\gamma(i+1, j-1) + \delta(i, j) = 0 + 1 = 1$  and this is the maximum value when all function parts in 4.1.1 are applied. (d) computes the scores in the next set of grey cells in a similar manner to (c), while in (f) after the part function is applied the maximum value obtained using  $\gamma(i, j) = \gamma(i+1, j)$ , so  $\gamma(3, 9) = \gamma(4, 9) = 2$ .

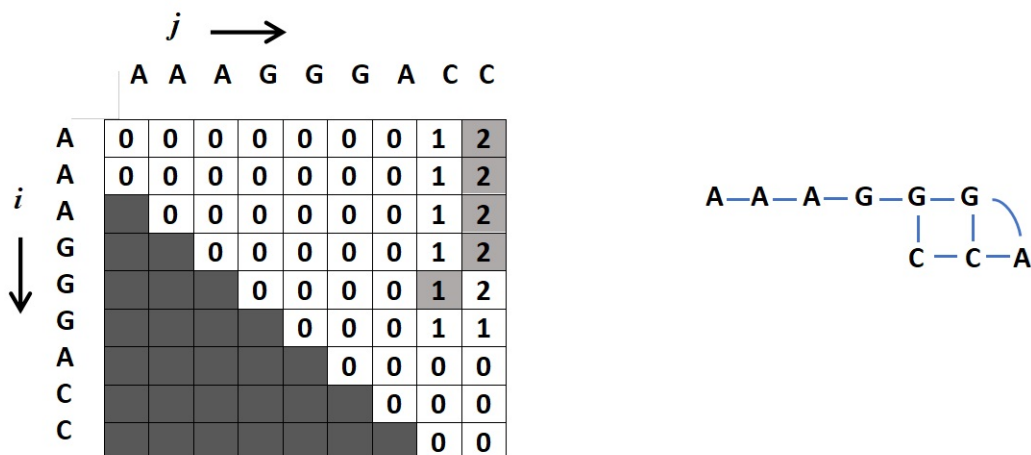


Figure 4.2: Region shaded light grey in matrix shows the traceback cells, on the right the folding over of the RNA is shown

## 4.2 Secondary Structure Prediction Algorithm Inspired by the Viterbi Algorithm

We introduce a dynamic programming algorithm that is inspired by the Viterbi Algorithm applied to Hidden Markov Models [49] to predict RNA secondary structure. As defined in Section 2.3.1, the *Viterbi value*  $V(w)$  of  $w \in L(G)$  i.e. the language of  $G$  (in this context  $w$  is an RNA primary sequence) and  $G$  a PCFG is the probability of the most likely derivation of  $w$ :

$$V(w) = \max_{r_1, \dots, r_t: \text{lmd}_{r_1, \dots, r_t}(S)=w} \mathbb{P}[r_1, \dots, r_t].$$

where  $\text{lmd}_{r_1, \dots, r_t}(S)$  denotes the leftmost derivation of  $r_1, \dots, r_t$  from the start non-terminal  $S$  and the probability of a derivation  $r_1, \dots, r_t$  in the grammar  $G$  denoted by  $\mathbb{P}[r_1, \dots, r_t]$ . The derivation  $r_1, \dots, r_t$  in the grammar  $G$  which produces  $V(w)$  gives the predicted structure.

The pseudocode for the algorithm for prediction is shown in Listing 3. It uses SRF parser (see Section 5.2) and the viterbi algorithm to obtain the most likely derivation for an RNA primary sequence  $a_1 a_2 \dots a_n$ , with a given PCFG grammar  $G$  with  $r$  nonterminals  $R_1, R_2, \dots, R_r$ . The algorithm uses the 4 types of rules defined in SRF (Section 5.2 covers this in detail) in the parsing algorithm. The SRF rules are:

- (i)  $A_i \rightarrow A_j A_l$     (ii)  $A_i \rightarrow \bullet$     (iii)  $A_i \rightarrow (A_j)$     (iv)  $A_i \rightarrow A_j$  and  $j < i$

Rules (i) and (ii) are same as the rules in Chomsky Normal Form (see Section 2.2.4). Rule (iii) takes care of the complementary base pairs in RNA secondary structure and (iv) places the nonterminals in a hierarchy in order to simplify parsing. Lines 9 - 20, in Listing 3, the unit productions  $R_v \rightarrow a_s$  are entered into the probabilities  $P[][][]$  and boolean  $B[][][]$  tables. The viterbi table  $V[][][]$ , stores the maximum probabilities, given a rule with nonterminal  $R_t$  on the left, partition  $p$  of length  $l$  and start index  $s$  in the entry  $V[l][s][t]$ . The `back_rule[][][]` (line 6) and `back[][][]` (line 5) arrays are used to store backtracking values only for rules whose probabilities are stored in  $V[][][]$ .  $B[][][]$  is set to true if the corresponding value in  $P[][][] > 0$ . Lines 27 - 41 iterates through rules of type  $R_d \rightarrow (R_c)$ . These rule types check for matching base pairs of rules and are key to obtaining accurate secondary structure prediction (without this rule type, all bases would be unpaired in the predicted structure). Rule type  $R_h \rightarrow R_g$  simply, substitute a nonterminal with a higher hierarchy with a lower one (see Section 5.2), the iteration for this rule type is found in lines 44 to 57.  $R_a \rightarrow R_b R_c$  rule type, partitions the span of the length or sublength of the RNA primary sequence in 2. Probabilities of each part is determined and their product gives the probability for rule  $R_a \rightarrow R_b R_c$  for the given span (see line 63-80). For any span and start point of the RNA sequence, the viterbi is updated and only with the maximum probability for the given span. The rule which provides this value is stored in `back_rule` and `back`. At the end of the iteration, if  $B[n][1][1]$  is true the `backtraceDerivation` returns the derivation for the RNA sequence, with the predicted secondary structure.

Listing 3: SRF-Viterbi Prediction Algorithm

```

1 // Assume the RNA primary sequence consists of n characters: a_1 a_2 ... a_n ←
   in this order
2 // Assume the grammar contains r nonterminal symbols R_1, R_2, ..., R_r, with ←
   R_1 as the start symbol
3 // Initialize arrays
4 float P[n][n][r] = {0};           // Array for probabilities
5 bool B[n][n][r] = {false};       // Array for boolean feasibility
6 float V[n][n][r] = {0};           // Array for maximum probabilities
7 list back[n][n][r];               // Array for backtracking information

```

```

8 rule back_rule[n][n][r];           // Array for backtracking rules
9 int back_split[n][n][r];         // Stores the split position for the subproblems

11 // Type 1 Rules: Unit productions R_v -> a_s
12 for (int s = 1; s <= n; s++) {
13     for (int v = 1; v <= r; v++) {
14         if (R_v -> a_s) { // Check if R_v -> a_s is valid//type
15             P[1][s][v] = p(R_v -> a_s); // Set probability
16             B[1][s][v] = true;          // Mark as derivable
17             V[1][s][v] = P[1][s][v];   // Update Viterbi value
18             back[1][s][v] = {{1, s, v}}; // Store backpointer
19             back_rule[1][s][v] = R_v -> a_s;
20         }
21     }
22 }

24 // Iterate over lengths of spans
25 for (int l = 2; l <= n; l++) { // Length of span
26     for (int s = 1; s <= n - l + 1; s++) { // Start of span
27         int e = s + l - 1; // End of span

29         // Type 3 Rules: R_d -> (R_c)
30         for each( Rule R_d -> R_c){
31             if (is_base_pair(a_s, a_e) && B[l - 2][s + 1][c]) {
32                 float prob = p(R_d -> (R_c)) * P[l - 2][s + 1][c];
33                 B[l][s][d] = true;
34                 P[l][s][d] = prob;

36                 if (V[s][e][d] < prob) { // Update max probability
37                     V[s][e][d] = prob;
38                     back[l][s][d] = {{l - 2, s + 1, c}};
39                     back_rule[l][s][d] = R_d -> (R_c); // Update back_rule
40                     back_split[l][s][d] = 1; // Type 3 rules have only ←
41                                     one partition
42                 }
43             }

```

```

45     // Type 4 Rules: R_h -> R_g
46     for each (Rule R_h -> R_g){
47         if (B[l][s][g]) {
48             float prob = p(R_h -> R_g) * P[l][s][g];
49             B[l][s][h] = true;
50             P[l][s][h] = prob;

52             if (V[s][e][h] < prob) { // Update max probability
53                 V[s][e][h] = prob;
54                 back[l][s][h] = {l, s, g};
55                 back_rule[l][s][h] = R_h -> R_g; // Update back_rule
56                 back_split[l][s][h] = 1;
57             }
58         }
59     }

62     // Type 2 Rules: R_a -> R_b R_c
63     for (int p = 1; p < l; p++) { // Partition the span
64         for each (Rule R_a -> R_b R_c) {
65             if (B[p][s][b] && B[l - p][s + p][c]) {
66                 float prob = p(R_a -> R_b R_c) * P[p][s][b] * P[l - p][s +
67                 p][c];
68                 B[l][s][a] = true;
69                 P[l][s][a] = prob;

70                 if (V[s][e][a] < prob) { // Update max probability
71                     V[s][e][a] = prob;
72                     back[l][s][a] = {p, b, c};
73                     back_rule[l][s][a] = R_a -> R_b R_c; // Update back_rule
74                     back_split[l][s][a] = p; // Position for partition split
75                 }
76             }
77         }
78     }
79 }
80 }

82 void backtraceDerivation(int l, int s, NonTerminal_Index t, List<Rule> result) ↵

```

```

        throws UnparsableException {
83     int p = back_split[l][s][t];
84     Rule rule = back_rule[l][s][t];
85     if (rule == null) throw new UnparsableException();

87     result.add(rule);

89     if (type1Rules.contains(rule)) { // R_a -> a
90         // Type 1 rules are terminal, so we return
91         return;
92     } else if (type2Rules.contains(rule)) { // R_a -> R_b R_c
93         backtrackDerivation(p, s, b, result);
94         backtrackDerivation(l - p, s + p, c, result);
95     } else if (type3Rules.contains(rule)) { // R_d -> (R_c)
96         backtrackDerivation(l - 2, s + 1, c, result);
97     } else if (type4Rules.contains(rule)) { // R_h -> R_g
98         backtrackDerivation(l, s, g, result);
99     }
100 }

102 // Final check: Is the sequence derivable?
103 if (B[n][1][1]) {
104     printf("A is a valid RNA primary sequence");
105     List<Rule> result = new ArrayList<>();
106     printf("Most likely derivation for structure = %s", backtrackDerivation(n, ↵
107         1, 1, result));
108 } else {
109     printf("A is not a valid RNA sequence");
110 }

```

To determine the complexity of the SRF-Viterbi Predictor algorithm we consider the main loop from lines 10 - 78, particularly the worst case in lines 60-73. The loop iterates as follows:

- Over all possible span lengths  $l$ , from 2 to  $n$ :  $O(n)$ .
- Over all possible start positions  $s$ :  $O(n)$ .
- Over all possible split points  $p$ :  $O(n)$ .

- For each rule application (bounded by the number of rules  $r$ ).

Thus, the worst-case complexity is:  $O(n^3r)$

### 4.3 Compression Ratio Vs. Prediction Quality

In this section, we take a close look at the correlation between compression and secondary-structure prediction as related to different grammars. For that, we reproduced the classic study of Dowell and Eddy [8] comparing several hand-crafted PCFG for their ability to correctly infer RNA secondary structures given only the RNA sequence as input. The SRF parsing algorithm was in its conceptual and design phase when we carried out this investigation, so we employed the probabilistic Earley parser from [50]. We use the original datasets from [8] (available at <http://eddylab.org/software/conus/>): The “benchmark” dataset was used in [8] to compare the prediction quality of PCFGs, whose rule probabilities have been trained on their “mixed80” dataset; see [8] for further details. Both datasets contain many non-canonical bonds and 8 RNAs contain empty hairpin loops; we again eliminated the latter. Mixed80 contains numerous ambiguous bases; these were randomly replaced with a compatible base.

At least for the grammars from [8], this shows that one can use compressed size as a more rigidly defined and robust proxy for secondary-structure prediction quality.

Figure 4.4 takes a closer look at the correlation on a per-RNA level. Even there, a correlation remains visible; in particular very accurately predicted structures are also well compressed. The right panel in Figure 4.4 shows that compressed size for different grammars is very strongly correlated; pictures for other grammar pairs are similar (excluding the poor performing  $G_1$ ,  $G_4$ , and  $G_5$ ). Note that despite the strong correlation at RNA level, there is a significant difference in the (mean) compression ratio between different grammars. This might indicate that there are intrinsically more and less “surprising” RNA secondary structures (knowing only the RNA sequence).

### 4.4 Summary

In this chapter, algorithms for secondary structure prediction were considered. A clear correlation was shown between RNA secondary structure prediction and RNA compression using 8 different PCFG grammars. This is promising evidence for the utility of compression

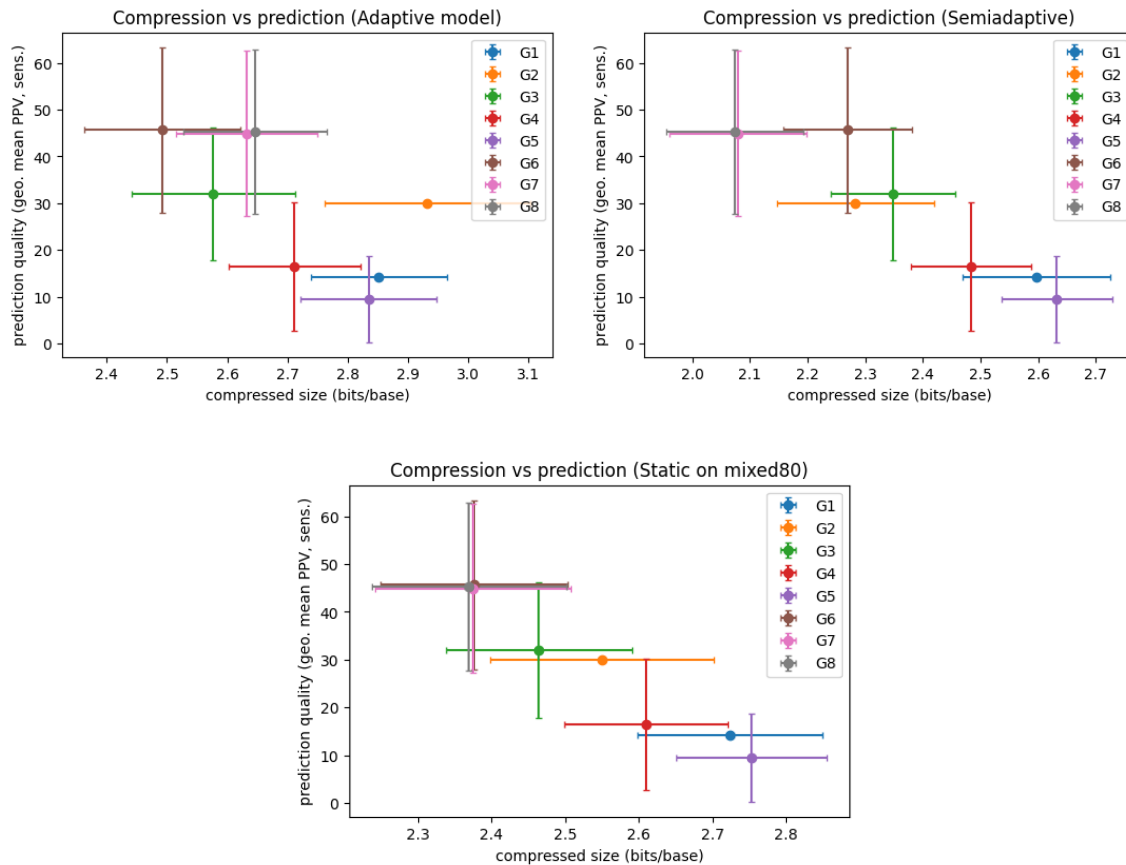


Figure 4.3: Scatter plot of compression vs. prediction quality for the grammars from [8] using the adaptive, semi-adaptive and static probability models. Each grammar is presented as one point with error bars. The  $x$ -axis shows the compressed size (in bits per base) for joint compression of RNA sequence and secondary structure, averaged over the benchmark dataset [8]. Horizontal error bars show one standard deviation of compressed size over the benchmark dataset. The  $y$ -axis shows the geometric mean of sensitivity and PPV (for each predicted RNA secondary structure, averaged over the benchmark dataset); error bars show one standard deviation. For the ambiguous grammars  $G_1$  and  $G_2$ , no vertical error bars are available (we did not reproduce predictions for these; the average is taken from [8]). Both compression and prediction use the same training dataset (mixed80 from [8]) to determine the parameters of the grammars; compression here uses the static model for rule probabilities..

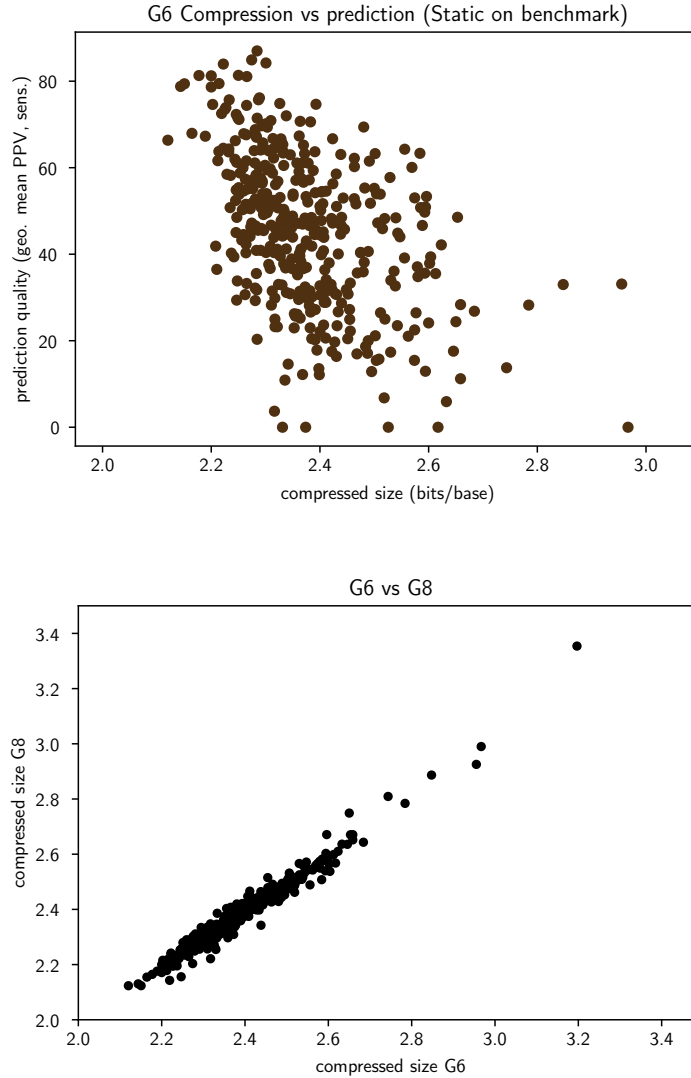


Figure 4.4: Scatter plots with one point per RNA sequence in the benchmark dataset. **top:** compressed size against prediction quality using  $G_6$ . **bottom:** compressed size using  $G_6$  against compressed size using  $G_8$ . All compression methods use the static rule probabilities trained on mixed80.

ability as a cheap and robust proxy for prediction quality for RNA secondary-structure prediction.

# Chapter 5

## Search Algorithms for Grammars

### 5.1 Introduction

In chapter 4, we showed that joint compression of RNA sequence and structure data can serve as a robust proxy for the prediction quality of different probabilistic context-free grammar (PCFG) models of RNA secondary structures, a state-of-the-art formalism for *ab initio* structure prediction.

In [39], we utilize PCFGs that were either designed or collected by human domain experts [8, 32, 34]. In this chapter, we transition from these isolated examples **to a framework for the systematic and automated search for *optimal* grammars**. By comparing PCFGs using their achieved compressed size instead of structure prediction performance, we eradicate intricacies and ongoing debates of how to measure the distance between predicted and true secondary structures [36]; our work thus paves the way for a well-defined open contest on finding PCFGs that best capture the essence of stable (minimum-free-energy) RNA structures.

On the technical side, we provide reference implementations of all key components needed for testing and evaluating PCFGs for RNA compression and prediction and we explore best practices for improving the efficiency of the search for good grammars.

Moreover, we report results from an initial exploration of the space of grammars. We find that the vast majority of grammars give rather poor compression, but a very small number achieve substantial compression. Among those, we could identify several new grammars that surpass the performance of similar-sized human expert grammars from the literature, indicating that further improvements are likely to be possible and that the intuitive grasp

of RNA structures even among domain experts has limitations.

## 5.2 Stochastic RNA Normal Form for Grammars

We consider PCFGs in a specific normal form, the *Stochastic RNA Form (SRF)*. It takes inspiration from existing expert PCFG designs used for RNA structure prediction [8, 32, 34]. Our normal form assumes a total order on the nonterminals. To simplify notation in this section, assume without loss of generality that  $N = \{A_1, \dots, A_k\}$ ; we define  $A_i < A_j$  if and only if  $i < j$ .

A PCFG  $G = (N, T, R, A_k, P)$  is in *Stochastic RNA Form* if each of its rules has one of the following forms:

$$\boxed{\text{(i) } A_i \rightarrow A_j A_l \quad \text{(ii) } A_i \rightarrow \bullet \quad \text{(iii) } A_i \rightarrow (A_j) \quad \text{(iv) } A_i \rightarrow A_j \text{ and } j < i}$$

The ordering constraint on type (iv) rules ensures that there is a finite number of derivations for every word and that we can retain a total order on subproblems in parsing (see below). Apart from making parsing more efficient, the Stochastic RNA Form makes it trivial to ensure that grammars produce valid RNA structures. It therefore massively reduces the search space in our exhaustive search by excluding many invalid grammars.

The Stochastic RNA Form is chosen to be as expressive as possible, fixing only features that all stable RNA structures share. One tacit assumption we impose on RNA structures is that they have no empty hairpin loops, i.e., no subword “( )”. Such a bond is indeed impossible due to physical limitations; it does get reported in databases, but rarely so (and likely erroneously).

Given a grammar in SRF, we can adapt the probabilistic CYK parser [16] to our grammars as follows: For that, we denote by  $V_A(w[i..j])$ , for  $A \in N$  and  $0 \leq i < j \leq n = |w|$  the probability of the most likely derivation of  $w[i..j]$  when starting with  $A$ . We then have  $V(w) = V_S(w[0..n])$  for  $S$  the start symbol of  $G$  and obtain the recursive equations following the allowed rule types:

$$V_A(w[i..i+1]) = \begin{cases} P(A \rightarrow w[i]) & \text{if } A \rightarrow w[i] \in R \text{ (ii)} \\ 0 & \text{otherwise} \end{cases}$$

$$V_A(w[i..j]) = \max \begin{cases} \max_{\substack{k \in [i+1..j] \\ A \rightarrow BC \in R}} P(A \rightarrow BC) V_B(w[i..k]) V_C(w[k..j]) & \text{(i)} \\ \max_{A \rightarrow w[i] B w[j-1] \in R} P(A \rightarrow w[i] B w[j-1]) V_B(w[i+1..j-1]) & \text{(iii)} \\ \max_{A \rightarrow B \in R} P(A \rightarrow B) V_B(w[i..j]) & \text{(iv)} \end{cases}$$

The ordering constraint on type (iv) rules implies that the subproblems  $V_A(w[i..j])$  can be totally ordered by  $(\ell, A)$  for  $\ell = j - i$  the length of the produced subword and  $A$  the nonterminal, allowing for an efficient bottom-up dynamic-programming parser.

### 5.3 Exhaustive Search for Grammars

We first report on the results of an exhaustive exploration of all small stochastic context-free grammars in Stochastic RNA Form. The goal is to shed light on the following questions: *Does RNA favor a specific shape of grammars? If so, which? How different is the compression ability of different grammars of the same size? Are the human-expert chosen grammars best possible for their size?*

For this, we implemented an exhaustive generation algorithm that can iterate over all possible grammars in Stochastic RNA Form by constructing for a given number of nonterminals all possible rules in SRF. Then we can iterate over all possible subsets, or all subsets of a given size to generate all possible SRF grammar with these parameters. Note that for  $k$  nonterminals, the total number of possible SRF rules is  $k^3$  type (i) rules,  $k$  type (ii) rules,  $k^2$  type (iii) rules and  $\binom{k}{2}$  type (iv) rules.

The number of grammars is large ( $> 2^{k^3}$ ) even for moderate  $k$ ; however many grammars do not even allow to parse all RNA structures (see Table 5.1) and can be discarded quickly. For that, we use a tiny dataset ‘‘parsable’’ of short RNA structures; any grammar that fails to parse this dataset is skipped. For the remaining grammars, we determine the normalized compressed size, i.e., the number of bits in the compressed representation divided by the number of bases of the RNA; both using adaptive and static rule-probability models. The dataset we use is the ‘‘benchmark’’ dataset from Dowell and Eddy [8]. As an efficient first filter we reduce it to a randomly chosen 10% subsample; we determined in preliminary experiments that this predicts the bits-per-base value on the entire dataset to within  $\pm 1\%$ . We hence determine the best grammars from this 10% subsample and then evaluate the most successful grammars on the full benchmark dataset.

#NTs	#rules	# SRF grammars	#parsing gr.	(%)	best bits-per-base
1	3	1	1	100%	3.6241
2	1	15	0	0%	—
2	2	105	0	0%	—
2	3	455	1	0.2%	3.6890
2	4	1365	28	2%	3.1424
2	5	3003	201	7%	2.9969
2	6	5005	783	16%	2.9762
2	7	6435	1831	28%	2.9927
2	8	6435	2801	44%	3.0088
2	9	5005	2953	59%	3.0516
2	10	3003	2198	73%	3.0668
2	11	1365	1158	85%	3.0856
2	12	455	424	93%	3.1229
2	13	105	103	98%	3.3456
2	14	15	15	100%	3.5364
2	15	1	1	100%	3.8076
3	1	42	0	0%	—
3	2	861	0	0%	—
3	3	11 480	1	0.00001%	3.6891
3	4	111 930	71	0.00001%	3.1424
3	5	850 668	2015	0.002%	2.9866
3	6	5 245 786	33 170	0.006%	2.6620
3	7	26 978 328	377 522	0.013%	2.5495
3	8	118 030 185	3 212 691	0.027%	2.5582

Table 5.1: Overview of the overall number of grammars of certain sizes and the number of grammars that are able to parse all RNA in our benchmark dataset. The best bits-per-base gives an indicative compression performance (adaptive rule-prob model on the 10% sample of “benchmark”). Note that the number of possible rules for 2 nonterminals is 15 and for 3 nonterminals is 42.

### 5.3.1 Distribution of compression ability

Using each possible small grammar (all grammars with 2 nonterminals, and all grammars with 3 nonterminals and at most 6 rules) to compress the 10% sample of the benchmark dataset, we obtain the normalized compressed size (in bits per base) using the adaptive and static rule-probability models. Figure 5.1 shows scatter plots of these results, split by grammar size. There is a clear correlation of the two measures, meaning that grammars mostly live on a single scale from better to worse compression approximately reflecting both models.

Moreover, the vast majority of grammars give rather poor compression. This is even more visible in Figure 5.2 which shows the distribution of bits-per-base for all grammars up to 3 nonterminals and 6 rules (same data as in Figure 5.1). We expect the normalized compressed size to be at least 2 bits per base, since the primary structure of RNA is not known to be substantially compressible (and thus needs roughly 2 bits per character), and we need some additional information to encode the structure on top of that. The vast majority of grammars just realize a compressed size around  $\lg(4 \cdot 3) \approx 3.58$ , which corresponds to storing each of the pairs of terminals (4 bases, 3 structure symbols) independently with uniform likelihood. Note that this is (approximately) the compression achieved by the trivial grammar with a single nonterminal and the three rules  $A \rightarrow (A) \mid \bullet \mid AA$ . It seems hence indeed the case that most grammars are not able to pick up the structure of RNA at all, and only a very small number of grammars achieve substantially better compression than almost all other grammars. Figure 5.3 shows some of these.

### 5.3.2 Comparison with expert-curated grammars

We compare the results from newly found grammars with the expert-curated RNA grammars collected in the literature in Table 5.2; the grammars can be found in Figure 5.3 resp. the appendix of [39]. Although by a narrow margin, the best grammar for adaptive rule probabilities known is our new grammar  $G_{6,10}^\dagger$ , clearly demonstrating that human-expert grammars are not necessarily best possible!

Two expert grammars are in the range where exhaustive exploration is still feasible and both are *not* the best possible grammar, even in their (flyweight) category:  $G_L$  by Liu et al. [32] is surpassed by a fair margin (2.95 vs 3.12 bits per base) by other grammars with 2 nonterminals. It is best possible, though if we insist on exactly 4 rules. For  $G_5$  from Dowell and Eddy [8], we find a better grammar of the same size, again with a substantial

Grammar	adaptive	static	#NTs	#rules	grammar size
grammar-1NT	3.6241	3.4731	1	3	3
$G_{L'}$ (Liu et al.)	3.1229	3.0201	2	4	18
$G_{2,5}^*$ (new)	2.9699	2.8200	2	5	19
$G_{2,6}^*$ (new)	2.9494	2.8011	2	6	20
$G_5$ (Dowell, Eddy)	2.8368	2.7423	3	6	32
$G_3$ (Dowell, Eddy)	2.5804	2.4549	5	11	69
$G_6$ (Dowell, Eddy)	2.4957	2.3687	5	9	61
$G_4$ (Dowell, Eddy)	2.7138	2.5974	6	11	78
$G_1$ (Dowell, Eddy)	3.0779	2.8956	6	13	87
$G_2$ (Dowell, Eddy)	2.9723	2.5525	18	296	1742
$G_7$ (Dowell, Eddy)	2.6343	2.3333	38	321	2883
$G_8$ (Dowell, Eddy)	2.7213	2.4561	39	322	2926
$G_5$ (Nebel, Scheid)	2.5045	2.2876	108	244	3396
$G_{3,6}^*$ (new)	2.6329	2.3797	3	6	32
$G_{3,7}^*$ (new)	2.5287	2.3878	3	7	34
$G_{6,10}^\dagger$ (new)	2.5091	2.3835	6	10	73
$G_{6,10}^{\dagger'}$ (new)	2.4902	2.3835	4	7	45

Table 5.2: Normalized compressed size (bits per base) for some newly found small grammars and the grammars from [8, 32, 34]. All results are for the “benchmark” dataset from [8]. “grammar size” is the #bits needed to encode the grammar using the following scheme: encode  $k$  (#NTs) in Elias gamma code, then  $r$  (# rules) in binary and finally the size- $r$  subset of rules using an enumeration of all size- $r$  subsets.

improvement of compression ability.

## 5.4 Conclusion

We formulated the competition for the best joint RNA compression grammar and gave first improvements on the best known grammars. Unfortunately, the combinatorial explosion of possible grammars and the fact that most grammars do not compress RNA meaningfully turns the problem into the search for a needle in a haystack. Further progress in future work will likely have to come from heuristics, potentially including learning systems.

The fact that the expert grammars are (in several ways) not best possible indicates that there are still unexplored patterns in RNA structures to be understood. Could analyzing patterns in RNA structures, alongside grammars that achieve high compression ratios, inspire more effective heuristics for optimizing grammar search?

The contest to find the best grammar for adaptive rule probabilities in particular has the potential to deepen our understanding of RNA structures and might lead to interesting compression methods on the path towards optimal grammars for RNA compression and prediction.

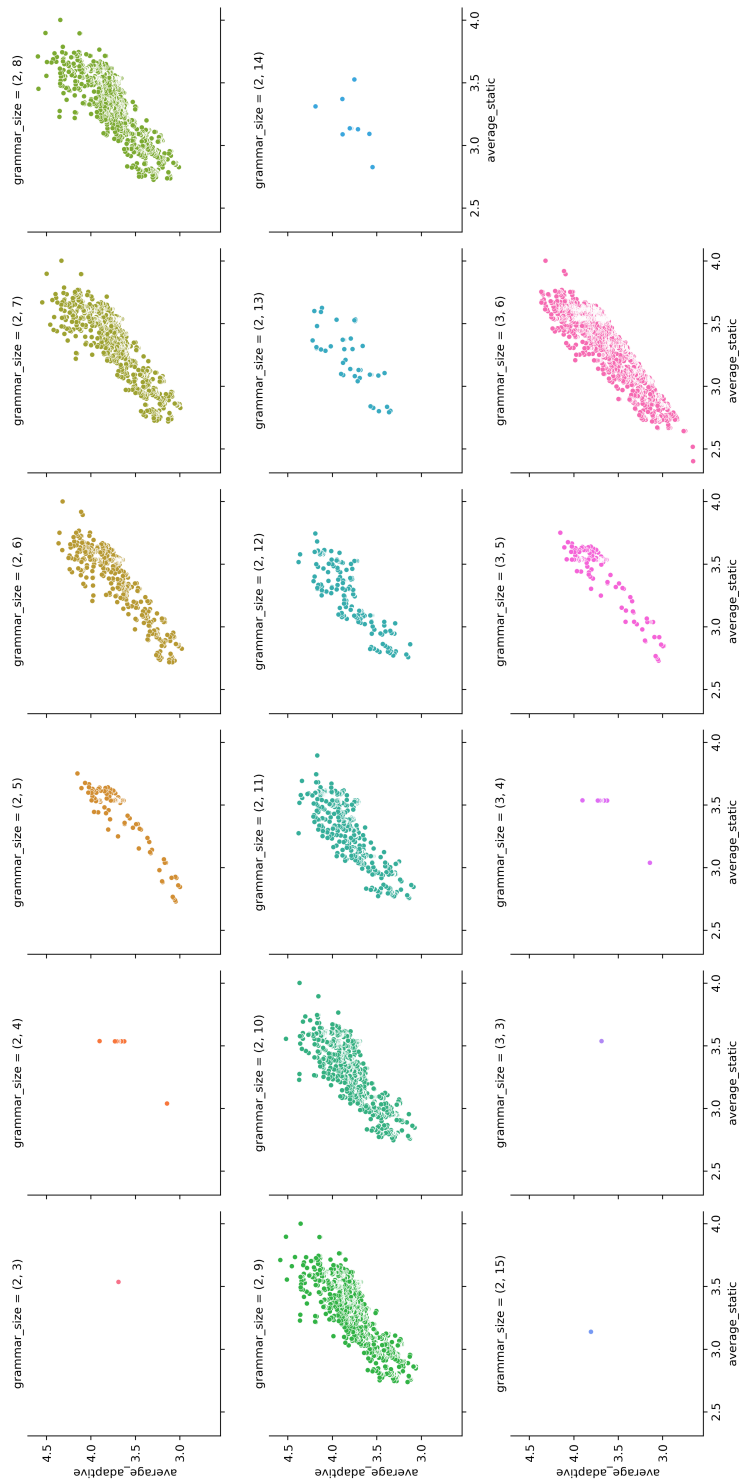


Figure 5.1: Normalized average compressed size (in bits per base) for all grammars of the given size ( $\#NTs$  (Nonterminals),  $\#rules$ ) on 10% sample of the “benchmark” dataset from [8]. Each dot is one grammar; the  $x$ -coordinate is using the static rule-probability model, with rule counts on the same dataset; the  $y$ -coordinate uses the adaptive rule-probability model.

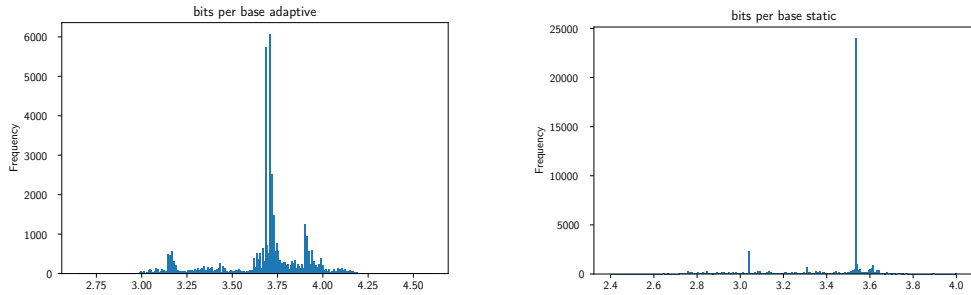


Figure 5.2: Histogram of the normalized average compressed size (in bits per base) for all grammars from Figure 5.1 on the 10% subsample of the benchmark dataset from Dowell and Eddy [8] using the adaptive (left) resp. static (right) rule-probability model.

$G_{2,6}^*$	$G_{2,5}^*$	$G_{3,6}^*$	$G_{3,7}^*$	$A_2 \rightarrow A_1 A_2$
$A_0 \rightarrow \bullet$	$A_0 \rightarrow \bullet$	$A_0 \rightarrow \bullet$	$A_0 \rightarrow \bullet$	
$A_0 \rightarrow A_1 A_0$	$A_1 \rightarrow A_1 A_1$	$A_0 \rightarrow (A_2)$	$A_0 \rightarrow (A_0)$	
$A_0 \rightarrow A_1 A_1$	$A_1 \rightarrow A_0 A_1$	$A_1 \rightarrow A_0$	$A_0 \rightarrow A_1 A_2$	
$A_1 \rightarrow A_0 A_1$	$A_1 \rightarrow (A_1)$	$A_2 \rightarrow (A_2)$	$A_1 \rightarrow \bullet$	
$A_1 \rightarrow (A_1)$	$A_1 \rightarrow \bullet$	$A_2 \rightarrow A_1$	$A_1 \rightarrow A_0$	
$A_1 \rightarrow A_0$			$A_2 \rightarrow A_1$	

Figure 5.3: Newly identified grammars;  $G_{k,r}^*$  is the best grammar with  $k$  NTs (Nonterminals) and  $r$  rules from exhaustive search.

## Chapter 6

# Heuristics for Autogenerating Grammars

### 6.1 Introduction

In this chapter we record the attempts at refining our search for better grammars by applying 2 heuristics, Random Search and Local Search:

1. **Random Search.** For this heuristic search we produced 3 types:
  - General Random Search.
  - Nonterminal Based Search: In order to get finer control over which grammars are tried this search method limits the number of rules per Nonterminal (Left hand side) or
  - Rule Type Based which limits the number of rules per rule type (i.e. SRF rules Section 5.2).
2. **Local Search.** This heuristic method searches around the neighbourhood of a known good grammar or a random grammar. We produced 2 types of Local Search heuristic.
  - Local Neighbour search.
  - Random Local Search.

## 6.2 Random Search

In Chapter 5 we saw from the exhaustive exploration that despite an increasing fraction of grammars not parsing all RNA and most grammars only giving trivial compression, compression quality does increase with (moderate) increase in grammar size (see Table 5.1 and Figure 5.1). It is therefore desirable to be able to search among larger grammars than accessible via exhaustive exploration. As a simple first step towards that, we implemented The General Random Search.

### 6.2.1 General Random search

This search does not aim at exploring every grammar by obtaining all possible combinations of the SRF rules  $S_n$  generated based on the number of Nonterminals  $n$  supplied rather it randomly selects a subset of  $k$  rules and obtains a grammar, and its compression ability is then tested, to determine if it is to be retained or discarded. Listing 4 is the pseudocode for the General Random Search

Listing 4: General Random Search

```

1 void RandomGrammarSearch() {
2     while (true) {
3         int n = rand() % 10 + 1; // Random n (1 to 10)
4         int k = rand() % (pow(n, 3) + n + pow(n, 2) + (n * (n - 1)) / 2) + 1;
5
6         RuleSet G_r = SelectRandomRules(GenerateSRFRules(n), k);
7
8         printf("n = %d, k = %d, Compression Ratio = %.2f\n",
9             n, k, TestCompressionAbility(G_r));
10    }
11 }

```

To efficiently test the grammars compression ability , we first check grammars for the ability to parse a tiny dataset of artificial RNA. Among those who parse that correctly, we evaluate their compression ability on a small dataset of short RNA and those who perform on this at least as good as the worst of the current top  $m$  grammars, we evaluate on the benchmark dataset. The top  $m$  grammars are always chosen based on the benchmark dataset. We ran this random exploration for grammars with 3–10 nonterminals and different numbers of rules, on **six** Intel<sup>®</sup> Xeon<sup>®</sup> CPU E5-2690 v3 @ 2.60GHz running on 64-bit

6.8.0-51-generic Unix operating system for roughly 336 hours. The machines explored over 24 billion grammars! The best grammar found in the process was  $G_{6,10}^\dagger$ , see Tables 6.1 and 6.2. Note that  $G_{6,10}^\dagger$  contains 2 nonterminals that are dead ends for any derivation:  $A_3$  has no rules, and hence cannot ever be replaced.  $A_2$  only has rules involving  $A_3$  and hence can likewise never be resolved to terminals. Removing those rules (and nonterminals) gives  $G_{6,10}'^\dagger$ .

$G_{6,10}^\dagger$	
$A_0 \rightarrow (A_5)$	$A_1 \rightarrow \bullet$
$A_1 \rightarrow A_0$	$A_2 \rightarrow A_3 A_5$
$A_2 \rightarrow A_4 A_3$	$A_4 \rightarrow A_1$
$A_4 \rightarrow A_4 A_1$	$A_5 \rightarrow A_0$
$A_5 \rightarrow A_0 A_2$	$A_5 \rightarrow A_4$

Table 6.1: Grammar  $G_{6,10}^\dagger$ . Without the grey rules (dead end rules) gives the best grammar found so far  $G_{6,10}'^\dagger$

Grammar	adaptive	static	#NTs	#rules	grammar size
$G_5$ (Dowell, Eddy)	2.8368	2.7423	3	6	32
$G_3$ (Dowell, Eddy)	2.5804	2.4549	5	11	69
$G_6$ (Dowell, Eddy)	2.4957	2.3687	5	9	61
$G_4$ (Dowell, Eddy)	2.7138	2.5974	6	11	78
$G_1$ (Dowell, Eddy)	3.0779	2.8956	6	13	87
$G_2$ (Dowell, Eddy)	2.9723	2.5525	18	296	1742
$G_7$ (Dowell, Eddy)	2.6343	2.3333	38	321	2883
$G_8$ (Dowell, Eddy)	2.7213	2.4561	39	322	2926
$G_S$ (Nebel, Scheid)	2.5045	2.2876	108	244	3396
$G_{6,10}^\dagger$ (new)	2.5091	2.3835	6	10	73
$G_{6,10}'^\dagger$ (new)	2.4902	2.3835	4	7	45

Table 6.2: Normalized compressed size (bits per base) for newly found random grammars and the grammars from [8, 32, 34]. All results are for the “benchmark” dataset from [8]. “grammar size” is the #bits needed to encode the grammar using the following scheme: encode  $k$  (#NTs) in Elias gamma code, then  $r$  (# rules) in binary and finally the size- $r$  subset of rules using an enumeration of all size- $r$  subsets.

## 6.2.2 Nonterminal Based Search

In applying this heuristics, the steps in Listing 5 are taken. After inputting the number of nonterminals  $n$ , heuristic number  $h$ , compression ratio  $c$ , of the best known grammar, the SRF rules are generated (lines 108 - 109) then they are grouped by Nonterminals (i.e. all rules with same left hand side are grouped together (lines 111-115)). From each group, at most  $h$  rules are randomly selected (lines 121 - 126), the selected rules are put together to produce a grammar  $G$  (line 129). The compression ratio  $c_G$  of  $G$  is determined (139-143). If  $c_G$  is less than  $c$ ,  $G$  is saved as the best known grammar and the value of  $c_G$  is stored in  $c$ .

Listing 5: Nonterminal Based Search

```

1  // Nonterminal-Based Grammar Search Algorithm
2  // Inputs:
3  //   - int n: Number of Nonterminals
4  //   - int h: Heuristic number
5  //   - float c: Compression ratio for the best known grammar
6  //   - Dataset d: Sample RNA dataset
7  // Outputs:
8  //   - Updated grammar G with improved compression ratio

10 void NonTerminalBasedSearch(int n, int h, float c, Dataset d) {
11     // Step 1: Generate all SRF rules for n
12     RuleSet SRF_Rules = GenerateSRFRules(n);

14     // Step 2: Group rules by their left-hand-side Nonterminals into n groups ↵
15     //           G1, ..., Gn
16     RuleSet Groups[n]; // Array to hold n groups
17     for (int i = 0; i < n; i++) {
18         Groups[i] = GroupRulesByNonTerminal(SRF_Rules, i);
19     }

20     // Step 3: Main loop
21     while (true) {
22         RuleSet G_r = {}; // Initialize G_r to store selected rules

24         // Step 4: Randomly select rules from each group

```

```

25     for (int i = 0; i < n; i++) {
26         int k_i = RandomValue(1, h); // Randomly pick k_i in [1, h]
27         RuleSet G_i_k = SelectRandomRules(Groups[i], k_i); // Select k_i ←
                rules from G_i
28         G_r = Union(G_r, G_i_k); // Add selected rules to G_r
29     }

31     // Step 5: Produce a new grammar G using the rules in G_r
32     Grammar G = ProduceGrammar(G_r / dead_end_rules); // Grammar G has no ←
                dead end rules

34     // Step 6: Test if G parses the RNA dataset
35     if (!ParsesRNA(G, d)) {
36         continue; // Skip this grammar and move to the next iteration
37     }

39     // Step 7: Compute the compression ratio for G
40     float c_G = ComputeCompressionRatio(G);

42     // Step 8: If the compression ratio is worse, skip
43     if (c_G > c) {
44         continue;
45     }

47     // Step 9: save the new grammar and update the best compression ratio
48     storeGrammar(G);
49     c = c_G;
50 }
51 }

```

### 6.2.3 Rule Type Based Search

This heuristic search is very similar to the Nonterminal based search except that the rules generated are grouped into 4 groups based on the SRF rule type (Section 5.2). At most  $h$  rules are selected from each group to produce a Grammar  $G$ , and its compression is determined to find out if it outperforms the best known grammar. The detailed steps are in Listing 6

Listing 6: Rule Type Based Search

```

1 // Rule-Type-Based Grammar Search Algorithm
2 // Inputs:
3 // - int n: Number of Nonterminals
4 // - int h: Heuristic number
5 // - float c: Compression ratio for the best known grammar
6 // - Dataset d: Sample RNA dataset
7 // Outputs:
8 // - Updated grammar G with improved compression ratio

10 void RuleTypeBasedSearch(int n, int h, float c, Dataset d) {
11     // Step 1: Generate all SRF rules for n
12     RuleSet SRF_Rules = GenerateSRFRules(n);

14     // Step 2: Group rules into 4 groups (G1, G2, G3, G4) based on SRF rule types
15     RuleSet G1 = GetGroup(SRF_Rules, 1); // Rules of type 1
16     RuleSet G2 = GetGroup(SRF_Rules, 2); // Rules of type 2
17     RuleSet G3 = GetGroup(SRF_Rules, 3); // Rules of type 3
18     RuleSet G4 = GetGroup(SRF_Rules, 4); // Rules of type 4

20     // Step 3: Main loop
21     while (true) {
22         RuleSet G_r = {}; // Initialize G_r to store selected rules

24         // Randomly select rules from each group
25         for (int i = 1; i <= 4; i++) {
26             int k_i = RandomValue(1, h); // Randomly pick k_i in [1, h]
27             RuleSet G_i = (i == 1) ? G1 : (i == 2) ? G2 : (i == 3) ? G3 : G4;
28             RuleSet G_i_k = SelectRandomRules(G_i, k_i); // Select k_i rules
29             G_r = Union(G_r, G_i_k); // Add selected rules to G_r
30         }

32         // Step 4: Form a new grammar G using the rules in G_r
33         Grammar G = ProduceGrammar(G_r / dead_end_rules); // Grammar G has no ←
                dead end rules

35         // Step 5: Check if G parses the RNA dataset
36         if (!ParsesRNA(G, d)) {

```

```

37         continue; // Skip this grammar and move to the next iteration
38     }

40     // Step 6: Compute the compression ratio for G
41     float c_G = ComputeCompressionRatio(G);

43     // Step 7: If the compression ratio is worse, skip
44     if (c_G > c) {
45         continue;
46     }

48     // Step 8: save the new grammar and update the best compression ratio
49     storeGrammar(G);
50     c = c_G;
51 }
52 }

```

### 6.3 Local Neighbour Search

The Local Neighbour Search determines all neighbours of an input grammar  $G$ , and tests the compression ability of each one. In this search, 2 grammars  $G_1$  and  $G_2$  are defined as neighbours if they differ by 1 rule i.e. all rules in  $G_1$  and  $G_2$  are same except either has one rule less or more than the other or if they have same number of rules then there exist exactly 1 rule in  $G_1$  i.e completely different from exactly 1 rule in  $G_2$ . The steps taken in this heuristic search are shown in Listing 7:

Listing 7: Local Neighbour Search

```

1 // Local Neighbour Search Algorithm
2 // Inputs: Grammar G, compression ratio c, dataset d, rule set R_G
3 // Outputs: Updated grammar G with improved compression ratio

5 void LocalNeighbourSearch(Grammar G, float c, Dataset d, RuleSet R_G) {
6     RuleSet S_r = GetAllSRFRules();
7     RuleSet R_prime_G = SubtractRuleSets(S_r, R_G);

9     for (int i = 0; i < SizeOf(R_G); i++) {

```

```

10     Rule r = GetRuleAtIndex(R_G, i);

12     for (int j = 0; j < SizeOf(R_prime_G); j++) {
13         Rule r_prime = GetRuleAtIndex(R_prime_G, j);

15         RuleSet S_R[] = {
16             Union(Difference(R_G, r), r_prime), // Replace
17             Difference(R_G, r_prime),          // Decrement
18             Union(R_G, r_prime)                // Increment
19         };

21         for (int k = 0; k < 3; k++) {
22             Grammar G_L = FormGrammar(S_R[k] / dead_end_rules); //Grammar ←
                G_L has no dead end rules
23             float c_L = ComputeCompressionRatio(G_L, d);

25             if (c_L < c) {
26                 G = G_L;
27                 c = c_L;
28                 i = -1; // Restart outer loop
29                 break;
30             }
31         }
32     }
33 }
34 }

```

## 6.4 Random Local Search

The Random Local Search algorithm (Listing 8) selects a random neighbor  $G_L$  (as defined in the Local Neighbour Search in Section 6.3) of the input grammar  $G$  (lines 276-292) and computes its compression ratio  $c_L$ . If the positive difference between  $c_L$  and the input grammar's compression ratio  $c$  is less than a small threshold  $\delta$ , then  $G_L$  becomes the new input grammar, and  $c_L$  becomes the new input compression ratio. The search for the next random neighbor then continues.

Listing 8: Random Local Search

```

1 // Random Local Search Algorithm
2 // Inputs:
3 // Grammar G with compression ratio c
4 // Sample RNA dataset d
5 // Rule set R_G
6 // Threshold delta (0 < delta < 1)
7 // Outputs:
8 // Updated grammar G with improved or near-optimal compression ratio

10 function RandomLocalSearch(G, c, d, R_G, delta) {
11     // Obtain all SRF rules for the grammar size
12     S_r = GetAllSRFRules();
13     R_prime_G = S_r - R_G; // Rules in S_r but not in R_G

15     while (true) {
16         // Randomly select a modification type: 0 = replace, 1 = add, 2 = delete
17         int i = RandomValue(0, 2);

19         // Generate a new rule set R based on the random choice
20         if (i == 0) {
21             // Replace a random rule in R_G with one from R_prime_G
22             R = ReplaceRandomRule(R_G, R_prime_G);
23         } else if (i == 1) {
24             // Add a random rule from R_prime_G to R_G
25             R = AddRandomRule(R_G, R_prime_G);
26         } else if (i == 2) {
27             // Delete a random rule from R_G
28             R = DeleteRandomRule(R_G);
29         }

31         // Form a new grammar G_L using the updated rules R
32         G_L = FormNewGrammar(R / dead_end_rules); // Grammar G_L has no dead end ↵
            rules;

34         // Compute the compression ratio for G_L
35         c_L = ComputeCompressionRatio(G_L, d);
36         store G; // store G

```

```

37      // If the new compression ratio is better or close to the previous one
38      if (c_L < c (c_L - c) < delta) {

40          G = G_L; // Update the grammar
41          c = c_L; // Update the compression ratio
42      }
43 }

45 }
```

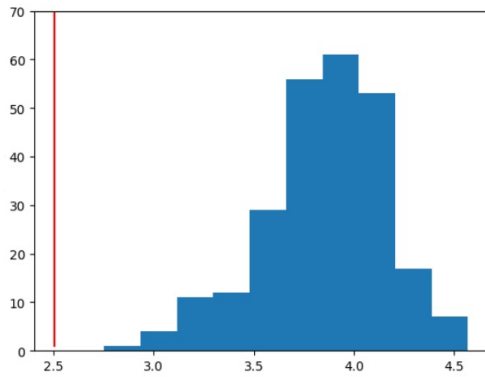
## 6.5 Snap shots of Results obtained

Experiments were carried out for the Nonterminal Based, Rule type Based, Local Search and Random Local Search heuristics. Several details of the experiments and results are covered here.

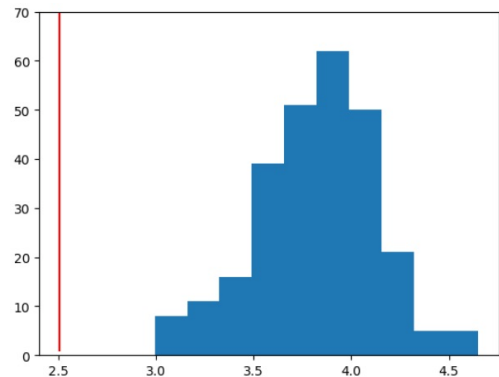
### 6.5.1 Results for Nonterminal Based search

Experiments were run for 3 and 4 Nonterminals, and for each of the Nonterminals we tried heuristic numbers 1, 2 and 3. Each of the combinations had 1000 iterations to find unique grammars. In Figure 6.1 the ranges of unique grammars and their compression ratios are shown in each chart. The red vertical line shows the compression ratio for the best known grammar. Figure 6.1 shows snapshots of some of the results obtained.

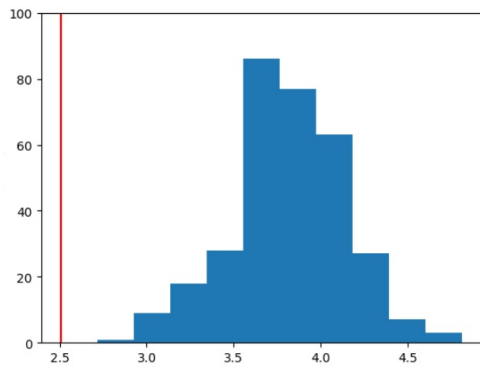
In Figures 6.1(a)-(f), each of the 6 histograms have single modal ranges, which fall between the compression ratios 3.5-4.0. Although the idea behind the design of the Nonterminal Based search is to refine and control the search for grammars which are tested for compressibility, the distribution of grammars produced, is similar to the distribution of grammars produced in the Exhaustive Search (see Figure 5.2). However, the experiments conducted so far, are not sufficient to conclude that the Nonterminal Based Search is no better than the Exhaustive Search. Using a larger range of nonterminals, heuristic number and iteration, one can make a more conclusive decision on the efficiency of the Nonterminal Based Search.



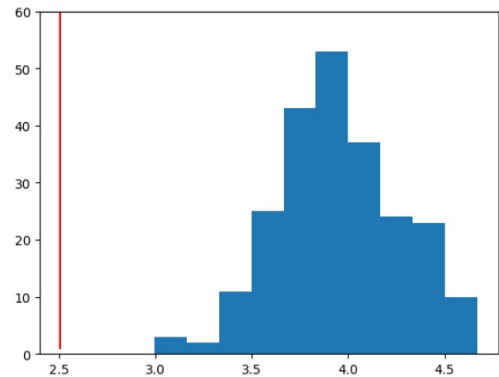
(a) 3 Nonterminals, NT-Based,  
H-Number=1, 1000 iterations, best-ratio=  
2.7543



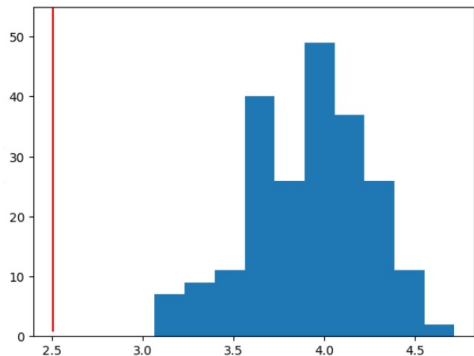
(b) 3 Nonterminals, NT-Based,  
H-Number=2, 1000 iterations, best-ratio=  
2.9938



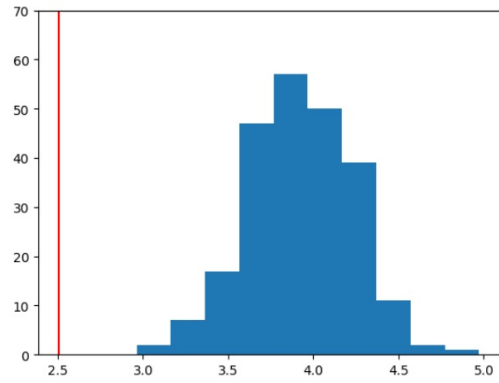
(c) 3 Nonterminals, NT-Based,  
H-Number=3, 1000 iterations, best-  
ratio=2.7176



(d) 4 Nonterminals, NT-Based,  
H-Number=1, 1000 iterations, best-  
ratio=2.9933



(e) 4 Nonterminals, NT-Based,  
H-Number=2, 1000 iterations, best-  
ratio=3.0661



(f) 4 Nonterminals, NT-Based,  
H-Number=3, 1000 iterations, best-  
ratio=2.9626

Figure 6.1: Results for Nonterminal based heuristic experiments. Each graph shows the distribution of the compression ratio of the generated grammars. Each bar shows the number of grammars with compression ratios within the range. The vertical line shows the compression ratio of the best known grammar.

### 6.5.2 Results for Rule Type Based Search

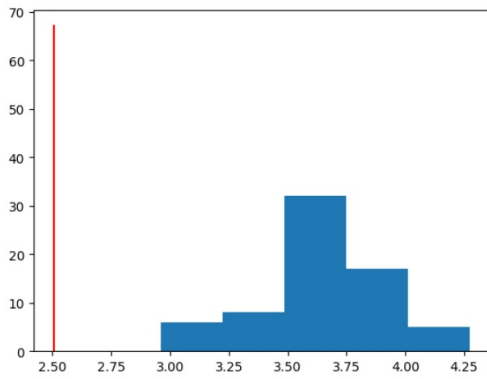
We ran experiments for the Rule Type Based Search using 3 and 4 nonterminals and heuristic numbers 1, 2 and 3 for the rules selection per rule type. Each combination was run for 1000 iterations. The results are shown in Figure 6.2, with the best known compression ratio shown with the vertical red line. There are a number of similarities between the results in Figures 6.1(a)-(f) and 6.2(a)-(f). Their modal values lie between 3.5-4.0. The best compression ratios for each experiment can be found in the range 2.7-3.1. Also with the Rule Type Based Search, more experiments would reveal the future potential of its efficacy in obtaining grammars with better compression.

### 6.5.3 Local Search Results

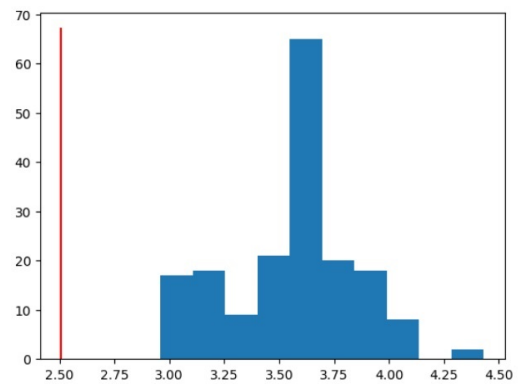
In the experiments for the local search we obtained the neighbours (i.e. all grammars, which differ from the input grammar by one rule), plotted the distribution of the compression ratio of the neighbours and indicated the compression ratio of the best known grammar using a vertical red line as shown in Figure 6.3. The input grammars used are the best known grammar i.e.  $G_{6,10}^\dagger$ ' (see Table 6.1), second, third and fourth best grammars. The second, third and fourth best grammars were obtained by applying the local search to the best known grammar, these 3 performed best in terms of compression-ratio and are thus ranked in that order(Appendix B shows the grammars in detail)

### 6.5.4 Radom Local Search Results

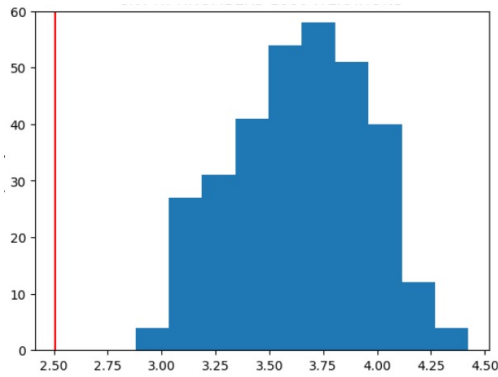
In the Random Local Search we conducted just 3 types of experiments. Starting with the best grammar, and using different seeds for the *randomValue i* (see algorithm in Listing 8), we obtained results for 100, 1000 and 10000 iterations. This is shown in Figure 6.4. The results in the Random Local Search show an interesting distribution. Although this search produces grammar which compete closely with the input grammar (the best known was used as input grammar), majority of the grammars had a compression ratio between 2.9 - 3.1. As for the result in Figure 6.4(c), which shows the Random Local Search experiment for 10000 iterations has 2 modal ranges 2.9 - 3.1 and 3.6-3.7. This result is an indication of the Random Local Search's ability to shift unpredictably to find grammars which may be better or worse than the input grammar.



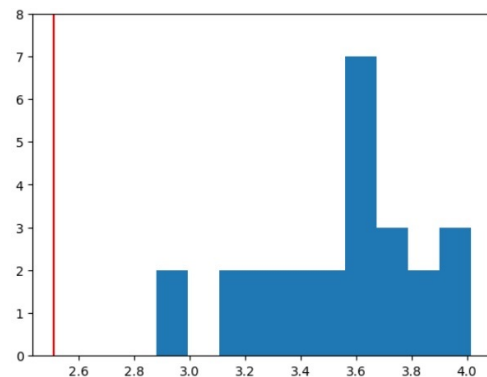
(a) 3 Nonterminals, RT-Based,  
H-Number=1, 1000 iterations, best-  
ratio=2.9597



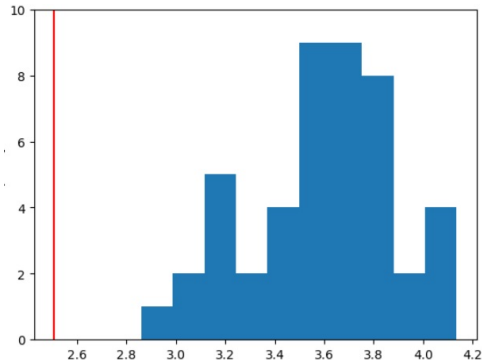
(b) 3 Nonterminals, RT-Based,  
H-Number=2, 1000 iterations, best-  
ratio=2.9589



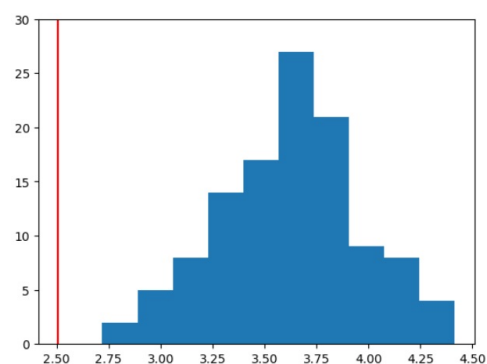
(c) 3 Nonterminals, RT-Based,  
H-Number=3, 1000 iterations, best-  
ratio=2.8780



(d) 4 Nonterminals, RT-Based,  
H-Number=1, 1000 iterations, best-ratio=  
2.8785



(e) 4 Nonterminals, RT-Based,  
H-Number=2, 1000 iterations, best-  
ratio=2.8607



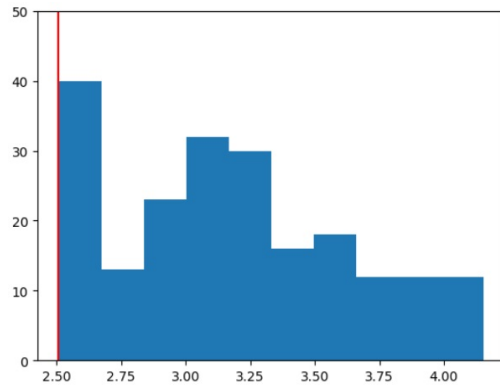
(f) 4 Nonterminals, RT-Based,  
H-Number=3, 1000 iterations, best-  
ratio=2.7193

Figure 6.2: Results for Rule-type based heuristic experiments. Each graph shows the distribution of the compression ratio of the generated grammars. The vertical line shows the compression ratio of the best known grammar.

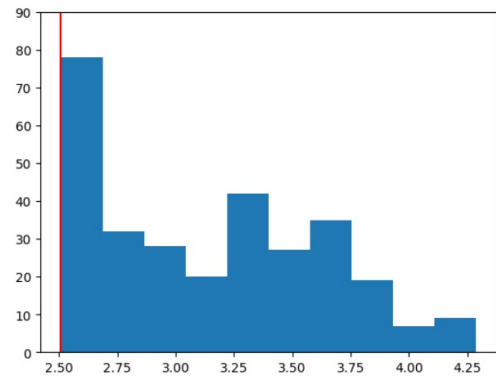
## 6.6 Summary of Results

The best grammar found, was obtained from the General Random Search. This heuristic involved taking a random number of nonterminals and a random number of rules, after exploring over 24 billion grammars we obtain the best known grammar. The other heuristics were designed to refine the General Random Search. For the experiments carried out on the Nonterminal Based, Rule Type Based and Random Local Search, majority of the grammars found, produced compression ratios between 3.0-4.0 we can see this in Figures 6.1, 6.2 and 6.4. This modal range is similar to the modal range in the exhaustive search shown in Figure 5.2, with modal range falling between 3.6-3.8.

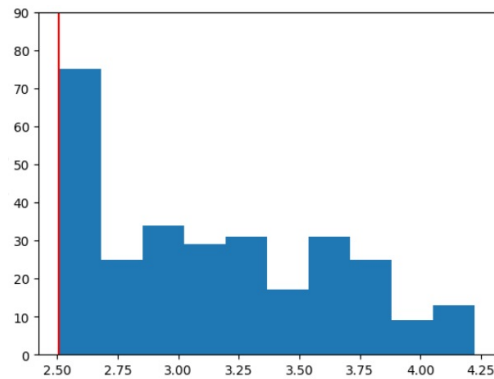
It is very interesting to see that for the Local Search, the majority of the grammars it obtained have compression ratios closely competing with the best grammar! In appendix B, we see the 9 best grammars found from the search. Improving on the Local Search heuristic may be the way forward to obtaining grammars with even better compression.



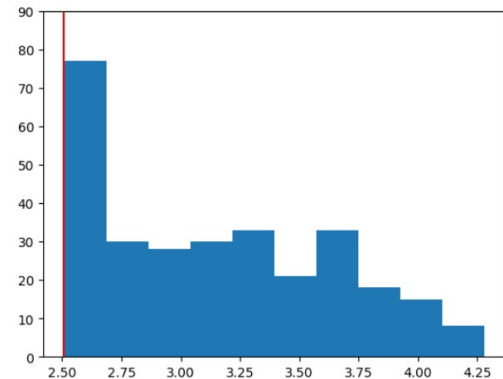
(a) Input grammar = Best-known,  
best-ratio=2.50809



(b) Input grammar = Second-Best  
(ratio=2.5099), best-ratio=2.50809

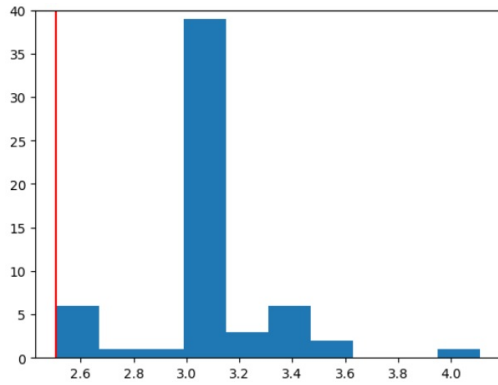


(c) Input grammar = Third-Best  
(ratio=2.5159), best-ratio=2.50809

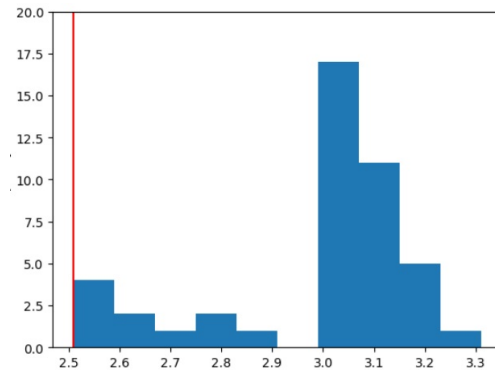


(d) Input grammar = Fourth-Best  
(ratio=2.5226), best-ratio=2.50809

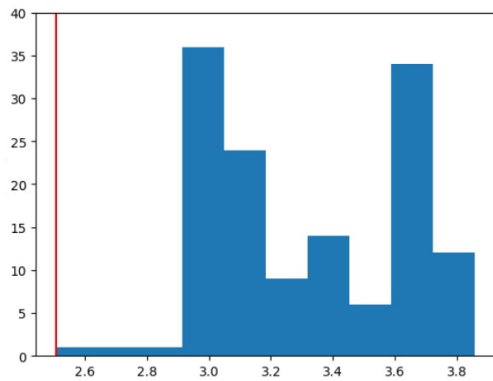
Figure 6.3: Results for Experiments using local search. Each graph shows the distribution of the compression ratio of the neighbours (using Local search) to the input grammar. The pictures show that majority of the grammars have a compression-ratio which fall near the best-known ratio



(a) Input grammar = Best-known  
(ratio=2.50809), best-ratio-found=2.5652  
100 iterations



(b) Input grammar = Best-known  
(ratio=2.50809), best-ratio-found=2.5197  
1000 iterations



(c) Input grammar = Best-known  
(ratio=2.50809), best-ratio-found=2.6610  
10000 iterations

Figure 6.4: Results for Random Local Search running for 100, 1000 iterations and 10000 iterations. Each graph shows the distribution of the compression ratio of the grammars found using the Random Local search. The input grammar is the best known grammar.

## Chapter 7

# The Software Development Process

### 7.1 Introduction

In this chapter we give an insight to the process involved in producing the software which gave the results in this research. The IEEE (Institute for Electrical and Electronics Engineers) states out a Standard for Developing Software[26]. The standard states a pre-development (Concept Exploration, System Allocation), development (Requirements, Design, Implementation), post-development (Installation, Operation and Support Maintenance and Retirement) and integral processes (Verification and Validation, Software Configuration Management, Documentation Development and Training) . In developing the software in this research, we went through 7 of the steps in this standard: Concept Exploration and Requirements, Design, Implementation, Verification, Validation and Documentation.

### 7.2 Concept Exploration and Requirements

At the start of the project (about 4 years ago), we had some clear ideas:

- Reproduce Liu et al.'s RNACompress [32], to provide a version that is readily available.
- Produce the Arithmetic Coding Compressor (AC-Compressor), an RNA compression software (without a parser) with the application of arithmetic encoding to outperform the then state of the art in RNA sequence and structured data compression [32].

Later re-organise the software and introduce the Earley Parser (Section 2.2.5) in the software.

- Investigate the correlation between Prediction quality of a grammar, with its compression ability by incorporating in the software a package for RNA structure prediction

In the third year, we decided to replace the Earley Parser with the SRF parser, which was suitable for generating grammars and parsing the auto-generated grammars. We aimed at the following:

- Produce another parser: SRF (Stochastic RNA Normal Form) Parser.
- Create a package which automatically generates SRF grammars.

## 7.3 Design and Implementaion

### 7.3.1 Design and Implementation to Reproduce RNACompress

As stated in Section 2.4.1. RNACompress uses a simple RNA grammar applying an LL(1) parser (Left to Right, Leftmost derivation with one token look ahead per parsing step) and Huffman codes. We created a simple package with 2 classes: **RNACompress** and **RNA-Derivation**. The Liu et al grammar, the Huffman code for each rule, were hard coded into the class RNACompress. Class RNACompress is fed with the dataset containing the RNAs, it iterates through the dataset, creates an object of RNA-Derivation, the RNA is passed to the constructor of RNA-Derivation. RNA-Derivation's method **encodeRNA** obtains the derivation of the RNA, calls the method **getHuffmanCodes**, which returns the concatenated string of binary codes which represent the huffman codes for each rule in the derivation. The flowchart in Figure 7.1 shows the simple process in the design of RNACompress and the associated classes.

### 7.3.2 Initial Design for AC-Compressor

One major difference in the design of AC-compressor when compared to the Liu et al's RNA compress, is in the use of arithmetic coding rather than huffman codes. The initial design involved 5 classes **ENCODE**, **DERIVATION**, **COMPRESS**, **DECODE**, **PROBABILITIES**. To compress RNAs, the dataset of RNAs are fed to COMPRESS. COMPRESS iterates through the dataset, calls its method **encode** by passing each individual RNA to

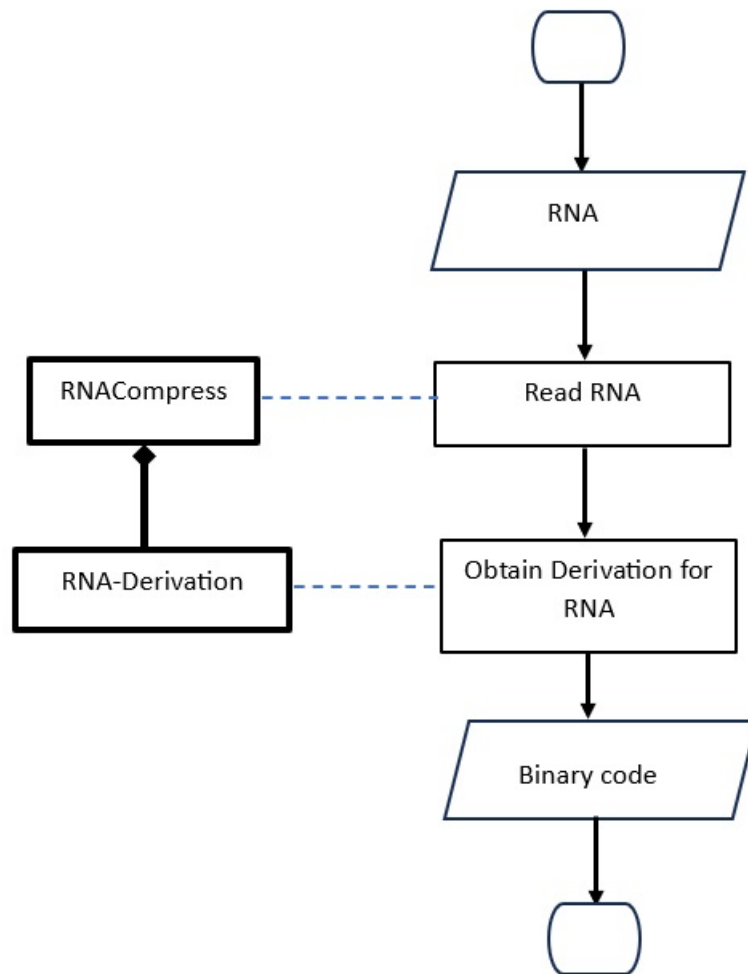


Figure 7.1: Flowchart for the design of Liu et al's RNAcompress. Dotted lines link associated classes with process

the method. Method **encode** creates an object of **ENCODE** with the RNA passed to the constructor. Within the constructor of **ENCODE** an object of class **DERIVATION** is created to obtain the LL(1) derivation of the RNA. The abstract class **PROBABILITIES** with child classes **STATIC-PROBABILITIES** and **ADAPTIVE-PROBABILITIES** supply the probabilities for each rule in the grammar. Class **ENCODE** uses arithmetic coding with the probabilities to encode the derivation. The decoding simply reverses the encoding process using the class **DECODE**. Figure 7.2 shows the flowchart and class diagrams for the AC-Compressor. The compression results of the AC-compressor were compared with Liu et al.'s RNACompress in section 3.3.1 compression output

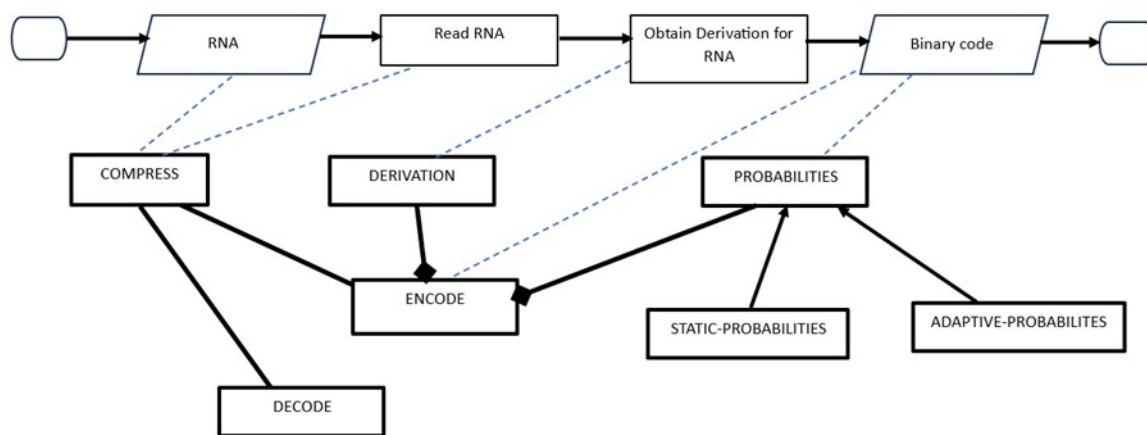


Figure 7.2: Flowchart for the encoding process of AC-Compressor. Dotted lines link associated classes with process

### 7.3.3 Modularising the Program and the Use of Earley Parser

The implementation for AC-Compressor was a good way to test the superiority of the use of arithmetic coding over huffman coding for the compression of RNA structured data. However, the grammar used [32], was hard coded into the program. Now, this is not convenient as we needed to implement compression for a considerable number of grammars from [46, 8] and more. This clearly showed a need to modularise the code and implement a parser for grammars. We decided to use Earley parser (from Section 2.2.5) - which has an easy-to-use open-source implementation. We created a parent package we called **AC-**

**Compressor** with 5 subpackages **coding**, **data**, **grammar**, **parser** and **util**. Package **coding** contains all interfaces and classes that handle the arithmetic process while **data** houses the interfaces and classes which check the validity of any dataset passed to the program and also reads the RNA data from the files. The classes in the package **grammar** are used to define the different grammar components e.g. Terminals, Nonterminals, Rules etc. All parsers are found under the package **parser** and **util** contains most of the executable classes and other functional classes. Figure 7.3 shows the subpackages of AC-Compressor and the dependencies.

### 7.3.4 RNA Secondary Structure Prediction

We investigated the relationship between the prediction of RNA secondary structure and the compression ability of different grammars (see Section 4.3).

We added another package **structureprediction**, containing all its classes which predict RNA structure and test the quality of the prediction. One key class in the package is the **PredictionWriter** (see <https://github.com/evita35/better-grammars/blob/main/src/compression/structureprediction/PredictionWriter.java>).

This class reads the RNA and converts the primary sequence into tokens using the **IgnoringSecondPartPairOfChar** class in the **grammar** package. It processes the RNA primary sequence into a list of terminals with the secondary structure excluded.

Using the Earley-style parser from [50], the secondary structure with the most likely parse tree is returned as the predicted secondary structure.

### 7.3.5 Data Package for Conformity of Dataset

In carrying out the data compression experiments we used datasets from Friemel [12], Dowell and Eddy [8]. Friemel sourced his dataset from [6] and [3] while Dowell and Eddy obtained their datasets from [43] and [2]. The data set from [3] had RNAs stored in Stockholm(STK) format and our software was designed for RNAs in dot-bracket notation. We had to create a class in **data** package to convert the RNAs from STK to dot-bracket notation. Further, the data package was designed to ensure the datasets used, conformed to a naming convention and validity checks are done on the datasets, before they are used for the experiments. The data sets can be found in our github repository: <https://github.com/evita35/better-grammars/tree/main/datasets>

### 7.3.6 An Earley Style Implementation for Prediction

In Section 4.3, we mentioned that we found bugs in the open source earley parser implementation we used for obtaining compression results. These bugs only affect RNA structure prediction accuracy. To obtain accurate predictions of RNA structure, we had to use a C++ implementation of the earley parser [50]. In order to use this implementation we had to create a class to read the grammars (which were already written in java codes) and print them into C++ readable codes. We created a class `PrintC` in the `util` package (shown in Figure 7.3), for this purpose. The printed output from this class was then fed to the C++ implementation to produce the prediction results.

### 7.3.7 Replacing Earley with SRF parser

With the RNA structure prediction results, we found an interesting connection: We found a clear correlation between compression and prediction in Chapter 4. This was a motivation for us to find better grammars (in terms of better compression). We decided to automate things, produce a parser, which can be used to parse grammars and also produce new grammars. This led to the birth of the SRF parser, explained in Section 5.2. We added the `SRF` class and `StochasticParser` interface to the `parser` package.

## 7.4 Verification, Validation and Documentation

To verify and validate the components of the AC-Compressor software, we conducted unit tests on the main classes within each component.

In the `compression` package, the unit test `EncodeNDecodeCorrectlyTest` was used to confirm that an encoded RNA sequence can be accurately decoded to recover the original RNA.

For the `coding` package, the `ExactArithmeticCoderTest` ensures that a subinterval of  $[0, 1)$  is correctly encoded into binary.

Within the `parser` component, the `SRFParserTest` verifies that the SRF parser produces an accurate derivation for a given RNA sequence.

In the `structureprediction` module, the `PPVNSensitivityTest` checks that the exact matching base pairs in both the predicted and trusted structures are used to compute the Positive Predictive Value (PPV) and sensitivity metrics. Finally, the `RandomGrammarTest` validates that the `GrammarGenerator` produces grammars capable of encoding

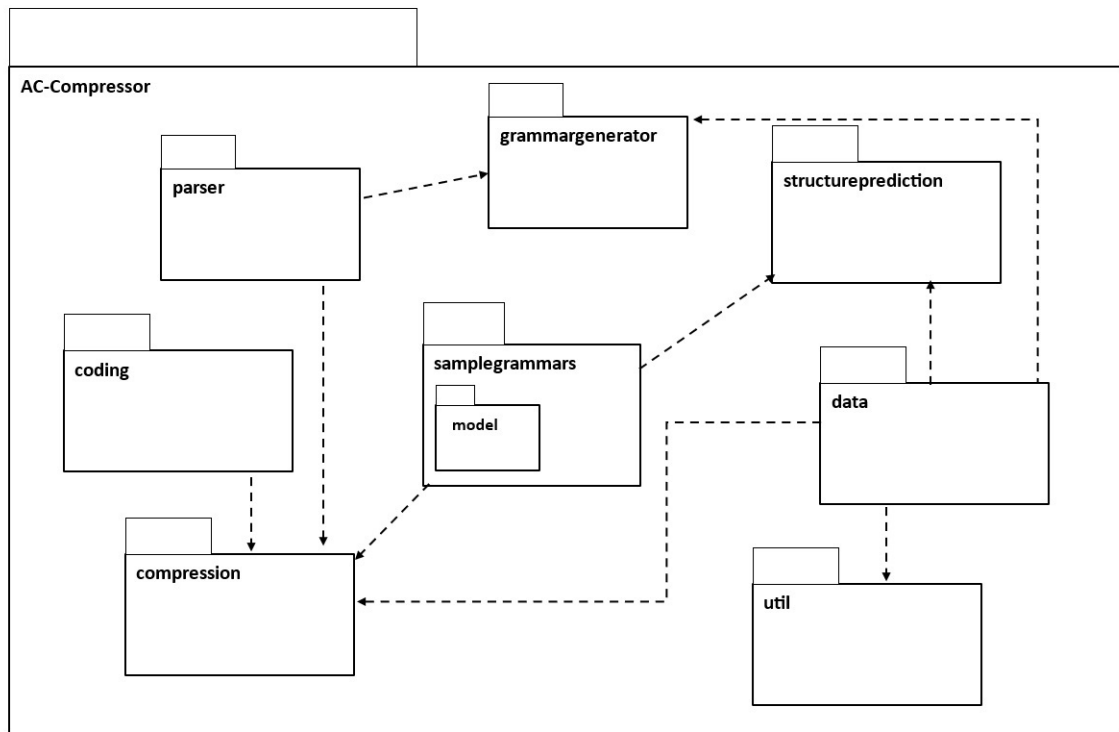


Figure 7.3: Package Diagram shows the AC-Compressor Package, with subpackages. Dotted arrows show dependency

and decoding RNA sequences correctly.

The **compression**, **coding**, **samplegrammars** packages are published on git under the joint-rna-compression repository: <https://github.com/evita35/joint-rna-compression>. This code was published as the companion code for the first publication produced from this research work [39]. The **Readme document** has important information on how to use the code and the google colab link for data from the experiments in the paper and how to reproduce them. To support our second publication [40] the entire packages in the AC-Compressor were published on <https://github.com/evita35/better-grammars.git> with a link to google colab page with necessary information to reproduce experiments in the paper.

## 7.5 Automatically Generated Grammars and Future Development

The last package included to the design was the **grammargenerator**. The classes in this package implement the heuristics for generating grammars in Chapter 6. Currently, we have code that compresses RNA, predicts RNA secondary structure, uses a new parser SRF parser and automatically generates grammars. Further improvements are being explored to develop more effective heuristics for the grammar generator. A key objective for future development is improving the **grammargenerator** through enhanced heuristics.

## Chapter 8

# Conclusion and Future Work

This chapter provides a comprehensive summary of the research conducted, highlighting key findings and their significance. Additionally, it explores potential avenues for future research, discussing how the work can be extended, improved, or applied in broader contexts.

### 8.1 Summary of Findings

#### 8.1.1 Joint Compression of RNA Primary and Secondary Structure

In the work on joint compression of RNA, a clear improvement over the RNACompress [32] method for RNA Compression was shown with the application of arithmetic coding along with various refined grammars [8, 46] designed by human experts. We showed that better compression ratio can be attained. Using Friemel’s dataset we attained over 30% improvement over RNACompress with the use of our best known grammar.

#### 8.1.2 RNA Secondary Structure Prediction and Compression

With the aid of the refined grammars from [8] and the Earley style implementation of [50] we were able to establish the relationship between compression ratio and RNA secondary structure prediction. Grammars which produce better RNA compression ratio, predict RNA structure more accurately.

### 8.1.3 Automatically generated Grammars and Heuristics

Our work in automatically generated grammars has shown that we do not need to rely solely on human experts to produce meaningful grammars which mirror the physical characteristics of RNA. Our best grammar (compression wise) is automatically generated. This is quite remarkable! We applied a good number of well thought out heuristics, random grammar, rule type based, non-terminal based and local search heuristics. We obtained our best known grammar with the use of the random grammar heuristics. At the time of this writing, experiments are still being run using these heuristics.

## 8.2 Lessons Learnt

In this research work, we saw the advantage of applying arithmetic coding, to the compression of RNA structured data, the connection between compression and prediction using various Probabilistic Context-Free Grammars (PCFGs) and the use of modular programming, in the development of the software. Separating components for compression, prediction, data, coding, grammar and grammargenerator, adds simplicity, efficiency, flexibility, ease of testing, validation, verification and maintenance. Our work on automatically generated grammars shows that machines can produce grammars that outperform (in terms of RNA structure compression) those produced by human experts!

## 8.3 Future Work

There is a need to apply more ideas to the fine tuning of heuristics in our search for better grammars. More experiments need to be conducted using the current heuristics in generating better grammars. Further exploration of this work could involve analyzing the structure of grammars that have demonstrated strong compression performance, as this could inspire the development of improved heuristics. The correlation between compression and prediction needs to be reaffirmed with the use of the automatically generated grammars.

All RNAs used in this work excluded those with pseudoknots (RNAs which violate the nesting convention). Our work on compression and prediction can be extended to RNAs with pseudoknots [42]. Operations on compressed RNA without decompression is another research area that is open to further exploration. The application of machine learning and deep learning in generating probabilistic models for protein folding, compression, and

---

secondary structure prediction is another open research area. [28].

# Bibliography

- [1] I. Brierley, S. Pennell, and R. V. Gilbert. RNA Pseudoknots: Versatile Motifs in Gene Expression and Replication. *Nature Reviews Microbiology* 5, 598–610, 130(4):313–336, 2007.
- [2] J. W. Brown. The Ribonuclease P Database. *Nucleic Acids Research* 1999 Jan 1;27(1):314., 1999.
- [3] J. J. Cannone, S. Subramanian, M. N. Schnare, J. R. Collett, L. M. D’Souza, Y Du, B. Feng, N. Lin, L. V. Madabusi, K. M. Muller, N. Pande, Z. Shang, N. Yu, and R. R. Gutell. The Comparative RNA Web (CRW) Site: An Online Database of Comparative Sequence and Structure Information for Ribosomal, Intron, and Other RNAs. *BMC Bioinformatics*, 3(1):2, 2002.
- [4] X. Chen, S. Kwong, and M. Li. A Compression Algorithm for DNA Sequences and Its Applications in Genome Comparison. *Proceedings of RECOMB*, 2000:107, 2000.
- [5] X. Chen, M. Li, B. Ma, and J. Tromp. DNACompress: Fast and Effective DNA Sequence Compression. *Bioinformatics*, 18(12):1696–1698, 2002.
- [6] RNAcentral Consortium. RNAcentral 2021: Secondary Structure Integration, Improved Sequence Search and New Member Databases. *Nucleic Acids Research*, 49(D1):D212–D220, 2020.
- [7] M. Trompper (digitalheir). Probabilistic Earley parser, 2017. <https://github.com/digitalheir/java-probabilistic-earley-parser>.
- [8] R. D. Dowell and S. R. Eddy. Evaluation of Several Lightweight Stochastic Context-Free Grammars for RNA Secondary Structure Prediction. *BMC Bioinformatics*, 5(1):1–14, 2004.

- 
- [9] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids, 267-273*. Cambridge University Press, 1998.
- [10] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13:94–102, February 1970.
- [11] S. Eddy. Non-coding RNA Genes and the Modern RNA World. *Nature reviews. Genetics*, 2(12), 919–929, 2001.
- [12] J. Friemel. *Contraction-Based Compression of RNA Secondary Structures*. Bachelor’s thesis, Universität Bielefeld, 2020.
- [13] K. S. Fu and T. Huang. Stochastic Grammars and Languages. *International Journal of Computer & Information Sciences*, 1, 1972.
- [14] J. W. Gibbs. A Method of Geometrical Representation of the Thermodynamic Properties of Substances by Means of Surfaces. *Transactions of the Connecticut Academy of Arts and Sciences. 2: 382–404*, 1873.
- [15] J. Goodman. *Parsing Inside-Out*. PhD thesis, Harvard University, 1998.
- [16] J. Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–605, 1999.
- [17] J. Gorodkin and W. L. Ruzzo, editors. *RNA Sequence, Structure, and Function: Computational and Bioinformatic Methods*, volume 1097 of *Methods in Molecular Biology*. Humana Press, 2014.
- [18] S. Grumbach and F. Tahi. Compression of DNA sequences. *Data Compression Conference, 1993 DCC’93 1993:340-350*, 1993.
- [19] D. Grune. *Parsing Techniques : A Practical Guide (2nd ed.)*. New York: Springer, 2008.
- [20] R. R. Gutell. *Comparative Analysis of the Higher-Order Structure of RNA*. In: Russell, R. (eds) *Biophysics of RNA Folding. Biophysics for the Life Sciences, vol 3*. . Springer, New York, NY., 2013.

- [21] R. R. Gutell, J. C. Lee, and J. J. Cannone. The Accuracy of Ribosomal RNA Comparative Structure Models. *Current Opinion in Structural Biology* 2002 Jun;12(3):301-10., 2002.
- [22] P. G. Higgs. RNA Secondary Structure: Physical and Computational Aspects. *Quarterly Reviews of Biophysics* 2001, 33(03):199-253, 2001.
- [23] I. L. Hofacker, W. Fontana, P.F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast Folding and Comparison of RNA Secondary Structures. *Chemical Monthly*, 125(2):167–168, 1994.
- [24] P. G. Howard and J. S. Vitter. Arithmetic Coding for Data Compression. *Proceedings of the IEEE*, 82, 1994.
- [25] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. doi:10.1109/jrproc.1952.273898, 1952.
- [26] IEEE. IEEE Standard for Developing Software Life Cycle Processes. in *IEEE Std 1074-1991*, pp.1-112, 29 Jan. 1992, doi: 10.1109/IEEESTD.1992.101080, 1992.
- [27] S. Itiroo. Syntax in universal translation. *International Conference on Machine Translation of Languages and Applied Language Analysis*, page 593–608, 1962.
- [28] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. Highly Accurate Protein Structure Prediction with AlphaFold. *Nature*, 596(7873):583–589, July 2021.
- [29] D. R. Karger. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. In: Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA. *ACM/SIAM*, 1993.
- [30] B. Knudsen and J. Hein. Pfold: RNA Secondary Structure Prediction using Stochastic Context-Free Grammars. *Nucleic Acids Research* 2003 Jul 1;31(13):3423-8. doi: 10.1093/nar/gkg614. PMID: 12824339; PMCID: PMC169020., 2003.

- [31] A. N. Kolmogorov. On Tables of Random Numbers. *Theoretical Computer Science*, 207, 1998.
- [32] Q. Liu, Y. Yang, C. Chen, J. Bu, Y. Zhang, and X. Ye. RNACompress: Grammar-based Compression and Informational Complexity Measurement of RNA Secondary Structure. *BMC Bioinformatics*, 9(1):1–12, 2008.
- [33] B. Ma, J. Tromp, and M. Li. PatternHunter—Faster and More Sensitive Homology Search. *Bioinformatics*, 18:440–445, 2002.
- [34] E. N. Markus and A. Scheid. Evaluation of a Sophisticated SCFG Design for RNA Secondary Structure Prediction. *Theory in Biosciences*, 130(4):313–336, 2011.
- [35] D. H. Mathews, J. Sabina, M. Zuker, and D. H. Turner. Expanded Sequence Dependence of Thermodynamic Parameters Improves Prediction of RNA Secondary Structure. *Journal of Molecular Biology* 288:911–940, 1999.
- [36] D. H. Matthews. How to Benchmark RNA Secondary Structure Prediction Accuracy. *Methods*. 2019 Jun 1;162-163:60-67. doi: 10.1016/j.ymeth.2019.04.003. Epub 2019 Apr 2. PMID: 30951834; PMCID: PMC7202366., 2019.
- [37] M. E. Nebel and A. Scheid. Analysis of the Free Energy in a Stochastic RNA Secondary Structure Model. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(6):1468–1482, 2011.
- [38] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for Loop Matchings. *SIAM Journal on Applied Mathematics*, 35:68–82, 1978.
- [39] E. Onokpasa, S. Wild, and P. W. H. Wong. RNA Secondary Structures: from Ab Initio Prediction to Better Compression, and Back. In *Data Compression Conference (DCC)*, pages 278–287, 2023.
- [40] E. Onokpasa, S. Wild, and P. W. H. Wong. Towards Optimal Grammars for RNA Structures. In *2024 Data Compression Conference (DCC)*, pages 332–341, 2024.
- [41] E. Rivals, J.-P. Delahaye, M. Dauchet, and O. Delgrange. A Guaranteed Compression Scheme for Repetitive DNA Sequences. In *Data Compression Conference, DCC '96 Proceedings*. IEEE, 1996.

- [42] E. Rivas and S. R. Eddy. A Dynamic Programming Algorithm for RNA Structure Prediction including Pseudoknots. *Journal of Molecular Biology, Volume 285, Issue 5*, <https://doi.org/10.1006/jmbi.1998.2436>., 1999.
- [43] M. A. Rosenblad, J. Gorodkin, B. Knudsen, C. Zwieb, and T. Samuelsson. SRPDB: Signal Recognition Particle Database. *Nucleic Acids Research 2003*, 31:363-364., 2003.
- [44] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, and K. Sjolander. Stochastic Context-Free Grammars for tRNA Modeling. *Nucleic Acids Research 1994*, 22:5112-5120., 1994.
- [45] Y. Sakakibara, M. Brown, R. Underwood, I. S. Mian, and D. Haussler. Stochastic Context-free Grammars for Modeling RNA. *International Conference on System Sciences, Wailea, HI, USA, 1994*, pp. 284-293, doi: 10.1109/HICSS.1994.323568., 1994.
- [46] A. Schulz. *Sampling and Approximation in the Context of RNA Secondary Structure Prediction Algorithms and Studies Based on Stochastic Context-Free Modeling*. PhD dissertation, Technische Universität Kaiserslautern, 2012.
- [47] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27, 1948.
- [48] D. H. Turner and D. H. Mathews, editors. *RNA Structure Determination*. Springer New York, 2016.
- [49] J. van der Hoek and R. J. Elliott. *The Viterbi Algorithm*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2018.
- [50] S. Wild. *An Earley-style Parser for Solving the RNA-RNA Interaction Problem*. Bachelor's Thesis, University of Kaiserslautern, 2010.
- [51] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30, 1987.
- [52] T. Xia, J. SantaLucia Jr., M. E. Burkard, R. Kierzek, S. J. Schroeder, X. Jiao, C. Cox, and D. H. Turner. Thermodynamic Parameters for an Expanded Nearest Neighbor Model for Formation of RNA Duplexes with Watson-Crick Base Pairs. *Biochemistry*, 37:14719-14735, 1998.

- 
- [53] D. H. Younger. Recognition and Parsing of Context-free Languages in time  $n^3$ . *Information and Control*, 10(2):189–212, 1967.
- [54] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data compression. *IEEE Transactions on Information Theory*, volume 23, no. 3, pp. 337–343, May 1977, doi: 10.1109/TIT.1977.1055714, 1977.
- [55] M. Zuker and D. Sankoff. RNA Secondary Structures and their Prediction. *Bulletin of Mathematical Biology*, 46:591–621, 1984.

## Appendix

### A Human Expert Grammars

Here, we list the used grammars; we use the compact notation from [8], where we only give  $a$  and  $\hat{a}$  as terminals. The actual RNA grammars would have 4 rules for each rule with a single “ $a$ ”; instead of  $A \rightarrow \alpha a \beta$ , we would actually have  $A \rightarrow \alpha \begin{bmatrix} \text{A} \\ \bullet \end{bmatrix} \beta$ ,  $A \rightarrow \alpha \begin{bmatrix} \text{C} \\ \bullet \end{bmatrix} \beta$ ,  $A \rightarrow \alpha \begin{bmatrix} \text{G} \\ \bullet \end{bmatrix} \beta$ , and  $A \rightarrow \alpha \begin{bmatrix} \text{U} \\ \bullet \end{bmatrix} \beta$ ;

Similarly, each rules with a “ $a\hat{a}$ ” pair for example  $A \rightarrow a\beta\hat{a}$  using only **canonical** rules stands for the following rules:

$$\begin{aligned} A &\rightarrow \begin{bmatrix} \text{A} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{A} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{C} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{G} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{G} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{C} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{G} \\ \text{G} \end{bmatrix} \text{ and} \\ A &\rightarrow \begin{bmatrix} \text{G} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}. \end{aligned}$$

If we allow **non-canonical rules** then it represents the 16 rules below.

$$\begin{aligned} A &\rightarrow \begin{bmatrix} \text{A} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{A} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{A} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{G} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{A} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{A} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{C} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{C} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{C} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{C} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{G} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{C} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{G} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{C} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{G} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{G} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{G} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{A} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{C} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{G} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{C} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{A} \\ \text{G} \end{bmatrix}, \\ A &\rightarrow \begin{bmatrix} \text{A} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{U} \\ \text{G} \end{bmatrix}, & A &\rightarrow \begin{bmatrix} \text{A} \\ \text{C} \end{bmatrix} \beta \begin{bmatrix} \text{G} \\ \text{G} \end{bmatrix} \end{aligned}$$

For the stacking grammars, non-terminals  $B^{a\hat{a}}$  are shorthand notation for 6 non-terminals (if we allow only canonical base pairs) and 16 different non-terminals (if we include non-canonical base pairs), which “remember” an enclosing pair. If there are several occurrences of the same  $a\hat{a}$  pair within one rule, these must be replaced consistently (with the same bases in all occurrences).

Our parsers require grammars to be free of  $\epsilon$ -rules, so we eliminated these in all grammars.

Moreover, the fast stochastic parser used for the prediction study requires a slightly more restrictive form: the grammars are not allowed to have left-recursive rules, and the

nonterminals must be ordered, so that  $B$  comes before  $A$  whenever one can derive  $B\alpha$  from  $A$ . We only use the unambiguous grammars  $G_3, \dots, G_8$  for the prediction study, so we directly give those grammars in the required form.

**Grammar  $G_1$**  (DowellGrammar1Bound)

Next,  $G_1, \dots, G_8$  are the grammars taken from Dowell and Eddy [8].

$$U \rightarrow a$$

$$B \rightarrow aS\hat{a}$$

$$C \rightarrow B \mid U$$

$$X \rightarrow UX \mid SX \mid U \mid S$$

$$S \rightarrow C \mid CX \mid US \mid USX$$

**Grammar  $G_2$**  (DowellGrammar2Bound)

$$U \rightarrow a$$

$$P^{a\hat{a}} \rightarrow aP^{a\hat{a}}\hat{a} \mid S$$

$$S \rightarrow aP^{a\hat{a}}\hat{a} \mid U \mid US \mid SU \mid SS$$

**Grammar  $G_3$**  (DowellGrammar3Bound)

$$U \rightarrow a$$

$$B \rightarrow aS\hat{a}$$

$$L \rightarrow B \mid UL$$

$$R \rightarrow U \mid UR$$

$$S \rightarrow B \mid UL \mid RU \mid LS \mid U$$

**Grammar  $G_4$**  (DowellGrammar4Bound)

$$U \rightarrow a$$

$$B \rightarrow aS\hat{a}$$

$$C \rightarrow B \mid U$$

$$D \rightarrow C \mid CD$$

$$Q \rightarrow B \mid BD$$

$$S \rightarrow U \mid US \mid Q$$

**Grammar  $G_5$**  (DowellGrammar5Bound)

$$\begin{aligned}
U &\rightarrow a \\
B &\rightarrow aS\hat{a} \\
S &\rightarrow U \mid B \mid US \mid BS
\end{aligned}$$

**Grammar  $G_6$**  (DowellGrammar6Bound)

$$\begin{aligned}
U &\rightarrow a \\
B &\rightarrow aM\hat{a} \\
T &\rightarrow B \mid U \\
M &\rightarrow B \mid TS \mid T \\
S &\rightarrow TS \mid T
\end{aligned}$$

An alternative version does not have the rule  $M \rightarrow T$ ; that grammar then disallows hairpins of length one, i.e., ' $(\bullet)$ '.

**Grammar  $G_7$**  (DowellGrammar7Bound)

$$\begin{aligned}
U &\rightarrow a \\
B &\rightarrow aV^{a\hat{a}}\hat{a} \quad (16 \text{ rules}) \\
B^{\hat{b}b} &\rightarrow aV^{a\hat{a}}\hat{a} \quad (16 \cdot 16 \text{ rules}) \\
L &\rightarrow B \mid UL \\
M &\rightarrow UM \mid U \\
T &\rightarrow U \mid UL \mid MU \mid LS \\
V^{a\hat{a}} &\rightarrow B^{a\hat{a}} \mid T \quad (16 \cdot 2 \text{ rules}) \\
S &\rightarrow B \mid UL \mid MU \mid U \mid LS
\end{aligned}$$

**Grammar  $G_8$**  (DowellGrammar8Bound)

$$\begin{aligned}
U &\rightarrow a \\
B &\rightarrow aV^{a\hat{a}}\hat{a} \\
B^{\hat{b}b} &\rightarrow aV^{a\hat{a}}\hat{a} \\
C &\rightarrow U \mid B \\
D &\rightarrow C \mid CD \\
E &\rightarrow B \mid BD \\
N &\rightarrow U \mid E \mid US \mid EU \mid EB
\end{aligned}$$

$$V^{a\hat{a}} \rightarrow B^{a\hat{a}} \mid N$$

$$S \rightarrow U \mid E \mid US$$

### Grammar $G_S$ (SchulzGrammar)

The grammar  $G_S$  is taken from [34]; see also [46, Def. A.1.2]; we have made the modifications described below to make the grammar more suitable for compression.

Since we have to expand every occurrence of  $a \hat{a}$  on the right-hand side into 6 (or even 16) rules in our RNA grammars, we replaced “ $aL\hat{a}$ ” in several right-hand sides with a nonterminal that expands to  $aL\hat{a}$  ( $A$  when we start a new stem and the new nonterminal  $I$  when we continue after an interior loop or bulge). This reduces the number of parameters and hence the expressive power a bit, but will keep the grammar substantially smaller.

$$p'_0 : S' \rightarrow S,$$

$$p'_1 : S \rightarrow A, \quad p'_2 : S \rightarrow AC, \quad p'_3 : S \rightarrow TA, \quad p'_4 : S \rightarrow TAC,$$

$$p'_5 : T \rightarrow A, \quad p'_6 : T \rightarrow AC, \quad p'_7 : T \rightarrow TA, \quad p'_8 : T \rightarrow TAC,$$

$$p'_9 : T \rightarrow C,$$

$$p'_{10} : C \rightarrow X^C, \quad p'_{11} : C \rightarrow CX^C,$$

$$p'_{12} : A \rightarrow aL\hat{a},$$

$$p'_{13} : L \rightarrow aL\hat{a}, \quad p'_{14} : L \rightarrow M, \quad p'_{15} : L \rightarrow P, \quad p'_{16} : L \rightarrow Q,$$

$$p'_{17} : L \rightarrow R, \quad p'_{18} : L \rightarrow F, \quad p'_{19} : L \rightarrow G,$$

$$p'_{20} : G \rightarrow Ia, \quad p'_{21} : G \rightarrow IX^B X^B, \quad p'_{22} : G \rightarrow IBX^B X^B,$$

$$p'_{23} : G \rightarrow aI \quad p'_{24} : G \rightarrow X^B X^B I \quad p'_{25} : G \rightarrow X^B X^B BI$$

$$p'_{26} : B \rightarrow X^B \quad p'_{27} : B \rightarrow BX^B$$

$$p'_{28} : F \rightarrow X^F X^F X^F \quad p'_{29} : F \rightarrow X^F X^F X^F X^F \quad p'_{30} : F \rightarrow X^F X^F X^F X^F X^F$$

$$p'_{31} : F \rightarrow X^F X^F X^F X^F X^F X^F H \quad p'_{32} : H \rightarrow X^H \quad p'_{33} : H \rightarrow HX^H$$

$$p'_{34} : P \rightarrow aIa \quad p'_{35} : P \rightarrow X^I IX^I X^I \quad p'_{36} : P \rightarrow X^I X^I IX^I \quad p'_{37} : P \rightarrow X^I X^I IX^I X^I$$

$$p'_{38} : Q \rightarrow X^I X^I IX^I X^I X^I \quad p'_{39} : Q \rightarrow X^I X^I IKX^I X^I X^I$$
  

$$p'_{40} : Q \rightarrow X^I X^I X^I IX^I X^I \quad p'_{41} : Q \rightarrow X^I X^I X^I JIX^I X^I$$

$$p'_{42} : Q \rightarrow X^I X^I X^I IKX^I X^I \quad p'_{43} : Q \rightarrow X^I X^I X^I JIKX^I X^I$$

$$p'_{44} : R \rightarrow X^I IX^I X^I X^I \quad p'_{45} : R \rightarrow X^I IKX^I X^I X^I \quad p'_{46} : R \rightarrow X^I X^I X^I IX^I,$$

$$p'_{47} : R \rightarrow X^I X^I X^I JIX^I$$

$$p'_{48} : J \rightarrow X^I \quad p'_{49} : J \rightarrow JX^I$$

$$p'_{50} : K \rightarrow X^I \quad p'_{51} : K \rightarrow KX^I$$

$$\begin{aligned}
p'_{52} : M &\rightarrow AA & p'_{53} : M &\rightarrow UAA & p'_{54} : M &\rightarrow AUA & p'_{55} : M &\rightarrow AAN \\
p'_{56} : M &\rightarrow UAU & p'_{57} : M &\rightarrow UAA & p'_{58} : M &\rightarrow AUAN & p'_{59} : M &\rightarrow UAUAN \\
p'_{60} : N &\rightarrow A & p'_{61} : N &\rightarrow UA & p'_{62} : N &\rightarrow AN & p'_{63} : N &\rightarrow UAN \\
p'_{64} : N &\rightarrow U \\
p'_{65} : U &\rightarrow X^U & p'_{65} : U &\rightarrow UX^U
\end{aligned}$$

We add the following rules:

$$F \rightarrow X^F \quad F \rightarrow X^F X^F \quad (\text{allow length 1 and 2 in hairpins})$$

$$I \rightarrow aL\hat{a} \quad (\text{new nonterminal for use inside bulges/interior loops})$$

$$S \rightarrow C \quad (\text{allow completely unpaired sequences})$$

Rules for all unpaired nonterminals:

$$X^B \rightarrow a, \quad X^C \rightarrow a, \quad X^F \rightarrow a, \quad X^H \rightarrow a, \quad X^I \rightarrow a, \quad X^U \rightarrow a$$

## B 9 Best Grammars from Local Search Heuristics

Here we rank the best performing (by compression, using the benchmark-10-percent dataset) grammars using Local Search Heuristics with the best known grammar as input (see Section 6.5.3).

### Second Best Grammar, Compression ratio=2.5099

$$A0 \rightarrow (A3)$$

$$A1 \rightarrow .$$

$$A1 \rightarrow A0$$

$$A2 \rightarrow A1$$

$$A2 \rightarrow A2 \ A1$$

$$A3 \rightarrow A0$$

$$A3 \rightarrow A0 \ A1$$

$$A3 \rightarrow A2$$

start symbol: A3

### Third Best Grammar, Compression ratio=2.5131

$$A0 \rightarrow (A3)$$

$$A1 \rightarrow .$$

$A1 \rightarrow A0$

$A2 \rightarrow A1$

$A2 \rightarrow A2 A1$

$A3 \rightarrow A0$

$A3 \rightarrow A0 A1$

$A3 \rightarrow A2$

start symbol:  $A3$

#### **Fourth Best Grammar, Compression ratio=2.5155**

$A0 \rightarrow (A3)$

$A1 \rightarrow .$

$A1 \rightarrow A0$

$A2 \rightarrow A1$

$A2 \rightarrow A2 A1$

$A3 \rightarrow A0$

$A3 \rightarrow A0 A2$

$A3 \rightarrow A2$

start symbol:  $A3$

#### **Fifth Best Grammar, Compression ratio=2.5159**

$A0 \rightarrow (A3)$

$A1 \rightarrow .$

$A1 \rightarrow A0$

$A2 \rightarrow A1$

$A2 \rightarrow A1 A1$

$A2 \rightarrow A2 A1$

$A3 \rightarrow A0$

$A3 \rightarrow A2$

start symbol:  $A3$

**Sixth Best Grammar, Compression ratio=2.5193** $A_0 \rightarrow (A_3)$  $A_1 \rightarrow .$  $A_1 \rightarrow A_0$  $A_2 \rightarrow A_1$  $A_2 \rightarrow A_0 A_1$  $A_2 \rightarrow A_2 A_1$  $A_3 \rightarrow A_0$  $A_3 \rightarrow A_2$ start symbol:  $A_3$ **Seventh Best Grammar, Compression ratio=2.5197** $A_0 \rightarrow (A_3)$  $A_1 \rightarrow .$  $A_1 \rightarrow A_0$  $A_2 \rightarrow A_1$  $A_2 \rightarrow A_2 A_1$  $A_3 \rightarrow A_0$  $A_3 \rightarrow A_1 A_0$  $A_3 \rightarrow A_2$ start symbol:  $A_3$ **Eighth Best Grammar, Compression ratio=2.5202** $A_0 \rightarrow (A_3)$  $A_0 \rightarrow A_0 A_0$  $A_1 \rightarrow .$  $A_1 \rightarrow A_0$  $A_2 \rightarrow A_1$  $A_2 \rightarrow A_2 A_1$  $A_3 \rightarrow A_0$

$A3 \rightarrow A2$

start symbol: A3

**Ninth Best Grammar, Compression ratio=2.5226**

$A0 \rightarrow (A3)$

$A1 \rightarrow .$

$A1 \rightarrow A0$

$A2 \rightarrow A1$

$A2 \rightarrow A2 A1$

$A3 \rightarrow A0$

$A3 \rightarrow A2$

$A3 \rightarrow A2 A0$

start symbol: A3