Predicting RNA Secondary Structure Using A Stochastic Conjunctive Grammar

by

Ryan Zier-Vogel

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2012

Thesis advisor

Michael Domaratzki

Author

Ryan Zier-Vogel

Predicting RNA Secondary Structure Using A Stochastic Conjunctive Grammar

Abstract

In this thesis I extend a class of grammars called conjunctive grammars to a stochastic form called stochastic conjunctive grammars. This extension allows the grammars to predict pseudoknotted RNA secondary structure. Since observing secondary structure is hard and expensive to do with today's technology, there is a need for computational solutions to this problem. A conjunctive grammar can handle pseudoknotted structure because of the way one sequence is generated by combining multiple parse trees.

I create several grammars that are designed to predict pseudoknotted RNA secondary structure. One grammar is designed to predict all types of pseudoknots and the others are made to only predict a pseudoknot called H-type. These grammars are trained and tested and the results are collected. I am able to obtain a sensitivity of over 75% and a specificity of over 89% on H-type pseudoknots.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to begin by thanking my advisor Dr.Domaratzki for his support, proof reading and comments on this thesis. As well as my wife H.Zanzerl for putting up with me through-out my Masters degree. Finally my parents C.Zier-Vogel and D.Zier-Vogel for always believing in me and supporting me in whenever I do.

Chapter 1

Introduction

Biologists study a macromolecule called ribonucleic acid (RNA), one of whose primary functions is to be a messenger for another macromolecule called deoxyribonucleic acid (DNA). RNA is made up of nucleotides held together by a ribose-phosphate backbone.

However, RNA may have other functions besides messenger RNA, and in some of these cases, the secondary structure is important. Nucleotides in a strand of RNA will bond together to form base pairs. The forming of these base pairs will cause the RNA strand to fold on itself. This folded structure is referred to as secondary structure. An example of RNA secondary structure is seen in Figure 1.1.

Predicting the secondary structure of RNA is an important problem for biologists, because the structure of RNA will affect its function. An important group of structures are called pseudoknots, an example of one which can be seen in Figure 1.2. As an example, these structures have been linked to RNA strands such as those in viruses [Brierley et al., 2007]. Being able to predict pseudoknotted secondary

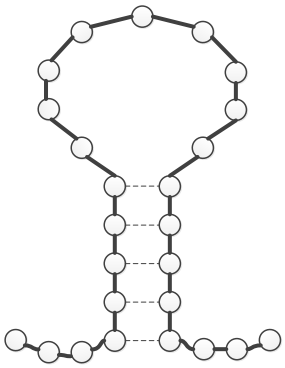


Figure 1.1: This structure is known as a hairpin. The thick black line is the backbone of the RNA and the dotted lines are the bonds in the secondary structure. More information about secondary structure is given in Section 2.1

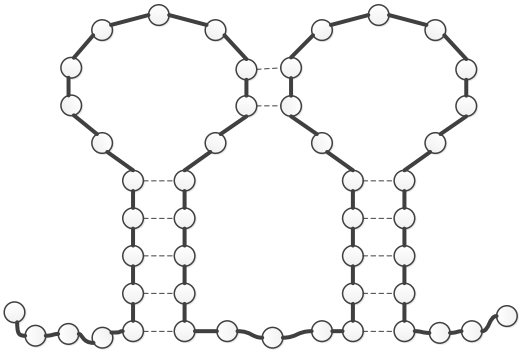


Figure 1.2: This pseudoknotted structure is known as a kissing hairpin; just like in Figure 1.1, the thick black line is the backbone of the RNA and the dotted lines are the bonds in the secondary structure. This structure is pseudoknotted because the two loops at the end of each stem have formed bonds with each other.

structure would allow biologists to have a better understanding of these strands.

From a computational perspective, predicting RNA secondary structure is the fol-

lowing problem: given a sequence of nucleotides (primary structure), find the most likely combination of bonds (secondary structure) for the sequence. Since observing secondary structure is time consuming to do with today's technology, there is a need for computational solutions to this problem. Techniques such as dynamic programming (Rivas and Eddy [1999], Akutsu [2000] and Jabbari et al. [2008]) or machine learning (Sakakibara et al. [1994], Rivas and Eddy [2000] and Fang et al. [2008]) have been applied to attempt to solve this problem.

Predicting RNA secondary structure has been shown to be NP-hard [Akutsu, 2000]. This means that the problem is at least as hard as the hardest NP-complete problem. NP-complete problems have no known polynomial-time algorithms. These problems can however have theoretical solutions in the form of powerful and nondeterministic models of computation.

The prediction of RNA pseudoknotted secondary structure has no known polynomial-time algorithm. However, there is a constrained version of predicting RNA secondary structure that can be solved in polynomial time. By restricting allowable solutions to secondary structures that do not contain pseudoknots, the computational complexity of RNA secondary structure prediction is greatly reduced. This turns a NP-hard form of RNA secondary structure prediction into a form that can be solved with a simple dynamic programming algorithm. An example of a pseudoknotted structure can be seen in Figure 1.2. In the thesis I consider the more general secondary structure prediction problem, in which pseudoknots are permitted.

In this thesis, I will use grammars to predict the pseudoknotted secondary structure of RNA. A grammar is a mathematical model that can be used to assign struc-

ture to a sequence [Linz, 2006]. Grammars work by rewriting special symbols call non-terminals using a set of rules called productions. By assigning probabilities to each production, grammars can to trained to find the most probable structure for a sequence. The probabilities that are assigned will be based on many examples of structure that the grammar is meant to predict, which is why the process is called training. When a grammar has probabilities assigned to its productions the grammar is called stochastic [Durbin et al., 1998]. When a stochastic grammar is developed and trained to solve a problem, the technique is a form of machine learning [Bildi and Brunak, 2001]. Machine learning with grammars is a very powerful tool when applied to prediction problems in bioinfomatics, as seen in Sakakibara et al. [1994], Rivas and Eddy [2000] and Fang et al. [2008] just to mention a few.

A Stochastic Context-Free Grammar (SCFG) is the stochastic form of a Context-Free Grammar (CFG) (CFG, see Section 2.2.1). SCFGs can be used to predict pseudoknot-free RNA secondary structure [Durbin et al., 1998]. However, they are not powerful enough to solve the problem of predicting pseudoknotted RNA secondary structure. This is in part because CFGs are unable to handle cross dependencies which are needed to represent pseudoknots.

CFGs can be expanded to handle cross dependencies. There have been many suggested generalization of CFGs but in my thesis I use Conjunctive Grammars (CG) Okhotin [2001]. These grammars are similar to CFGs, except that they have rules in place that will allow strings to be "anded" together (i.e., intersected). The ability to "and" will make this class of grammars more powerful than CFGs. In this thesis, I define a model called Stochastic Conjunctive Grammars (SCG) which are a

stochastic form of Conjunctive Grammars. I use this powerful model to predict pseudoknotted RNA secondary structure. A SCG can be applied to solve this problem because of the way one sequence is generated by combining multiple parse trees.

In my thesis I use a SCG to predict the pseudoknotted secondary structure of RNA. I design several different grammars to predict the structure. The first grammar is similar to SCFGs that have been used to predict pseudoknot-free secondary structure [Durbin et al., 1998]. Then productions are added to the grammar so it can handle pseudoknots.

I test this grammar using all the RNA pseudoknotted sequences in a database called pseudoBase++ Taufer et al. [2009]. This database is a collection of 304 pseudoknotted RNA sequences with known secondary structures. Testing on all types of pseudoknots was done in the hopes that this grammar could successfully predict pseudoknots.

This grammar did not yield the results that were desired so new grammars were designed. The new grammars that I designed would only predict H-type pseudoknots (see the example in Figure 1.3). This is the most common type of pseudoknot, with 236 out of 304 RNA sequences in pseudoBase++ having only this type of pseudoknot. These new grammars were designed based on statistics about 235 of the 236 H-type pseudoknots (one sequence was excluded). The statistics that were gathered were parts of a hairpin structure. I called these parts prefix, stem, loop and suffix (an example can be seen in Figure 1.4). These new grammars yield much better results than the first grammar to predict RNA pseudoknotted secondary structure. I was able to obtain a sensitivity of over 75% and a specificity of over 89% on H-type

pseudoknots.

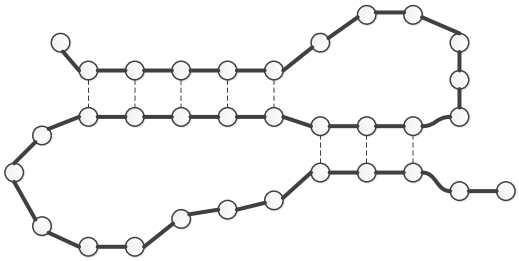


Figure 1.3: This is an example of a H-Type; just like in Figure 1.1, the thick black line is the backbone of the RNA and the dotted lines are the bonds in the secondary structure. This structure is pseudoknotted because the tail after the stem will form bonds with the loop of the hairpin.

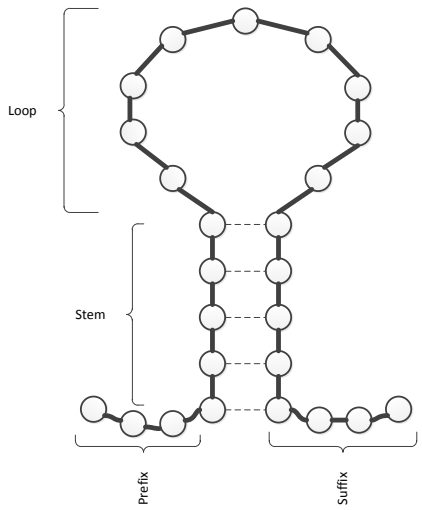


Figure 1.4: This shows how I break down a hairpin into its parts to allow for statistic gathering.

Chapter 2

Background

2.1 Biology

Ribonucleic acid (RNA) is a single-stranded sequence of four different nucleotides, adenine (A), cytosine (C), guanine (G), or uracil (U). RNA can be viewed as a sequence of elements from the alphabet $\{U, G, C, A\}$. The sequence of nucleotides held together by a ribose-phosphate backbone is called the primary structure, an example of which seen in Figure 2.1.

Figure 2.1: The thick black line is the ribose-phosphate backbone, and the circles with the letters in them represent the nucleotides.

The nucleotides in a RNA strand will bond to each other, forming base pairs in the strand of RNA. The bonds that occur will form most commonly in a Watson-

Crick matter. This is named after the two researchers who first observed this bond pattern Watson and Crick [1953]. They made their original discovery on DNA but the principle can be applied to RNA as well. They observed that a purine, which are A and G, will need to bond with a particular pyrimidine, which are U and C. In particular, A will bond to U and G will bond to C. Another bond which is also common is the bond between G and U; this bond it sometime referred to as a "wobble" bond.

For bonds to form, the RNA strand will have to fold on itself. This folded structure is known as the secondary structure (an example of which is seen in Figure 2.2).

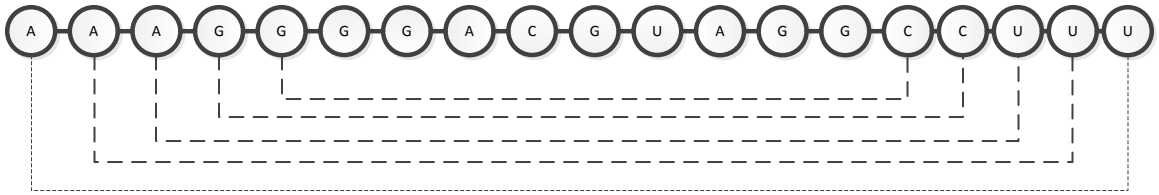


Figure 2.2: The thick black line is the ribose-phosphate backbone, and the circles with the letters in them represent the nucleotides. The thin dotted lines represent the base pairs.

Secondary structure can be either pseudoknot-free, which means it has only nested base pairs, or pseudoknotted, which means it includes non-nested base pairs. Each base pair is represented by a pair (i,j) which represents that nucleotides in positions i and j of the strand have bonded together. Base pairs are nested if, when (i,j) and $(i^{'},j^{'})$ are bonds then

a) $i < i' < j' < j$;

b) $i' < i < j < j'$;

c) $j' < i' < i < j$;

d) or $j < i < i' < j'$.

Thus a pseudoknotted structure contains bonds (i,j) and (i',j') where

a) $i' < i < j' < j$;

b) $i < i' < j < j'$;

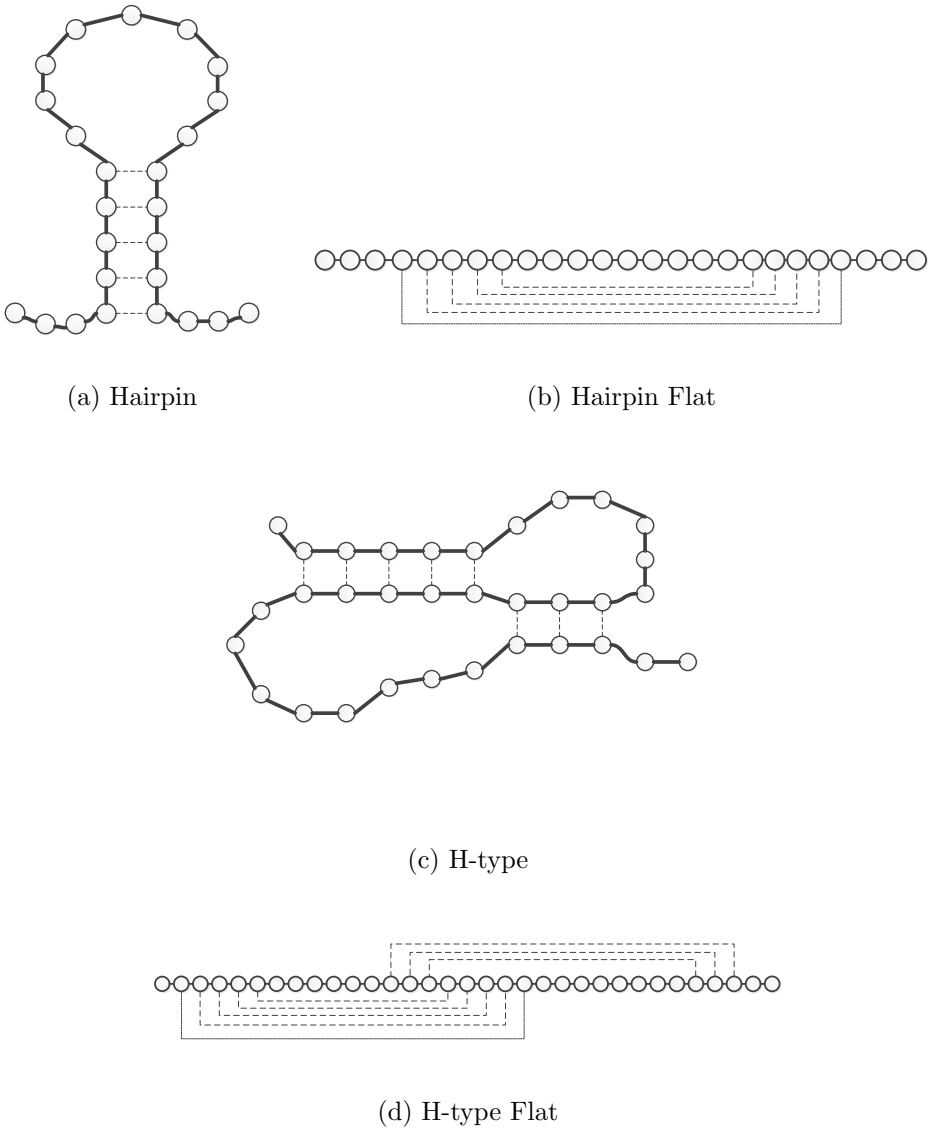
c) $j' < i < i' < j$;

d) or $j < i' < i < j'$.

Secondary structure has been well examined over the years and is nicely defined by Nowakowski and Tinoco, Jr. [1997]. They define the types of secondary structures that can be formed: stems, hairpins, bulges, internal loops, H-type pseudoknots, kissing hairpins and hairpin loop-bulge contacts. These structures can be seen in Figures 2.3 and 2.4.

2.2 Grammars

Grammars are mathematical structures that use productions, non-terminals and terminals to generate a sequence of terminal symbols. Productions are rules that will involve rewriting one non-terminal with a string of non-terminals and terminals. Non-terminal symbols can be rewritten by a production rule. A terminal symbol can



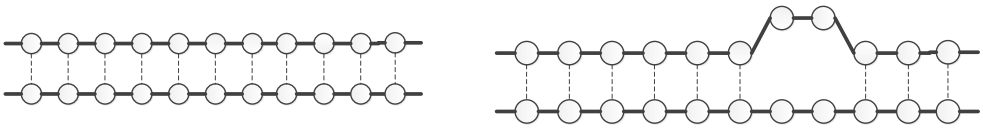
(a) Hairpin

(b) Hairpin Flat

(c) H-type

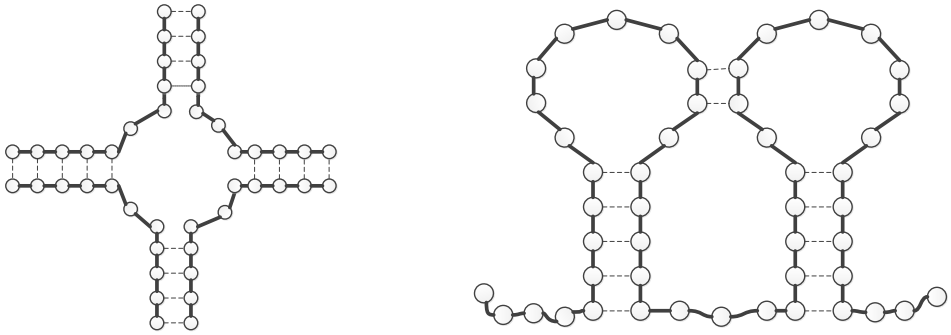
(d) H-type Flat

Figure 2.3: (a) A pseudoknot-free structure known as a hairpin. The thick black line is the backbone of the RNA and the dotted lines are the bonds in the secondary structure. (b) A hairpin structure if the backbone is pulled flat. (c) A pseudoknotted structure known as a H-type pseudoknot. The thick black line is the backbone of the RNA and the dotted lines are the bonds in the secondary structure. (d) A H-type structure if the backbone is pulled flat.



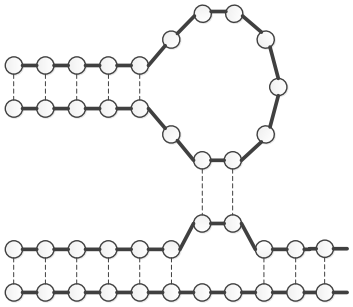
(a) Stem

(b) Bulge



(c) Internal loops

(d) Kissing hairpins



(e) Hairpin loop-bulge contacts

Figure 2.4: (a,b,c) Other pseudoknot-free structures. (d,e) Other pseudoknotted structures.

not be rewritten and will be part of the terminal sequence that is being generated by the grammar. Grammars are used when a sequence has a structure that associates with a sequence.

2.2.1 Context-Free Grammar

A Context-Free Grammar (CFG) G can be described as a four-tuple $G = (\Sigma, N, R, S)$ where

- Σ is the set of terminal symbols,
- N is the set of non-terminal symbols,
- R is the set of all the productions for the grammar. The productions are written as $A \rightarrow \alpha$ where $\alpha \in (\Sigma \cup N)^*$, i.e., α is a string of terminals and non-terminals,
- $S \in N$ is the start symbol.

These productions will have one non-terminal producing a string of non-terminal and terminal elements. These productions will be represented by one non-terminal on the left hand side of an $\rightarrow$ symbol and the string it will produce on the right hand side. If a non-terminal can produce more the one string, each of the strings will be on the right hand side and separated by a | symbol.

These productions will be applied by rewriting a non-terminal element by one of the strings on the right hand side of its production. The rewriting of a non-terminal will be represented by a $\Rightarrow$ symbol. When several of these rewritings are combined together it is known as a derivation. A derivation will start with the start symbol of a grammar and begin by rewriting that symbol. A derivation will continue until there

are no more non-terminals to be rewritten. All possible sequences that a grammar can produce are generated by a derivation starting with the start symbol. An example of a derivation can be seem in Equation 2.1. For Equation 2.1 the four-tuple $G = (\Sigma, N, R, S)$ will be

- $\Sigma = \{a, u, c, g\}$ (These nucleotides have been switched to lower case letters so they will not get confused with the non-terminals which are upper case.)
- $N = \{S\}$
- $R = \{S \rightarrow aSu|uSa|cSg|gSc|uS|aS|cS|gS|a|u|c|g\}$
- $S \in N$ is the start symbol.

The following example uses the grammar G to generate the sequence *aaggagcuu*. This is just one possible derivation for this sequence, and it is easy to see that other derivations would be possible.

$$\begin{aligned}
& S && (\text{use } S \rightarrow aSu) \\
\Rightarrow\ & aSu && (\text{use } S \rightarrow aSu) \\
\Rightarrow\ & aaSuu && (\text{use } S \rightarrow gSc) \\
\Rightarrow\ & aagScuu && (\text{use } S \rightarrow gS) \\
\Rightarrow\ & aagaScuu && (\text{use } S \rightarrow aS) \\
\Rightarrow\ & aaggaScuu && (\text{use } S \rightarrow g) \\
\Rightarrow\ & aaggagcuu
\end{aligned} \tag{2.1}$$

Another example is given in Equation 2.2.

$$
\begin{aligned}
& S && (\text{use } S \to aS) \\
& \Rightarrow aS && (\text{use } S \to aSu) \\
& \Rightarrow aaSu && (\text{use } S \to gS) \\
& \Rightarrow aagSu && (\text{use } S \to Su) \\
& \Rightarrow aagSuu && (\text{use } S \to gSc) \\
& \Rightarrow aaggScuu && (\text{use } S \to aS) \\
& \Rightarrow aaggaScuu && (\text{use } S \to g) \\
& \Rightarrow aaggagcuu
\end{aligned}
\tag{2.2}
$$

A derivation can be represented as seen in Equations 2.1 and 2.2. Another representation is a parse tree; this is a tree structure where non-terminal symbols translate to nodes with children and terminal symbols translate to leaf nodes. An example of a parse tree is is seen in Figure 2.5.

The parse tree in Figure 2.5 shows how a derivation will translate to a parse tree. A better example of how parse trees translate to secondary structure is seen in Figure 2.6. Note how the tree's structure is very similar to the RNA secondary structure. If a line were drawn connecting the leaf nodes, it would be the backbone of the secondary structure. If you then removed all the productions that only produced one terminal, and replaced the productions that produced two terminals with a dotted line between the terminals, these three steps would transform the parse tree into the secondary structure given by the derivation.

By extending a CFG to a SCFG it is able to learn how to solve problems. This is possible because a stochastic grammar can have probabilities assigned to its productions based on other examples. A SCFG will be the same as a CFG but each of the

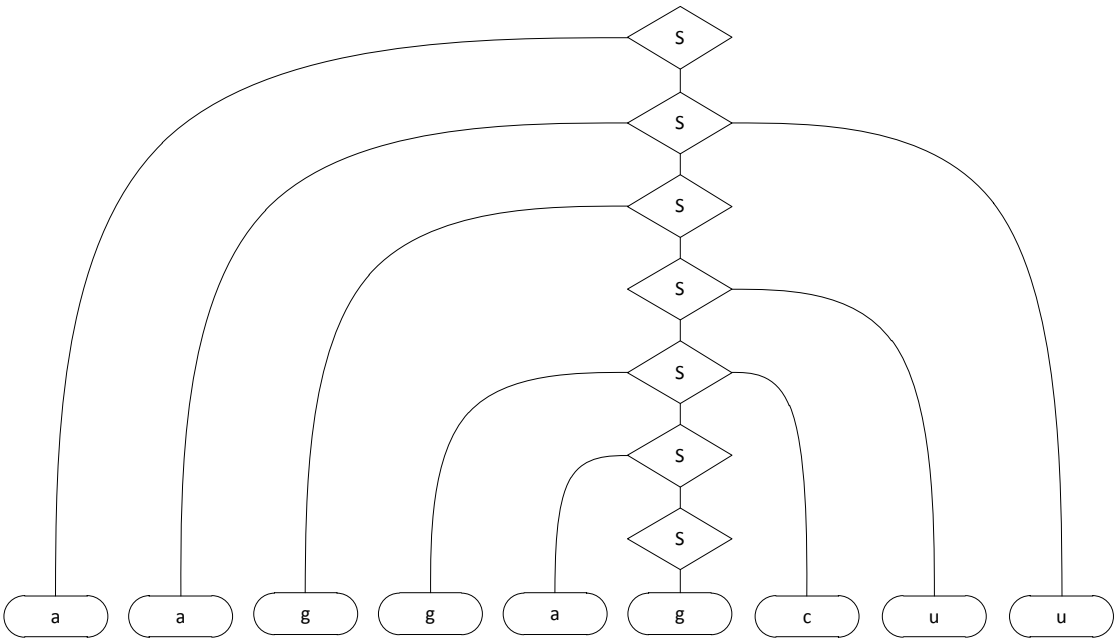
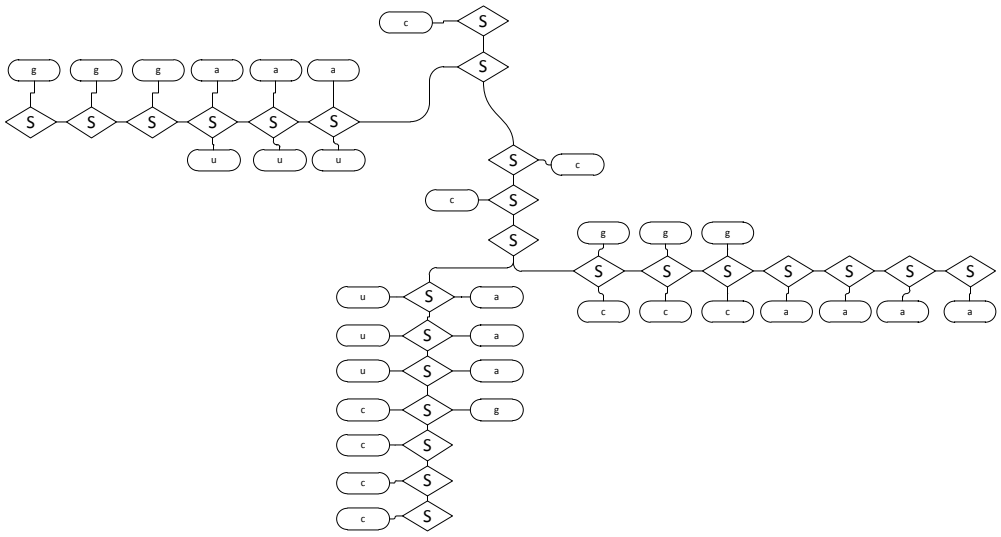


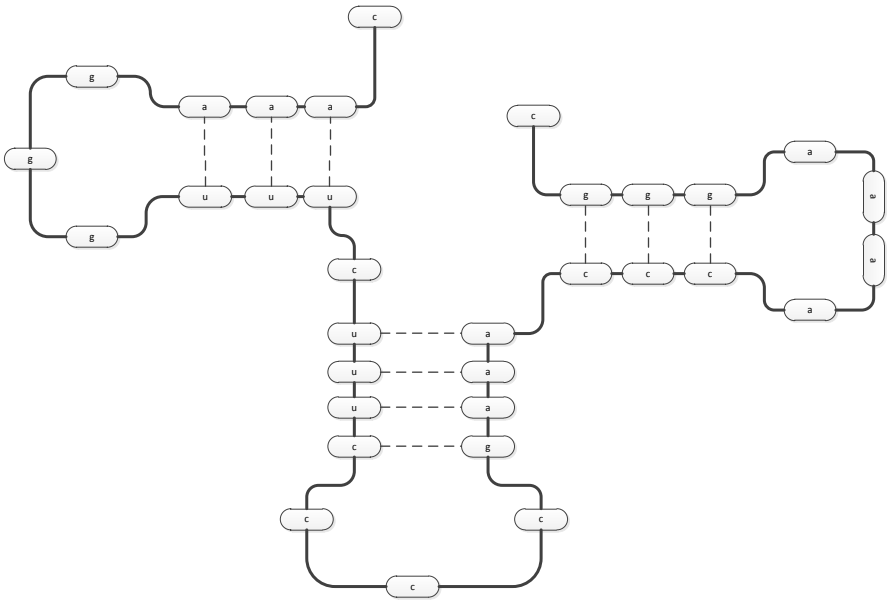
Figure 2.5: In this parse tree the non-terminal symbols are represented by the diamond nodes and the terminal symbols are represented by the oval nodes. This parse tree is a representation of the derivation in Equation 2.2. If the leaf nodes are read from left to right it will be the same as the sequence generated in Equation 2.2.

productions for a SCFG will have a probability associated to it. Also, if probabilities were assigned, Equations 2.1 and 2.2 would likely have different probabilities and one parse would be more likely to form than the other.

The concept of a stochastic grammar will be explained more in the next section. It is briefly mentioned in this section to demonstrate how the converting of a non-stochastic class of grammars to a stochastic class of grammars has been done before.



(a) Parse Tree



(b) Traced Parse Tree

Figure 2.6: (a) A parse tree representation of the secondary structure of RNA sequence. (b) The secondary structure of the same sequence in part (a).

2.2.2 Conjunctive Grammar

There is a more powerful class of grammars known as Conjunctive Grammars (CG). This class of grammars is more powerful because they will have the ability to use cross dependencies which are important to predict pseudoknotted RNA secondary structure. As defined by Okhotin [2001], a CG is a four-tuple $G =(\Sigma, N, P, S)$ where

- Σ is the set of terminal symbols.
- N is the set of non-terminal symbols.
- P is the set of rules for the grammar. The productions are written as $A \rightarrow \alpha_1 \ \& \ \ldots \ \& \ \alpha_n$ where $n \geq 1$ and $\alpha_i \in (\Sigma \cup N)^*$. Each α_i is called a conjunct in the production (if $X \rightarrow \alpha$ is a production with $\alpha = \alpha_1 \& \alpha_2 \& \ldots \& \alpha_n$ then we use the shorthand $\alpha_i \in \alpha$ to mean that α_i is a conjunct in α).
- $S \in N$ is the start symbol.

The & symbol denotes conjunction or intersection, which means that the parsing of each α_i must have one word in common. In CGs, $A \rightarrow \alpha_1 \& \ldots \& \alpha_n$ means A is rewritten by $(\alpha_1 \& \ldots \& \alpha_n)$ and that all deviations by α_i must lead to the same terminal word or the entire deviation is unsuccessful. If at any point two conjuncts being "anded" together are not the same sequence of terminal symbols then the whole derivation is invalid.

In the same way as a CFG was extended to a SCFG, I have extended a CG to an SCG. I now introduce one of the main contribution of this thesis: a Stochastic Conjunctive Grammar G is a five-tuple $G = (\Sigma, N, P, \Phi, S)$ where,

- Σ is the set of terminal symbols,
- N is the set of non-terminal symbols,
- P is the set of productions for the grammar. The productions are written as $A \to \alpha_1 \ \& \ \ldots \ \& \ \alpha_n$ where $n \geq 1$ and $\alpha_i \in (\Sigma \cup N)^*$,
- $\Phi : P \to [0, 1]$. Φ associates a probability to each production. For each $A \in N$, let $P_A \subseteq P$ be the set of all productions with A on the left-hand side; then we require that Φ satisfies $\sum_{r \in P_A} \Phi(r) = 1$ for all $A \in N$,
- $S \in N$ is the start symbol.

Just like a CG, SCGs have a & symbol; this symbol will allow the grammar to "and" conjuncts together. Assigning probabilities is important when dealing with machine learning because a string can have many parse trees to derive it. It is possible for these parse trees to have very different structure so when probabilities are applied it is possible to see which parse tree is the most likely.

Below is an example of a SCG. On the left are the non-terminals and on the right is the string they can generate, followed by their probabilities in brackets.

$$
\begin{aligned}
&S \to A\&BC \ (p = 1)\\
&A \to cA \ (p = .45)\\
&A \to bA \ (p = .45)\\
&A \to \varepsilon \ (p = .1)\\
&B \to bB \ (p = .75)\\
&B \to \varepsilon \ (p = .25)\\
&C \to cB \ (p = .75)\\
&C \to \varepsilon \ (p = .25)
\end{aligned}
\tag{2.3}
$$

I will introduce some necessary definitions around the concept of derivations. A sentential form is a string over $(\Sigma \cup N \cup \&)^*$. Just like with CFGs, the $\Rightarrow$ symbol indicates that one sentential form can derive another sentential form by a production in P. The symbol $\Rightarrow^*$ will represent that by starting with the symbol on the left hand side and preforming a derivation, the sequence on the right hand side will be generated.

The examples below will attempt to derive the word *bbbcc* and will use a non-stochastic form of the grammar that is given above (this means it is given without probabilities). The four-tuple $G = (\Sigma, N, R, S)$ is

- $\Sigma = \{a, u, c, g\}$
- $N = \{S, A, B, C\}$
- $R =$ Productions in Equation 2.3
- $S \in N$ is the start symbol.

$$
\begin{aligned}
& S \\
& \Rightarrow A\&BC \\
& \Rightarrow bA\&BC \\
& \Rightarrow bbA\&BC \\
& \Rightarrow bbbA\&BC \\
& \Rightarrow bbbcA\&BC \\
& \Rightarrow bbbccA\&BC \\
& \Rightarrow bbbcc\&BC \\
& \Rightarrow bbbcc\&bBC \\
& \Rightarrow bbbcc\&bbBC \\
& \Rightarrow bbbcc\&bbbBC \\
& \Rightarrow bbbcc\&bbbC \\
& \Rightarrow bbbcc\&bbbcC \\
& \Rightarrow bbbcc\&bbbccC \\
& \Rightarrow bbbcc\&bbbcc \\
& \Rightarrow bbbcc
\end{aligned}
\tag{2.4}
$$

The above derivation is successful because both conjuncts are *bbbcc* in the second last step. It is also possible to have an invalid derivation as well; consider the following

derivation.

$$
\begin{aligned}
& S \\
\Rightarrow\ & A\&BC \\
\Rightarrow\ & bA\&BC \\
\Rightarrow\ & bbA\&BC \\
\Rightarrow\ & bbbA\&BC \\
\Rightarrow\ & bbbcA\&BC \\
\Rightarrow\ & bbbccA\&BC \\
\Rightarrow\ & bbbcc\&BC \\
\Rightarrow\ & bbbcc\&bBC \\
\Rightarrow\ & bbbcc\&bbBC \\
\Rightarrow\ & bbbcc\&bbbBC \\
\Rightarrow\ & bbbcc\&bbbC \\
\Rightarrow\ & bbbcc\&bbbcC \\
\Rightarrow\ & bbbcc\&bbbc \\
\Rightarrow\ & \text{invalid}
\end{aligned}
\tag{2.5}
$$

The above derivation is unsuccessful because, in the second last step, one conjunct is *bbbcc* and the other is *bbbc*, which are not equal.

The following derivation is the same as the Equation 2.4 but with probabilities being applied. It will have the following five-tuple:

- $\Sigma = \{a, u, c, g\}$
- $N = \{S, A, B, C\}$
- $R =$ Productions in Equation 2.3

- Φ = Probabilities for the productions in Equation 2.3
- $S \in N$ is the start symbol.

$$
\begin{aligned}
& S \\
& \Rightarrow A\&BC \ (p = 1) \\
& \Rightarrow bA\&BC \ (p = .45) \\
& \Rightarrow bbA\&BC \ (p = .2025) \\
& \Rightarrow bbbA\&BC \ (p = .091125) \\
& \Rightarrow bbbcA\&BC \ (p = .04100625) \\
& \Rightarrow bbbccA\&BC \ (p = .01845281) \\
& \Rightarrow bbbcc\&BC \ (p = .00184528) \\
& \Rightarrow bbbcc\&bBC \ (p = .00138396) \\
& \Rightarrow bbbcc\&bbBC \ (p = .00103797) \\
& \Rightarrow bbbcc\&bbbBC \ (p = .00077848) \\
& \Rightarrow bbbcc\&bbbC \ (p = .00019462) \\
& \Rightarrow bbbcc\&bbbcC \ (p = .00014596) \\
& \Rightarrow bbbcc\&bbbccC \ (p = .00010947) \\
& \Rightarrow bbbcc\&bbbcc \ (p = .00002737) \\
& \Rightarrow bbbcc \ (p = .00002737)
\end{aligned}
\tag{2.6}
$$

The above derivation is the same as Equation 2.4 but with probabilities being applied. Whenever a non-terminal is rewritten, the derivation's current probability is multiplied by the probability of the new production being written in. Since a probability is assigned to the whole production, when simplifying the & symbol the

probability is not affected (i.e., when going from the second last to last step the probability remains at .00002737). The final sequence is *bbbcc* and the probability of the generating this sequence is .00002737.

Probabilities that are used for SCGs will get very small. Multiplying them over and over again will create smaller and smaller numbers. This will cause problems of underflow; to deal with the issue of multiplying probabilities, log odds will be used. This means that the logarithm of the probabilities will be used and they will be summed instead of being multiplied. Log odds are negative numbers, where a lower value corresponds to a smaller probability.

Parse trees can be used to represent CG or SCG derivations (an example of a CG parse tree is seen in Figure 2.7). The only difference between a stochastic parse tree and a non-stochastic parse tree is that there would be a probability assigned to the tree once the parse is finished. For a CG parse tree there are some additional rules that are added so that the parse trees can handle more than one conjunct. It is important when there is more then one conjunct that the leaf nodes are used in the same order for each conjunct. If the leaf nodes are not used in the same order it would be a representation of an invalid derivation.

An example of a CG parse tree is seen in Figure 2.8. It is harder to see the link between parse tree and secondary structure than with Figure 2.6. Note how the top tree only generates stems on the round brackets and the bottom tree generates the square brackets.

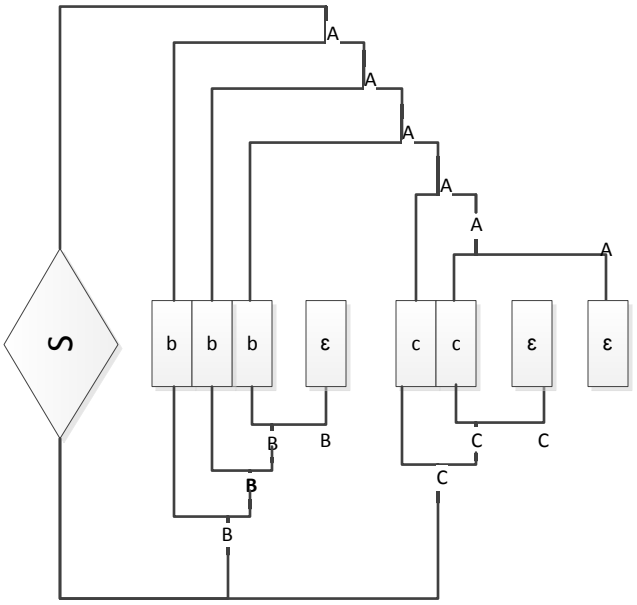


Figure 2.7: This parse tree is a representation of the derivation in Equation 2.4. The children of the diamond shaped node will each represent a different conjunct.

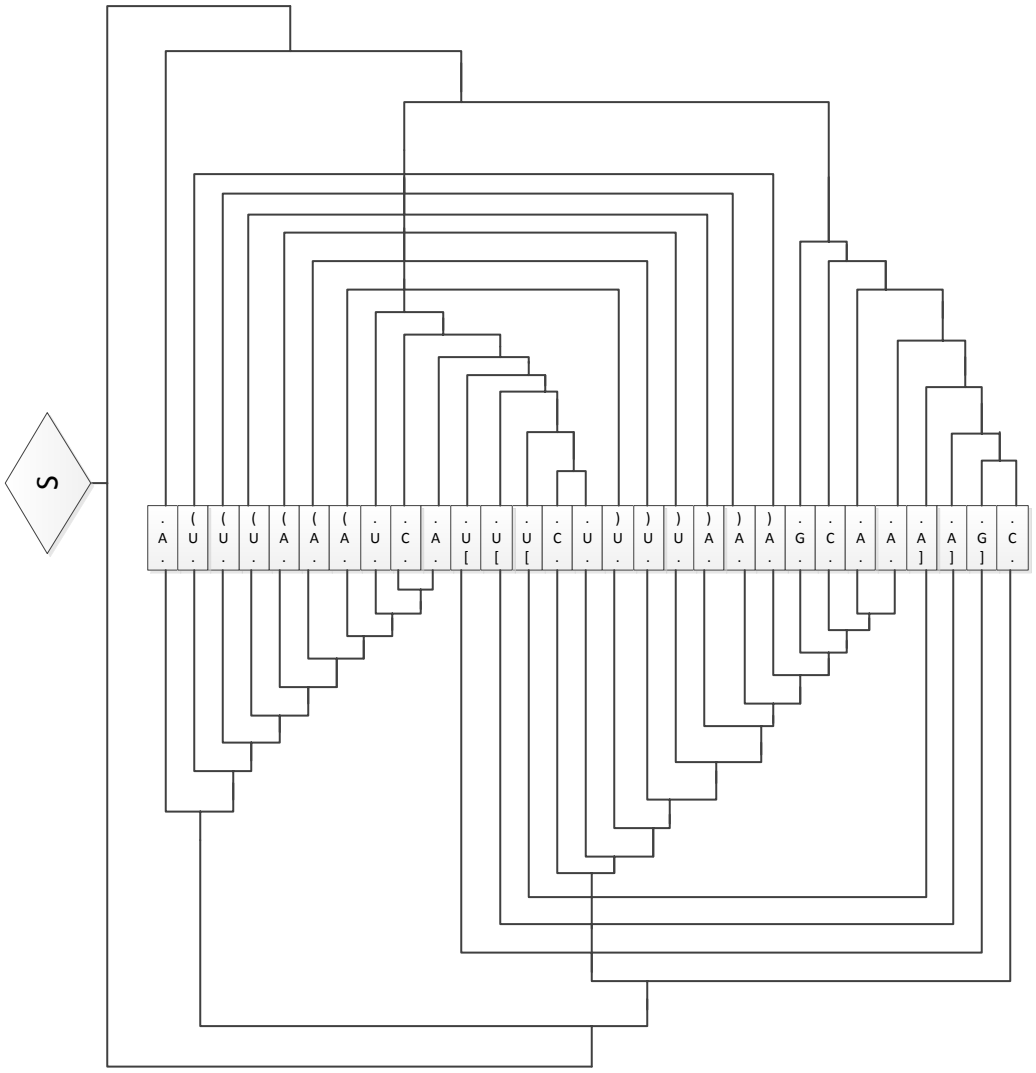


Figure 2.8: This parse tree will start at S. The tree above the sequence will generate one structure and the tree below will generate another structure. When both of these structures are combined it will generate an H-type pseudoknot. Since this is pseudoknotted it will have non-nested brackets; these non-nested brackets will be represented by different bracket types, either a round bracket or a square bracket. Note how the two trees do not generate the stems in the same spots. I have removed labels of the internal nodes for clarity.

Chapter 3

Related Work

Applying stochastic grammars to solve the problem of predicting RNA secondary structure has been used for many years. An early example of the use of a stochastic grammar for solving the problem of predicting RNA secondary structure is an algorithm that was designed by Sakakibara et al. [1994]. Their algorithm was developed to address the shortcomings that Hidden Markov Models (HMM) had when trying to predict the secondary structure of RNA.

HMMs are computational models that have a starting state and several other states. For a more complete introduction to HMMs see the text by Durbin et al. [1998]. Each state in an HMM will have a set of emissions that it can produce. The chance of transiting from one state to another will be governed by values called the transition probabilities. What emission each state will make from the alphabet is governed by a separate emission probability. HMMs do not lend themselves to RNA secondary structure prediction because when predicting stems in RNA secondary structure, the nucleotides in both portions of the strand need to be able to bond.

HMMs work in a very sequential way, which means they are unable to predict these bonds.

RNA secondary structure prediction is a problem for HMMs but it is easily implemented when using a grammar to solve the problem. This is because grammars are able to have productions such as $A \rightarrow aBu$, which is a good representation of a single base pair. The algorithm that was designed by Sakakibara et al. [1994] used a SCFG to predict the secondary structure of RNA. They focused on only predicting transfer RNA (tRNA) because of its simple and well known structure. tRNA will form a cloverleaf pattern with three hairpins that are joined together. Their grammar had 75 non-terminals and 660 productions and it was trained with folded RNA sequences. The grammar was trained using the CYK algorithm which is well-described in Durbin et al. [1998]. The CYK algorithm is a common algorithm to use when training with labelled training data.

The algorithm of Sakakibara et al. [1994] is unable to predict pseudoknotted secondary structure because a context-free grammar can only represent nested structures and pseudoknots are non-nested. However this algorithm does successfully predict pseudoknot-free secondary structure of tRNA 93% of the time.

Another model that was developed at the same time was a covariance model by Eddy and Durbin [1994]. This model will use phylogenetic information in the form of multiple sequence alignments to aid in the prediction of RNA secondary structure. This model is closer to an HMM than it is to a SCFG. It is able to make predictions based on pairwise information. This is due to a complex architecture of this model which incorporates base pairs and allows the model to bifurcate. This will

allow the model to build a tree-like structure.

Just like with the model built by Sakakibara et al. [1994], the covariance model of Eddy and Durbin [1994] also predicted RNA secondary structure of tRNA. This model was able to predict the tRNA secondary structure with an accuracy of over 90%.

Other algorithms have been developed to improve the pseudoknot-free secondary structure predicting power of SCFGs. One example of this kind of algorithm is a profile-SCFG that was developed by Fang et al. [2008]. This model not only used a SCFG but also used two HMMs to aid in predicting of secondary structure. It used the HMMs so that it could incorporate phylogenetic information to make a profile.

A profile is a structure that represents the probability of a certain nucleotide being in a certain location in a strand of RNA. A HMM will build a profile by comparing several strands of RNA that are phylogenetically related, and build its emission probabilities based on the multiple strands. It will look at all the nucleotides in position i of a multiple sequence alignment for each strand and then those counts will be used for the emission probabilities for state i in the profile. Transition probabilities are computed based on the gaps in the multiple sequence alignment.

The algorithm designed by Fang et al. [2008] was able not only to make predictions based on the input RNA sequence but it can also make predictions based on phylogenetic information. This will allow their algorithm to make better predictions overall.

Although neither Sakakibara et al.'s nor Fang et al.'s algorithms were able to predict pseudoknotted secondary structure, there have been many algorithms that

have been developed to do so. Both dynamic programming and stochastic models have been applied to this problem. However, Sakakibara [2005] defended the idea that there are significant advantages to using grammatical inference when solving biological problems. These include the grammar's ability to provide a framework to solve a sequence analysis problem, and then being trained so they can learn details about the problem. This will make for a powerful tool for solving biological problems.

A SCFG is not powerful enough to predict RNA pseudoknotted secondary structure so researchers have made many models that will extend a SCFG to make it suitable to solve this problem. Rivas and Eddy [2000] developed a class of grammars that was able to parse a tree that could represent RNA secondary structure that includes pseudoknots. The class of grammars was called crossed-interaction grammars, because it had extra non-terminals that would allow for interactions in the grammars. These grammars were able to represent the structure but were not built to be stochastic like SCGs, proposed in this thesis, so they were not able to solve the problem of predicting RNA secondary structure. Since Rivas and Eddy [2000] did not make their grammars stochastic they do not have any experimental results to compare against my results.

Another example of using grammars to predict pseudoknotted structures is a model that was developed by Cai et al. [2003]. Their solution to this problem was to use a model called parallel communication grammar systems (PCGS). PCGSs are able to run several SCFGs at once in parallel. Communication between the parallel SCFG allows them to build an RNA secondary structure that can include pseudoknots. These grammars can communicate because there is a special non-terminal symbol

called a query symbol. This will allow multiple grammars to all work together on the same sequence. Their test used 85 sequences to test their results; 42 were used for training and 43 were intended for testing. For various reasons (5 pairs of sequences had the identical pseudoknot and 2 sequences did not have to right type of pseudoknot) their testing size was reduced to 36. They claimed that their model was able to predict pseudoknots with a 69% accuracy.

A more advanced profile-HMM method, developed by Yoon and Vaidyanathan [2008], is a profile-csHMM. This method constructs a profile for a strand of RNA but does so in a way that considers that nucleotides must bond in a Watson-Crick manner. Two new states are created in the profile-csHMM, one that can push onto a stack to affect the probability of future states, and one that can pop off a stack, so its probability can be affected by past states. These new states allow this model to assign probability to stems more accurately than a profile-HMM. This will allow it to make a profile that is more based on structure and less on the sequence alone.

The result of the Yoon and Vaidyanathan [2008] model was very successful, having accuracy for some groups of RNA sequences of up to 97%. This model has the shortcoming of only being able to predict pseudoknotted secondary structure on RNA after building a profile on a group of RNA that is phylogenetically related. On the other hand, the model that is used in this thesis will be able to predict pseudoknotted secondary structure of all RNA or for a specific pseudoknot type. This will make my model much more versatile than the model used in Yoon and Vaidyanathan [2008].

Another class of grammars that has the ability to predict pseudoknotted secondary structure is a tree adjoining grammar (TAG). This model works by building up a

parse tree by adding on more trees to a start tree. This class of grammars was generalized by Uemura et al. [1999] to construct two classes of grammars called simple linear tree adjoining grammars and extended linear tree adjoining grammars. These classes of grammars are not stochastic so instead of having probabilities assigned based on training, they make predictions based on minimal free energy. Free energy is a measurement that is used to test the stability of a folded RNA strand. The algorithms that parse these classes of grammars will run in $O(n^4)$ time for simple linear tree adjoining grammars and $O(n^5)$ time for extended simple linear tree adjoining grammars. No actual experimental results were given in Uemura et al. [1999] but they did state that there were no large mismatches in the predictions by their grammars. A faster parsing algorithm for extended linear tree adjoining grammars was designed by Rajasekaran et al. [2010], reducing the run time to $O(n^4)$.

Using phylogenetically related sequences in order to predict secondary structure is also seen in the model designed by Matsui et al. [2005]. They used a model called pair stochastic tree adjoining grammars. Similar to TAGs, this model works by building up a parse tree by adding on trees to a start tree. To get the model to predict secondary structure, they gave their algorithm a folded sequence that is phylogenetically related to other sequences they wish to predict and use it to build a tree they call a skeletal tree. Their model was able to achieve a specificity of 88.9% and a sensitivity of 96%. They achieve these results with an algorithm that has a running time of $O(n^5)$. Just like the model built by Yoon and Vaidyanathan [2008], their model can only make predictions if they have folded sequences that are phylogenetically related. Another TAG-based model that uses evolutionary information and yields successful predictions

is given by Kato et al. [2006b].

There are also many dynamic programming methods to solve the problem of predicting RNA secondary structure. One of the best known is an algorithm called pknots which was developed by Rivas and Eddy [1999]. It is a extension of an algorithm that is used to find pseudoknot-free secondary structure. This algorithm will use free energy rather than a machine learning technique. This algorithm finds the optimal global pseudoknotted secondary structure of a strand of RNA. Because of the complexity of this problem, this algorithm will run in $O(n^6)$ time and $O(n^4)$ space where n is the length of the sequence.

Other dynamic program algorithms have been designed to reduce the running time for predicting RNA secondary structure with pseudoknots. An example of this type of algorithm is one designed by Akutsu [2000] which runs in $O(n^4)$ time for predicting what they called simple pseudoknots and $O(n^5)$ for complex pseudoknots. Another algorithm was designed by Jabbari et al. [2008] who, by using a new concept called hierarchical folding and only predicting a restricted number of pseudoknot types, designed an algorithm with a running time of $O(n^3)$. Dynamic programming algorithms will all make predictions based on minimal free energy or other measurement methods. This means they are all incapable of learning how to make predictions based on real world examples. The SCG class of grammars that I designed will be able to learn how to predict secondary structure and will have a prediction time complexity of $O(n^3m)$ and space complexity of $O(n^2m)$ where n is the length of the sequence to be predicted and m is the number of non-terminals in the grammar.

Chapter 4

Methods

4.1 Training

For a stochastic grammar to be useful it needs to be trained, which is the process of assigning probabilities to productions. Training a grammar will allow it to learn how it should produce parse trees for new input sequences. Although training will vary between grammars there will be some common elements.

All the training that was done for the SCGs used in this thesis was on labelled training data. An example of labelled training data can be seen in Figure 4.1.

Since all the training data is labelled, it will make the training of the grammar much easier. These training algorithms will all start with a grammar which has no probabilities associated with productions. The productions are hand crafted by me to reflect RNA secondary structure. When the algorithm is done the grammar will hopefully capture the RNA pseudoknotted secondary structure. The training algorithm will go through sequences and count occurrences. How and what will be

```
UAGGGGGGUCAGGGUCAGGAGCCCCCCCCCUGAACCCAGGAUAACCCUCACUGUCGGGGGGCA
::::::::::::(((((((((:[[[[[[[[))))):))))::::::::::::::::::]]]]]]]]:
```

Figure 4.1: Sample of labelled training data, from the website PseudoBase++ by Taufer et al. [2009], sequence id PKB78. The top sequence is the RNA strand and the sequence at the bottom represents its structure. A pair of brackets represents a base pair in the structure. Different types of brackets are needed to denote non-nested structures. The colons represent unpaired nucleotides.

counted will change depending on the grammar and how it will be intended to be trained. Once the counting algorithm is performed the counts can be used to assign probabilities to the productions in the grammar. Once the probabilities are applied the grammar is ready to be used for parsing sequences.

The algorithms for training will take a sequence and its structure and split it into two parts (an example is seen in Figure 4.2). The importance of splitting the sequence and structure in two is because the grammars will predict the square and round brackets independently. This will allow the grammars to predict pseudoknots. The only non-terminal that may need to consider both sets of brackets is the start symbol S. This is because S is the only production that will have an 'and' symbol in it in my grammars.

The simplest form of these training algorithms is to just count the number of base pairs and number of unpaired nucleotides. When the algorithm is run on the sequence seen in Figure 4.2, it will generate the statistics in Table 4.1. A sample grammar to illustrate the concept is seen in Equation 4.1; this will be used to demonstrate how probabilities will be assigned.

```
UAGGGGGGUCAGGGUCAGGAGCCCCCCCCCUGAACCCAGGAUAACCCUCACUGUCGGGGGGCA
::::::::::((((((((:::::::::))))):))))::::::::::::::::::::::::::

UAGGGGGGUCAGGGUCAGGAGCCCCCCCCCUGAACCCAGGAUAACCCUCACUGUCGGGGGGCA
::::::::::::::::::::[[[[[[[:::::::::::::::::::::::::::]]]]]]]:
```

Figure 4.2: This is the RNA sequence in Figure 4.1 split into two. The top two lines are the sequence and structure only for the round brackets and the other uses the square brackets.

$$
\begin{aligned}
&S \rightarrow A\&XKX \\
&A \rightarrow AB|a|u|c|g|aCu|uCa|gCc|cCg|gCu|uCg \\
&B \rightarrow a|u|c|g|aCu|uCa|gCc|cCg|gCu|uCg \\
&C \rightarrow AB|aCu|uCa|gCc|cCg|gCu|uCg \\
&X \rightarrow a|u|c|g|aX|uX|gX|cX \\
&K \rightarrow aKu|uKa|gKc|cKg|gKu|uKg \\
&K \rightarrow aK|uK|gK|cK|Ku|Ka|Kc|Kg \\
&K \rightarrow aX|uX|gX|cX
\end{aligned}
\tag{4.1}
$$

An example of the counts being applied to assign probabilities to productions is seen in Equation 4.2. The non-terminal B is used to predict structure for round brackets, so those counts are used. Note that the counts for the base pairs are doubled; this is because there are two nucleotides involved. The length of the sequence is 62 and that is why it is used when calculating the probability.

	Round	Square
A	10	11
C	14	14
G	16	15
U	6	8
AU	1	0
UA	1	0
GC	5	1
CG	1	6
GU	0	0
UG	0	0

Table 4.1: This table shows the count of all structure elements in the sequence used in Figure 4.2.

$$
\begin{aligned}
B &\rightarrow a\ (10/62) \mid u\ (6/62) \mid c\ (14/62) \mid g\ (16/62) \\
B &\rightarrow aCu\ (2/62) \mid uCa\ (2/62) \mid gCc\ (10/62) \\
B &\rightarrow\ cCg\ (2/62) \mid gCu\ (0/62) \mid uCg\ (0/62)
\end{aligned}
\tag{4.2}
$$

The non-terminal B was chosen as an example in Equation 4.2 because the probabilities are straightforward. Other non-terminals will have productions that are harder to assign probabilities when just counting unpaired and paired nucleotides. One example of this is the production $A \rightarrow AB$ because this is the probability of continuing on with the sequence of unpaired nucleotides. It is a probability that would

be pre-assigned which will take away the advantage of training a grammar.

Since the first example of a technique for training may be overly simple, it tends to assign less than optimal probabilities to productions. Another technique that can be used is one that will process a sequence and its structure, then create a count of the productions that are needed to generate that sequence with that structure. Then the algorithm will take the counts that have been collected and divide it by the total number of times a production for their non-terminal was used. An example of pseudocode that would gather these counts can be seen in the algorithm in Figure 4.3. Note that this algorithm will have to be slightly different depending on the grammar it will be training for (i.e., the algorithm in Figure 4.3 will only work for the grammar in Equation 4.1). The algorithm will have a running time of $O(L)$ where L is the size of the training data.

	Total	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
S	1	1																	
A	48	44	1	0	1	0	0	0	1	1	0	0							
B	45	9	6	14	16	0	0	0	0	0	0								
C	7	1	1	1	0	4	0	0											
X	47	2	0	1	0	10	8	15	11										
K	8	0	0	1	6	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Table 4.2: This is a table which will be used to represent the grammar (Equation 4.1) for the gatherCounts algorithm. Each row will represent a non-terminal (i.e. row 1 is the non-terminal S, A is row 2 and so on). The first column will be the total count of how many times the non-terminal is used and the following columns will represent the productions in the order they appear in Equation 4.1. The values that are filled out are based on the sequence in Figure 4.2. Not all rows will have the same number of columns because not all non-terminals have the same number of productions.

```
// rBracket = input round bracket structure
// sBracket = input square bracket structure
// Seq = input RNA sequence
gatherCounts(String rBracket, String sBracket, String Seq) {
1    Stack S
2    c // used as a temp variable
3    L = length of Seq
4    G = Table 4.2
5    for i = 1 to L {
6        if sBracket[i] == ":" {
7            if continuing a sequence of unpaired nucleotides
8                increase G[1][1] and G[2][index of Seq[i]]
9            if not continuing a sequence of unpaired nucleotides
10               increase G[1][index of Seq[i]]
11       }
12       else if sBracket[i] == "[" {
13           S.push(Seq[i])
14       }
15       else if sBracket[i] == "]" {
16           c = S.pop
17           if first base pair in a stem
18               increase G[2][index for base pair c and Seq[i]]
19           if continuing a stem
20               increase G[3][index for base pair c and Seq[i]
21       }
22   }
23   for i = 1 to L {
24       if rBracket[i] == ":" {
25           if continuing a sequence of unpaired nucleotides
26               increase G[4][index of Seq[i]+4]
27           if not continuing a sequence of unpaired nucleotides
28       }
29               increase G[4][index of Seq[i]]
30           if it is part of a bulge
31               increase G[5][index of Seq[i] of a bulge]
32       else if rBracket[i] == "(" {
33           push Seq[i] on to S
34       }
35       else if rBracket[i] == ")" {
36           c = S.pop
37           increase G[5][index for base pair c and Seq[i]]
38       }
39   }
40 }
```

Figure 4.3: gatherCount Algorithm.

Often when training grammars there are structural elements that might be missing in the training data, leading to suboptimal predictions. To avoid this training will often involve pseudocounts [Durbin et al., 1998]. This means an extra count will be added to all productions even if they are not seen in the training data; this extra count is called a pseudocount. All grammars I used in this thesis had pseudocounts applied to them. I used a pseudocount of one for all productions in my grammars.

4.2 Parsing

It is important to be able to determine whether or not a sequence can be generated by a grammar. This can be done by working from the start symbol, applying productions until the sequence is generated (an example of this process as a parse tree is seen in Figure 4.4) and if so, determining what is the most likely parse tree for that sequence. It is not only important to know if a grammar can generate a sequence; with a stochastic grammar, the most probable parse tree that generates it is also useful. All parse trees that generate the sequence will be found, and then the tree with the highest probability is generated. Since the grammars that I will be using will have productions in the form on $A \rightarrow bA$ where is $b \in \{a,\ c,\ g,\ u\}$, the grammars will be able to generate every sequence so the most likely parse tree becomes important.

These processes can become very hard to do by hand when the grammars become more complex. The Cocke-Younger-Kasami (CYK) algorithm is a dynamic programming algorithm for parsing sequences. The CYK algorithm has been applied to this problem for other classes of grammars such as CFGs, SCFGs [Durbin et al., 1998]

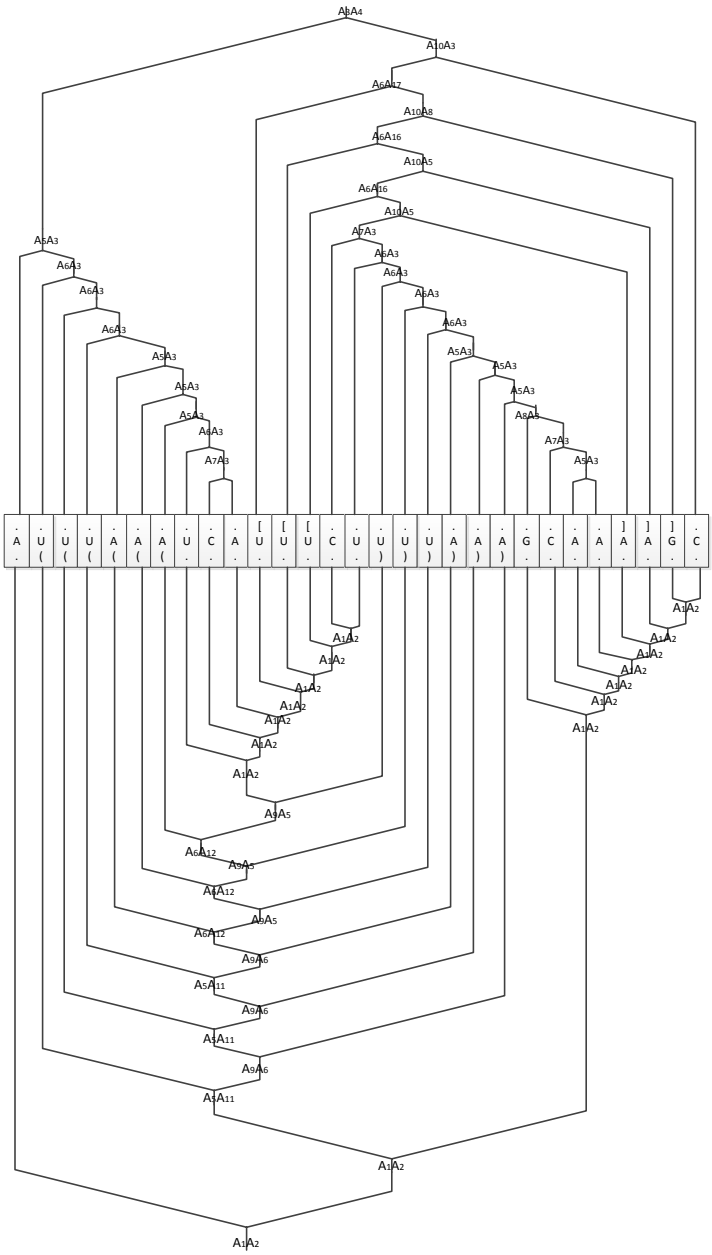


Figure 4.4: This is a diagram the shows how the parse tree of the sequence *auuuaaaucauuucuuuuaaagcaaagc* (ID PKB287 in PseudoBase++) is constructed of the grammar in Equation 5.3.

and CGs [Okhotin, 2001]. I will extend this algorithm to apply it to SCGs and it will have a time complexity of $O(n^3m)$ and space complexity of $O(n^2m)$ where n is the length of a sequence and m is the number of non-terminals in the grammar.

Let $G = (\Sigma, N, R, S, \Phi)$ be a SCG. The Stochastic CYK algorithm is a dynamic programming algorithm that will use two upper-triangular three dimensional matrices called γ and τ for parsing sequences according to G. Let $x = a_1 \dots a_L$ be the input RNA sequence. The indices of the matrices will be i, j, A where i is a start position for a subsequence of length j in x and A represents a nonterminal symbol. A is represented by an integer value, because the algorithm will map all non-terminals to A_i where $0 \leq i < |V|$. For example S will become A_0, the next non-terminal will be A_1 and the last non-terminal will be A_n for some $n \geq 0$. Each entry in the γ matrix will represent the probability that the subsequence $a_i \dots a_j$ can be generated by nonterminal A (i.e., the probablity that $A \Rightarrow^* a_i \dots a_j$). An entry in the τ matrix will be a triplet (B, C, k), where B and C are nonterminals that generated part of the subsequence by using k as a cut point. A cut point is a point k in the sequence $a_i a_{i+1} \dots a_j$ where $i \leq k \leq j$. The triplet (B, C, k) will represent $B \Rightarrow^* a_i \dots a_k$ and $C \Rightarrow^* a_{k+1} \dots a_j$.

4.3 Parsing Algorithm

The version of the CYK algorithm that will be implemented for SCGs will have similar upper-triangular three dimensional matrices. My version of the CYK algorithm will only work on grammars that are in binary normal form. A grammar that is in binary normal form will only have productions in the form of $A \rightarrow$

CKY() {
1 L = length of sequence
2 V = number of nonterminals
3 γ = the probability matrix
4 τ = the back tracing matrix
5 for $i = 1$ to L {
6 for $v = 0$ to $V - 1$ {
7 $\gamma(i, i, v) = \Phi(v \to a_i)$
8 $\tau(i, i, v) = (0, 0, 0)$
9 }
10 }
11 for $j = 1$ to L {
12 for $i = j - 1$ down to 1 {
13 for $v = 0$ to $V - 1$ {
14 $\gamma(i, j, v) = \max_{v \to \alpha \in P_v} \sum_{v \to BC \in \alpha} \max_{k=1,2,\ldots,j-1}$
$\{ \gamma(i, k, B) + \gamma(k + 1, j, C) + \Phi(v \to BC)\}$
15 $\tau(i, j, v) = \{\bigcup_{v \to BC \in \alpha}(B, C, k) | \gamma(i, j, v) = \sum_{v \to BC \in \alpha} \max_{k=1,2,\ldots,j-1}$
$\{\gamma(i, k, B) + \gamma(k + 1, j, C) + \Phi(v \to BC)\}\}$
16 }
17 }
18 }
19 }

Figure 4.5: Modified CYK Algorithm for SCGs

$B_1C_1\&B_2C_2\&\ldots\&B_mC_m$ or $A \to a$. All CGs are able to be transformed into binary normal form as shown in Okhotin [2001]. Each grammar must be in binary normal form in order to use the CYK parsing algorithm. Since the productions of SCGs are the same as CGs, the transformation will work for SCGs. Since I hand crafted these grammars they will not need to be converted to binary normal form because I designed them to be in binary normal form.

4.3.1 CYK Algorithm

The algorithm given in Figure 4.5 will work by initializing all the entries on the diagonals in both matrices. This is done by the first set of loops. This information is important because it will represent the productions where one non-terminal will produce a terminal. When a SCG is in binary normal form this is the only type of production that can directly generate a terminal.

With that information filled in the next step for the algorithm is to check if it is possible to reach all the appropriate terminal products at the appropriate locations. This will be accomplished by the second set of loops. It will do so by filling out the rest of the matrix based on possible productions and previous cells in the matrix. The previous cells that will be used to calculate a cell's probability will be all the cells to the left in the same row and all the cells below in the same column.

Line 14 of the CYK algorithm in Figure 4.5 is the line of code that does most of the work in the algorithm. It is also the line I modified so that the CYK algorithm in Durbin et al. [1998] could work on SCGs. The algorithm will check all the productions $v \rightarrow \alpha$ then check all of the $B_iC_i \in \alpha$. This algorithm will check all the conjuncts B_iC_i for all possible cut-points k to parse the current subsequence and find the best one. That is, the algorithm will determine the most likely split of the current subsequence into a prefix which is generated by B_i and the suffix that is generated by C_i. If a production has multiple conjuncts, then the best k for each conjunct will need to be found. All the probabilities for all conjuncts will be added together because log odds are used, and then the production $v \rightarrow \alpha$ with the highest probability is found.

Once the algorithm has finished running the probability of the sequences being

generated will be stored in cell $(1, L, 0)$, which is the probability of $S \Rightarrow^* a_1 \dots a_L$. This cell will represent the probability of a subsequence from 1 to L, which is the whole sequence, being generated starting with S. That is the exact question asked when determining if a grammar can generate a sequence and what the most likely parse tree is. This algorithm will have a running time of $O(n^3 m)$.

Let's consider an example of the CYK algorithm for SCGs. Given the following grammar G where $G = (\Sigma, V, P, S, \Phi)$ with $V = \{S, A_1, A_2\}$ and P defined by

$$\begin{aligned} &S \rightarrow A_1 A_2 \& A_2 A_1 \ (p = 1) \\ &A_1 \rightarrow a \ (p = .5) | g \ (p = .5) \\ &A_2 \rightarrow A_2 A_1 \ (p = .35) | A_1 A_2 \ (p = .35) | u \ (p = .3) \end{aligned} \tag{4.3}$$

The CYK algorithm when run with G to parse the sequence *auga* will form the following tables.

- The 3D matrices expressed as 3 separate 2D matrices at the start of the algorithm:

S

	a	u	g	a
a				
u				
g				
a				

A_1

	a	u	g	a
a				
u				
g				
a				

A_2

	a	u	g	a
a				
u				
g				
a				

- After the first set of loops are run and the terminal productions are filled in. Log odds are used and that is why $\gamma(0,\ 0,\ 1) = -0.693$ and not .5:

S

	a	u	g	a
a	0			
u		0		
g			0	
a				0

A_1

	a	u	g	a
a	-0.693			
u		0		
g			-0.693	
a				-0.693

A_2

	a	u	g	a
a	0			
u		-1.203		
g			0	
a				0

- After the second column is filled in each matrix. Remember that log odds are being used so probability are added together not multiplied. S's production has an 'and' symbol so the probability is equal to the sum of the probability of both conjuncts, A_1A_2 and A_2A_1:

S

	a	u	g	a
a	0	-3.794		
u		0		
g			0	
a				0

A_1

	a	u	g	a
a	-0.693	0		
u		0		
g			-0.693	
a				-0.693

A_2

	a	u	g	a
a	0	0		
u		-1.203		
g			0	
a				0

- After the third column are filled in each matrix.

S

	a	u	g	a
a	0	-3.794	0	
u		0	0	
g			0	
a				0

A_1

	a	u	g	a
a	-0.693	0	0	
u		0	0	
g			-0.693	
a				-0.693

A_2

	a	u	g	a
a	0	0	0	
u		-1.203	0	
g			0	
a				0

- After the fourth and last column is filled in each matrix:

S

	a	u	g	a
a	0	-3.794	0	-10.766
u		0	0	0
g			0	0
a				0

A_1

	a	u	g	a
a	-0.693	0	0	0
u		0	0	0
g			-0.693	0
a				-0.693

A_2

	a	u	g	a
a	0	0	0	0
u		-1.203	0	-4.69
g			0	-2.436
a				0

- Once the algorithm is done, it would have produced a 3D matrix that looks like the tables above. Since cell (1,L,0) is -10.766 it means that the grammar can generate the sequence and the log odd probability to do so is -10.766. A parse tree that will parse this sequence is seen in Figure 4.6.

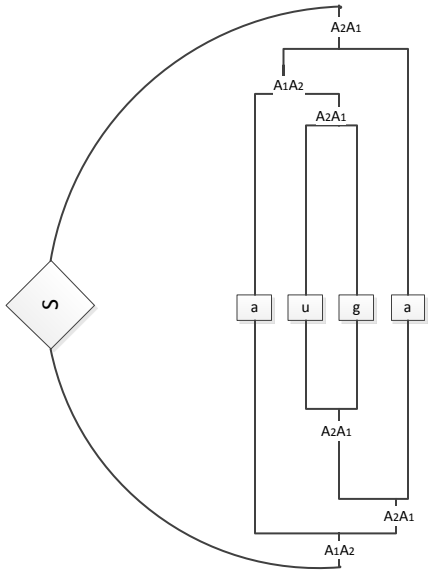


Figure 4.6: A parse tree that will parse the sequence *auga* with the grammar from Equation 4.3.

4.3.2 CYK Back tracing

Since the τ matrix is constructed during the first algorithm, it will keep track of the production that produced the entries. Starting at entry $(1, L, 0)$ this algorithm will trace the steps that the CYK algorithm did to produce the sequence. It will do so by first seeing the productions that were involved in producing entry $(1, L, 0)$.

```
backTrace() {
1    create Stack s
2    s.push(1, L, 0)
3    while (s is not empty) {
4        (i, j, v) = s.pop
5          if τ(i, j, v) = (0, 0, 0) {
6              attach x_i as a child of v
7          } else {
8              τ(i, j, v) = {(B_i, C_i, k_i) : 1 ≤ i < l }
9              for r = 1 to l {
10                 attach B_r and C_r as children to v
11                 s.push(k_r + 1, j, C_r)
12                 s.push(i, k_r, B_r)
13             }
14       }
15   }
16 }
```

Figure 4.7: CKY backtracing algorithm

Then the algorithm will push that information on to a stack. After that, it will pop another entry off the stack and see what was involved in producing it. This will happen until the stack is empty. This algorithm will end up with the complete parse tree of this sequence. If there is no entry for $(1, L, 0)$ this means that grammar is unable to produce the sequence. This algorithm is given in Figure 4.7.

Chapter 5

Results

5.1 Training Data

I obtained RNA strands with known structure from a online database called PseudoBase++ Taufer et al. [2009]. This is a database of all RNA strands with known secondary structure which include pseudoknots. There are currently 304 pseudoknotted secondary structured RNA sequences on PseudoBase++. One of the grammars will be trained and tested on all the RNA sequences but others will be trained on a subset of these 304 sequences.

The subset that will be used is one that will only contain H-type pseudoknots. There is a total of 236 RNA sequences that are H-type pseudoknots. One of those 236 has an H-type structure that differs for the other. This is because it has a bulge that contains hairpins. Since it differs from all others I removed it from the subset of H-type pseudoknots, leaving this to be a set of 235 sequences.

From the 235 sequences, statistics were gathered about the length of the prefix, stem, loop and suffix. The prefix refers to the sequence of nucleotides before a stem

and the suffix is the sequence at follows the stem. The prefix and suffix are somewhat artificial measurements because they are measurements that will just apply to the H-type structure in PseudoBase++, which are only sequences which directly contain pseudoknots and not the surrounding sequence. These statistics were gathered for the round and square brackets independently. Also, to keep all the sequences the same the first set of brackets was always considered to be round brackets, so in other words the sequences were relabelled so they followed this pattern. The average length of an H-type pseudoknot in PseudoBase++ is 39.29 nucleotides with the shortest being 21 and the longest is 121 nucleotides.

5.2 Grammars

In the thesis I designed three grammars to predict RNA pseudoknotted secondary structure. The first was a grammar to predict all types of pseudoknots. Then the next was to only predict H-type pseudoknots and lastly an improved version of the grammar to only predict H-type pseudoknots.

Grammars will be trained using the gatherCount algorithm in Section 4.1. The 235 sequences will be split in half randomly with half being used for training and the other half used for testing. After I craft and train a grammar, I evaluate how well the grammar predicts the pseudoknotted secondary structure. I will test both the sensitivity (Equation 5.1) and specificity (Equation 5.2) of the grammar. These are standard measurements of evaluation for prediction algorithms. I will be testing to see if the grammars are predicting bonds correctly. This means that true positives will occur when the given SCG correctly predicts that there is a bond. A false positive

occurs when it predicts a bond where there is none. A true negative occurs when it correctly predicts no bond. A false negative occurs when it misses a bond that it should have predicted. An example of these prediction types can be seen in Figure 5.1.

$$\text{Sensitivity} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \tag{5.1}$$

$$\text{Specificity} = \frac{\text{true negatives}}{\text{true negatives} + \text{false positives}} \tag{5.2}$$

```
Sequence         agugguuaucccuccacuuaaaucgaaggg
Actual           ((((.....[[[[[))))......].]]]]
Predicted round  :((((::::::::))))::::::::::::::
Predicted square :::::::::[[[[[:::::::::::]:]]]]
Misses           x   x        x   x
```

Figure 5.1: Sample of output of a prediction by my grammar in Appendix A, sequence id PKB62 on PseudoBase++. The top sequence is the RNA strand, the next sequence sequence is to the actual structure of the pseudoknot. The third sequence is how the algorithm predicted the round brackets and the last sequence is the algorithm's prediction of the square brackets. This prediction has four mispredictions represented by the x's at the bottom. The first and fourth x's are examples of a false negative, the second x is an example of a false positive and the third is an example of a double prediction which is always a false positive.

5.2.1 General pseudoknot grammar

Equation 5.3 is the general pseudoknot grammar that I used to attempt to predict RNA secondary structure. I used a simple form of training that would determine

what productions were used to generate the secondary structure and the count of those productions.

This form of training counts the round and square brackets independently; when considering round brackets, square brackets are treated as unpaired nucleotides. As well, when training for square brackets, round brackets were treated as unpaired nucleotides. The algorithm runs through the sequence twice, once for round brackets and once for square.

When going through the sequence and its structure, it will match a corresponding production to the structure. When deciding what production is being used to generate a structural element it will look at the current element and the ones before and after the current element. This is important because what is before and after will affect the productions. It will also have to use a stack so it can recognise a base pair of the structure accurately. It will only count productions when popping off the stack, as this will prevent counting a production twice.

First, if a non-terminal only has one production then the probability will be one. This is the case for the set of non-terminals $\{S$, $A4$, $A5$, $A6$, $A7$, $A8$, $A11$, $A12$, $A13$, $A14$, $A15$, $A16$, $A17$, $A18\}$ in Equation 5.3. These non-terminals were needed so the grammar could be in binary normal formal.

Then probabilities will need to be assigned to non-terminals that have many productions. For the productions $A1 \rightarrow a \mid u \mid c \mid g$ these probabilities will be the probability of that unpaired nucleotide being present in the round bracket training. For the productions $A1 \rightarrow A5A11 \mid A6A12 \mid A7A13 \mid A8A14 \mid A8A11 \mid A6A13$ these probabilities will be the probabilities of this base pair appearing in the round bracket

training. These are base pairs because for example $A5A11 \Rightarrow aA11 \Rightarrow aA9A6 \Rightarrow aA9u$. This will correspond with the base pair au.

The productions $A2 \rightarrow A1A2$ will represent the probability of the grammar continuing to produce more unpaired nucleotides and base pairs. The remaining productions for $A2$ will have the same probability as their corresponding production in $A1$ but will be scaled down by the probability of $A2 \rightarrow A1A2$.

$A3 \rightarrow a \mid u \mid c \mid g$ will be the probability of a sequence of unpaired nucleotides ending with that nucleotide for the square brackets. $A3 \rightarrow A5A3 \mid A6A3 \mid A7A3 \mid A8A3$ will be equal to the probability of a sequence of unpaired nucleotides having this nucleotide for the square bracket training.

$A9 \rightarrow A1A2$ will be the probability of the grammar stopping a stem and going into a loop. $A1 \rightarrow A5A11 \mid A6A12 \mid A7A13 \mid A8A14 \mid A8A11 \mid A6A13$ will be equal to the probability of a base pair being in a stem for the round bracket training.

For the productions $A10 \rightarrow A5A15 \mid A6A16 \mid A7A17 \mid A8A18 \mid A8A15 \mid A6A17$, these will have probabilities equal to probability of this base pair being in a stem for square bracket training. For the productions $A10 \rightarrow A5A10 \mid A6A10 \mid A7A10 \mid A8A10 \mid A8A15 \mid A6A17 \mid A10A5 \mid A10A6 \mid A10A7 \mid A10A8$, these will be the probability of bugles happening in a stem. For the productions $A10 \rightarrow A5A3 \mid A6A3 \mid A7A3 \mid A8A3$ these will represent the probability of a stem ending with a loop starting with that unpaired nucleotide.

$$
\begin{aligned}
&S \rightarrow A_1A_2\&A_3A_4\\
&A_1 \rightarrow a|u|c|g|A_5A_{11}|A_6A_{12}|A_7A_{13}|A_8A_{14}|A_8A_{11}|A_6A_{13}\\
&A_2 \rightarrow A_1A_2|a|u|c|g|A_5A_{11}|A_6A_{12}|A_7A_{13}|A_8A_{14}|A_8A_{11}|A_6A_{13}\\
&A_3 \rightarrow a|u|c|g|A_5A_3|A_6A_3|A_7A_3|A_8A_3\\
&A_4 \rightarrow A_{10}A_3\\
&A_5 \rightarrow a\\
&A_6 \rightarrow u\\
&A_7 \rightarrow c\\
&A_8 \rightarrow g\\
&A_9 \rightarrow A_1A_2|A_5A_{11}|A_6A_{12}|A_7A_{13}|A_8A_{14}|A_8A_{11}|A_6A_{13}\\
&A_{10} \rightarrow A_5A_{15}|A_6A_{16}|A_7A_{17}|A_8A_{18}|A_8A_{15}|A_6A_{17}\\
&A_{10} \rightarrow A_5A_{10}|A_6A_{10}|A_7A_{10}|A_8A_{10}|A_{10}A_5|A_{10}A_6|A_{10}A_7|A_{10}A_8|A_5A_3|A_6A_3|A_7A_3|A_8A_3\\
&A_{11} \rightarrow A_9A_6\\
&A_{12} \rightarrow A_9A_5\\
&A_{13} \rightarrow A_9A_8\\
&A_{14} \rightarrow A_9A_7\\
&A_{15} \rightarrow A_{10}A_6\\
&A_{16} \rightarrow A_{10}A_5\\
&A_{17} \rightarrow A_{10}A_8\\
&A_{18} \rightarrow A_{10}A_7
\end{aligned}
\tag{5.3}
$$

The result of the grammar and training technique did not yield good results. The sensitivity was on average 0.248 and specificity was 0.8. The structure it would often

predict would have the round and square brackets in the same position, which is counted as a misprediction. This is what lead to the low sensitivity. The specificity was high because the positions with the double bracket prediction did align with either a square or round bracket, which means the grammar rarely predicted a bracket where there was not one. As seen in Figure 5.2 both set of brackets are aligned with the square brackets in the actual structure.

```
Sequence         ggggugcgacucccccgucuauccugaacgucaucaggacca
Actual           ..............(((...[[[[[[)))...]]]]]]...
Predicted round  ::::::::::::::::::::((((((:::::))))))::::
Predicted square ::::::::::::::::::::[[[[[[:::::]]]]]]::::
Misses                          xxx   xxxxxxxxx   xxxxxx
```

Figure 5.2: Sample of output of a prediction by the first grammar, sequence id PKB13 on PseudoBase++. The top sequence is the RNA strand, the next sequence sequence is to the actual structure of the pseudoknot. The third sequence is how the algorithm predicted the round brackets and the last sequence is the algorithm's prediction of the square brackets.

5.2.2 H-type grammar

To solve the problem of double bracket prediction, I developed a new grammar. Unlike the first grammar this one would be built to only predict one type of pseudoknot. Since the H-type is the most common type (77% of pseudoknots in PseudoBase++ are H-type), I chose to predict H-type. Out of the 236 H-type pseudoknots I excluded one (sequence and structure given in Figure 5.3); all other H-type pseudoknots will follow a pattern of open round brackets, then open square brackets, then close round brackets and finally close square brackets. This sequences will have closed

round brackets before opening square bracket. After that sequence is excluded from the H-type pseudoknots, it leaves a set of 235 RNA sequences.

```
CCUCCCGGGAGA~ACUGCCUGAUAGGGUGCUUGCGAGUGCCCCGGGAGGUCUCGUAG
((((((((((((.~.))(((......)))((.[[[[[[.))))))))))).]]]]]].
```

Figure 5.3: The excluded H-type pseudoknot. It was excluded because its structure differs significantly from other H-types (ID PKB181 in PseudoBase++).

The next grammar was made to model the structure of the H-type pseudoknot. Counts were gathered about the structure of H-type pseudoknots. Since an H-type is constructed from two interconnected hairpins the structural elements that were counted are prefix, suffix and stem from each hairpin. An example of how two hairpins make a H-type and what is meant by prefix, suffix and stem are seen in Figure 5.4.

All 235 RNA sequences were used to see if there was information on prefix, suffix, stem and loop length that could be useful. The statistics that were collected were minimums, maximums, averages and in general the counts of how many sequences are of a certain length. These statistics were collected for the four structures prefix, stem, loop and suffix, as well as for the sequences themselves.

Since I used all of the sequences to design my grammars but not all of them for training, I had to make sure that I was not over-training. To check for this I gathered the length statistics from 117 random chosen sequences and I did this 100 times. Then for each structure element that I used in the design I found the highest, lowest and average length. I then compared those values to the average lengths for the full sample of all 235 sequences. The results of this test is seen in Figure 5.5. In this figure it is clear to see that values are fairly stable so using all the sequences to design

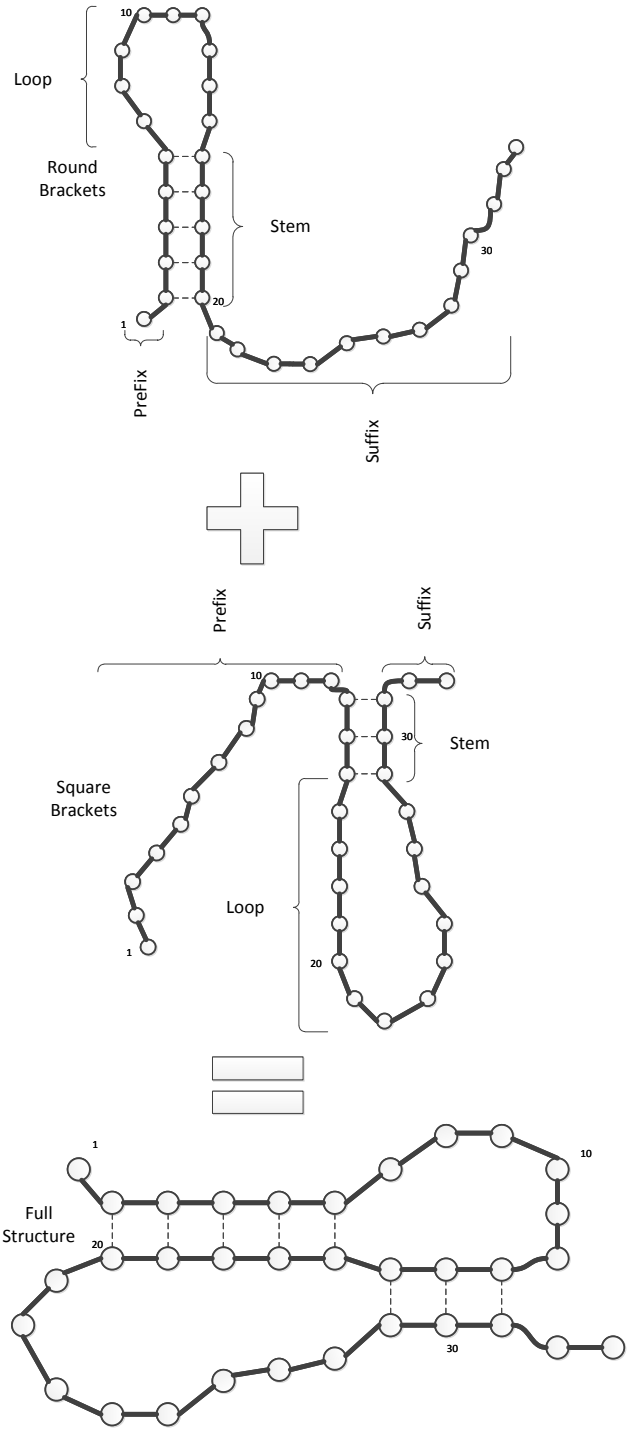


Figure 5.4: This is an example of how a H-type pseudoknot is a combination of two hairpins. If the first two structures are combined together they would produce the H-type pseudoknotted structure at the bottom.

the grammar should not cause over-training.

After gathering those statistics, I constructed a grammar that would predict the secondary structure of RNA with H-type pseudoknots. The grammar that was constructed has 117 non-terminals and 436 productions. Many of these non-terminals and productions were added because the grammar had to be in binary normal form; this means there were many non-terminals that needed to be created to allow for base pairs (see the example in Equation 5.4 and 5.5).

In the examples below A_n refers to the non-terminal that is next in the chain when generating a chain of productions and A_a, A_u, A_c, A_g are non-terminals that only produce the terminal symbols a, u, c, g respectively. Here is an example of productions not in binary normal form:

$$A \rightarrow aA_nu|uA_na|gA_nc|cA_ng|g_nAu|uA_ng \tag{5.4}$$

Here is an example of a same productions in binary normal form;

$$\begin{aligned} A_1 &\rightarrow A_aA_2|A_uA_3|A_cA_4|A_gA_5|A_gA_3|A_uA_5 \\ A_2 &\rightarrow A_nA_u \\ A_3 &\rightarrow A_nA_a \\ A_4 &\rightarrow A_nA_c \\ A_5 &\rightarrow A_nA_g \end{aligned} \tag{5.5}$$

The grammar was run 100 times with 117 sequences being used for training and 118 sequences being used for testing. The sequences that were used for the tests were randomly chosen each time. This grammar has an average sensitivity of 0.743 and an

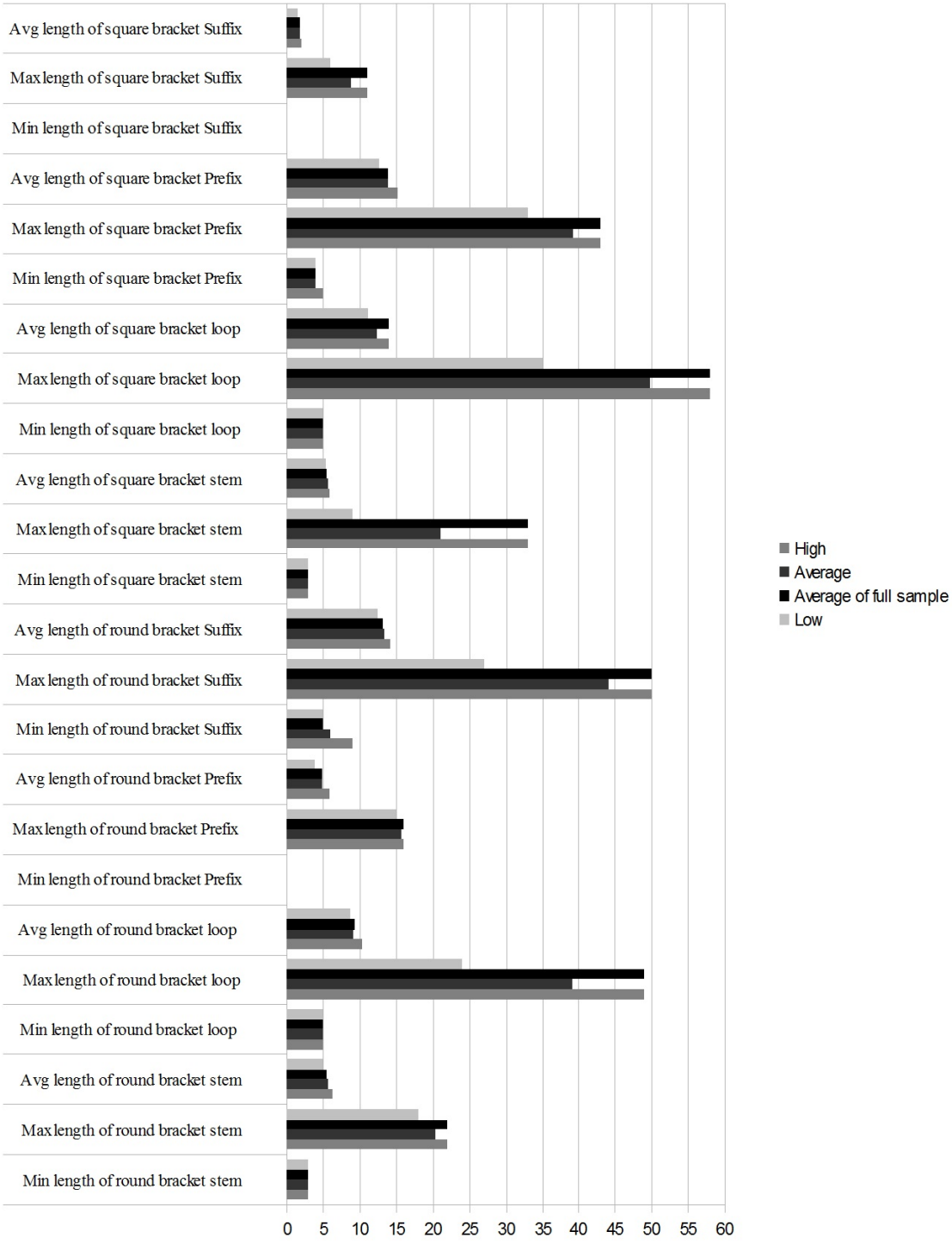


Figure 5.5: The data used in this chart was gathered from a sample of 117 sequence that was run 100 times. It shows the maximum, minimum and average length for each of the structure types that I incorporate into the design of my grammars.

average specificity of 0.895. By designing a grammar that was only meant to predict H-type pseudoknots in RNA secondary structure, my predictions saw an increase in both sensitivity and specificity.

To express the importance of pseudocounts for this grammar, when pseudocounts are removed the average sensitivity drops to 0.418 and the average specificity to 0.838. This is a significant drop in sensitivity because without pseudocounts the grammar is restricted from making some predictions.

5.2.3 Improved H-type grammar

There was one type of structure that was left out of the first grammar to predict H-type pseudoknots. That structure is the loop which is the sequence of unpaired nucleotides that is formed when base pairs form a stem. When the information about loop length is added, the grammar will increase the number of non-terminals to 133 and the number of productions to 552. The grammar was ran 100 times with 117 sequences being used for training and 118 sequences being used for testing. The sequences that were used for the tests were randomly chosen each time. This grammar has an average sensitivity of 0.754 and an average specificity of 0.891, thus with the addition of the loop productions in the grammar I saw a slight increase in sensitivity.

The last grammar will also get an average of 25 sequences out of 117 that are 100% correctly predicted. The H-type prediction grammar without loop information will get an average of 18 sequences 100% correct and the first grammar will get an average of 3 sequences 100% correct. An example of this in seen in Figure 5.6.

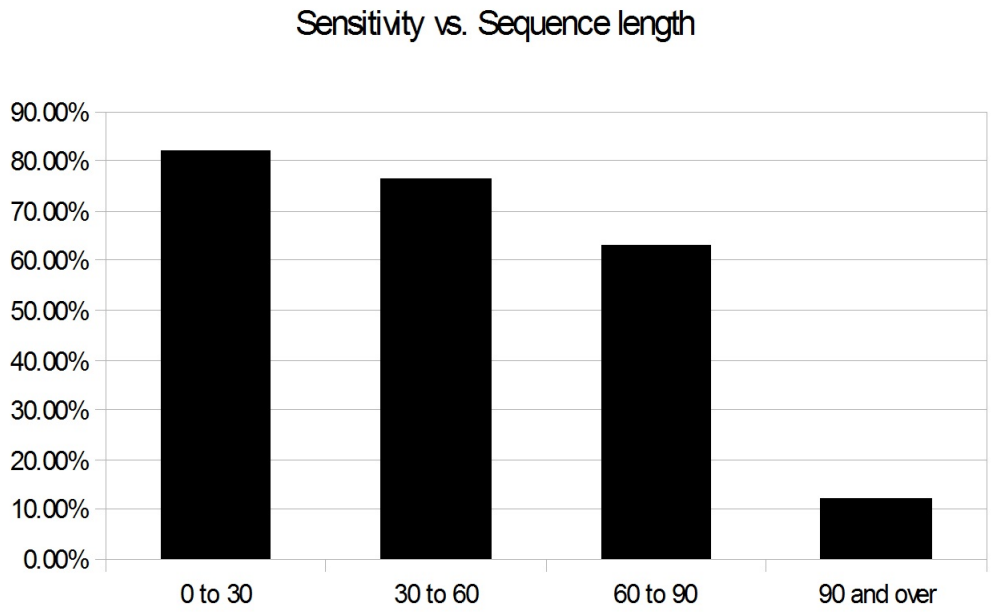
The last grammar for predicting H-type pseudoknots does have a much higher

```
Sequence         ggggugcaacucccccgucuauccuggacgucaccaggacca
Actual           ..............(((...[[[[[[)))...]]]]]]...
Predicted round  ::::::::::::::(((:::::::::)))::::::::::::
Predicted square ::::::::::::::::::::::[[[[[[::::::]]]]]]:::
```

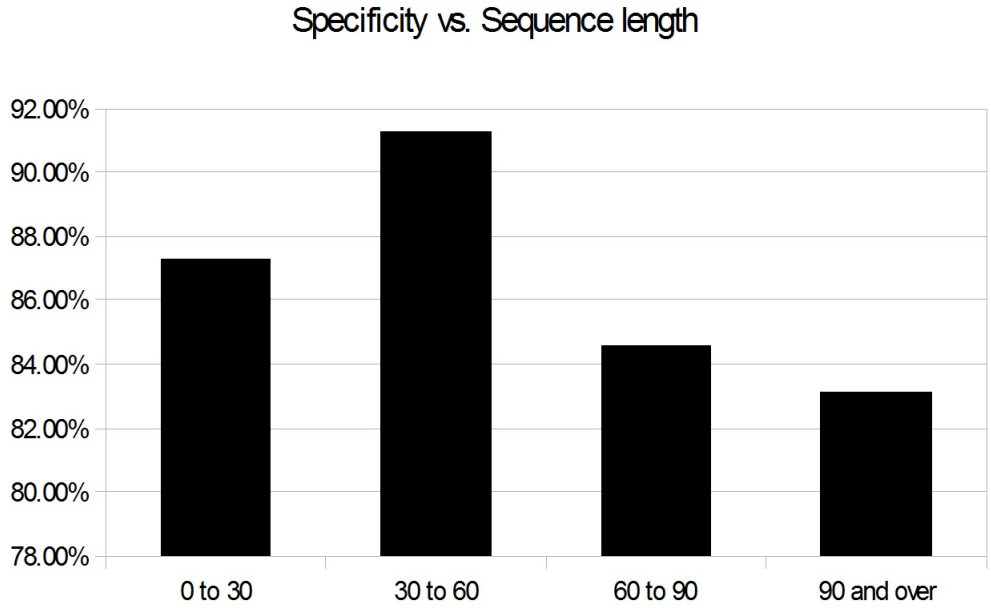
Figure 5.6: Sample of output that is 100% correct, sequence id PKB6 on PseudoBase++. The top sequence is the RNA strand, the next sequence will be the actual structure of the pseudoknot. The third sequence is how the algorithm predicted the round brackets and the last sequence is the algorithm's prediction of the square brackets.

sensitivity when it is predicting sequences that have length less than 60. After that point the sensitivity drops off. This is not seen for specificity which is fairly high no matter what size sequences it was predicting. These statistics can be seen in Figure 5.7. Since the average length of a H-type sequence in about 40 the drop off after 60 will not affect too many of the sequences.

As well as testing to see how well this final grammar did on sequences of different length, I was also interested in seeing how different training sizes would do. I tested the grammar on three different training sizes, one was 25% training (58 sequences) and 75% testing (177 sequences), one was 50% training 117 sequences) and 50% testing (118 sequences) and the last was 75% training (175 sequences) and 25% testing (60 sequences). The result of this test can been seen in Figure 5.8. All the results given in the chart are an average of over 100 tests. This chart shows that the size of training data has little effect on the sensitivity and specificity of the predictions.



(a) Sensitivity



(b) Specificity

Figure 5.7: (a) A break down of average sensitivity for 100 tests based on various length sequences. (b) A break down of an average specificity for 100 tests based on various length sequences.

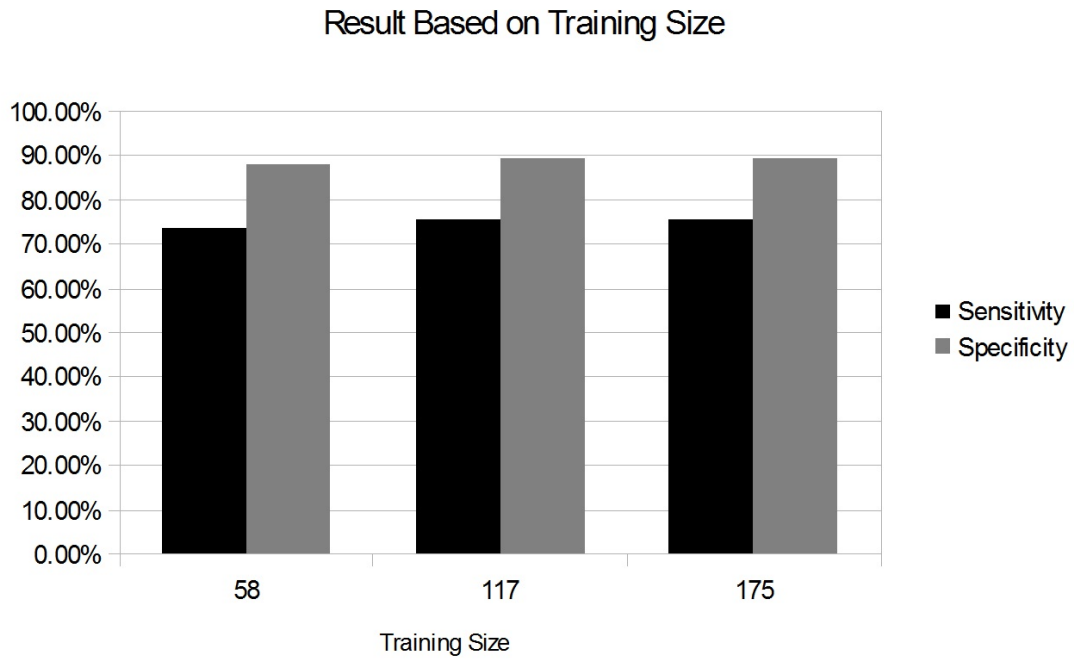


Figure 5.8: The data used in this chart was gathered from a samples that was run 100 times. It shows the sensitivity and specificity for each of the training sizes.

5.3 Time and Space Complexity

As described in the Chatper 3 the dynamic programming methods called pknots by Rivas and Eddy [1999] has $O(n^6)$ time and $O(n^4)$ space complexity. Also the pair scholastic tree adjoining grammar designed by Matsui et al. [2005] has a running time of $O(n^5)$. My algorithm will have a time complexity of $O(n^3m)$ and space complexity of $O(n^2m)$ where n is the length of a sequence and m is the number of non-terminals in the grammar. This is because it will be using a modified CYK algorithm that has run time and complexity as stated in Durbin et al. [1998]. Even though the Matsui et al. [2005] model has a higher sensitivity, their algorithm is slower and does not have

the versatility of being able to predict all H-types sequences rather than prediction based on phylogenetic family.

Chapter 6

Conclusion

In this thesis I showed that a CG can be successfully extended to a SCG. I did this by adding an extra component to allow it to incorporate probabilities. This allows a SCG to be used for machine learning. Since there was a new component, I had to change the CYK algorithm so that it will be able to work for SCGs.

Once the SCG was shown to be a valid model and was able to be trained successfully, I set out to build grammars that were able to predict RNA pseudoknotted secondary structure. I was unable to build a grammar that was able to predict all types of pseudoknots successfully. However, when I reduced the predictions to only H-type pseudoknots, the grammars that I built for predicting RNA sequences with H-type pseudoknots only were very successful with an average sensitivity of over 75% and an average specificity of over 89%.

6.1 Future Work

I introduced a grammar that can successfully predict H-type pseudoknotted secondary structure. It would be interesting to see if grammars can be made to predict other pseudoknot types. This may be difficult because there are few training examples for other types of pseudoknots, as there are only about 70 non-H-type psuedoknotted sequences in PseudoBase++. As well, it remains to be seen if there is a grammar that could predict every type of pseudoknot. As well, it would be interesting to find other kinds of problems that this class of grammars can be applied to in bioinformatics and other fields. One problem that SCGs could be applied to is RNA-RNA interaction prediction which has had grammars applied to it before, as seen in Kato et al. [2006a].

There are also algorithms that are used for training grammars called the Inside-Outside algorithms. The inside algorithm will compute the probability that a sequence can generate a parse tree given a SCG. The outside algorithm will compute the probability of all remaining parse trees with holes in them. These algorithms are used when training a grammar on data that is not labelled. Since all data was labelled in this thesis, I did not implement this algorithm. But it would be an interesting algorithm to implement and would allow this class of grammars to be applied to problems where labelled training data us unavailable.

One more additional problem is to see if SCG can be used to predict whether or not a sequence contains pseudoknots or not and to predict the pseudoknots if there are any. One way of doing this would be to build a grammar that could explicitly say if a sequence has a pseudoknot portion or not. This would involve adding productions in the grammar to predict pseudoknot-free secondary structure which are independent of

the pseudoknotted productions. Then the grammar would make a top level decision on which set of productions (pseudoknot or pseudoknot-free) to use based on probability.

Another technique that can be used to predict if a unknown sequence contains pseudoknots is to use a sliding window. One idea for doing this is to design two grammars, one to predict pseudoknotted structure and one to predict pseudoknot-free structure. Each of these grammars would be run on the window as it slides across a sequence. By examining the results of these grammars on RNA with known structure secondary, both pseudoknotted and pseudoknot-free, a threshold may be able to be developed. Then by running these grammars on RNA with unknown secondary structure and comparing the results to the threshold, a decision about whether the structure is pseudoknotted or not could be made.

Appendix A

H-type Grammar

$S \rightarrow aA_5\&aA_{52}|uA_5\&uA_{52}|cA_5\&cA_{52}|gA_5\&gA_{52}$

$S \rightarrow A_4A_{14}\&aA_{52}|A_4A_{14}\&uA_{52}|A_4A_{14}\&cA_{52}|A_4A_{14}\&gA_{52}$

Figure A.1: Start productions

$A_1 \rightarrow aA_2|uA_2|cA_2|gA_2|A_4A_{14}$

$A_2 \rightarrow aA_3|uA_3|cA_3|gA_3|A_4A_{14}$

$A_3 \rightarrow aA_3|uA_3|cA_3|gA_3|A_4A_{14}$

$A_4 \rightarrow aA_5u|uA_5a|cA_5g|gA_5c|uA_5g|gA_5u$

$A_4 \rightarrow aA_4|uA_4|cA_4|gA_4|A_4a|A_4u|A_4c|A_4g|A_{11}A_{12}$

Figure A.2: Productions for round bracket prefix

$A_5 \rightarrow aA_6u|uA_6a|cA_6g|gA_6c|uA_6g|gA_6u$

$A_5 \rightarrow aA_5|uA_5|cA_5|gA_5|A_5a|A_5u|A_5c|A_5g|A_{11}A_{12}$

$A_6 \rightarrow aA_7u|uA_7a|cA_7g|gA_7c|uA_7g|gA_7u$

$A_6 \rightarrow aA_6|uA_6|cA_6|gA_6|A_6a|A_6u|A_6c|A_6g|A_{11}A_{12}$

$A_7 \rightarrow aA_8u|uA_8a|cA_8g|gA_8c|uA_8g|gA_8u$

$A_7 \rightarrow aA_7|uA_7|cA_7|gA_7|A_7a|A_7u|A_7c|A_7g|A_{11}A_{12}$

$A_8 \rightarrow aA_9u|uA_9a|cA_9g|gA_9c|uA_9g|gA_9u$

$A_8 \rightarrow aA_8|uA_8|cA_8|gA_8|A_8a|A_8u|A_8c|A_8g|A_{11}A_{12}$

$A_9 \rightarrow aA_{10}u|uA_{10}a|cA_{10}g|gA_{10}c|uA_{10}g|gA_{10}u$

$A_9 \rightarrow aA_9|uA_9|cA_9|gA_9|A_9a|A_9u|A_9c|A_9g|A_{11}A_{12}$

$A_{10} \rightarrow aA_{10}u|uA_{10}a|cA_{10}g|gA_{10}c|uA_{10}g|gA_{10}u$

$A_{10} \rightarrow aA_{10}|uA_{10}|cA_{10}|gA_{10}|A_{10}a|A_{10}u|A_{10}c|A_{10}g|A_{11}A_{12}$

$A_{11} \rightarrow a|u|g|c$

$A_{12} \rightarrow A_{11}A_{12}|A_{11}A_{11}$

Figure A.3: Productions for round bracket stem and loop

$A_{13} \rightarrow a|u|g|c$

$A_{14} \rightarrow A_{13}A_{15}$

$A_{15} \rightarrow A_{13}A_{16}$

$A_{16} \rightarrow A_{13}A_{17}$

$A_{17} \rightarrow A_{13}A_{18}$

$A_{18} \rightarrow A_{13}A_{19}$

$A_{19} \rightarrow A_{13}A_{19}|A_{13}A_{13}$

Figure A.4: Productions for round bracket suffix

$A_{20} \rightarrow aA_{21}|uA_{21}|cA_{21}|gA_{21}$

$A_{21} \rightarrow aA_{22}|uA_{22}|cA_{22}|gA_{22}$

$A_{22} \rightarrow aA_{23}|uA_{23}|cA_{23}|gA_{23}|A_{23}A_{36}|A_{23}A_{37}|aA_{24}u|uA_{24}a|cA_{24}g|gA_{24}c|uA_{24}g|gA_{25}u$

Figure A.5: Productions for square bracket prefix

$A_{23} \rightarrow aA_{24}u|uA_{24}a|cA_{24}g|gA_{24}c|uA_{24}g|gA_{24}u$

$A_{23} \rightarrow aA_{23}|uA_{23}|cA_{23}|gA_{23}|A_{23}a|A_{23}u|A_{23}c|A_{23}g|A_{34}A_{35}$

$A_{24} \rightarrow aA_{25}u|uA_{25}a|cA_{25}g|gA_{25}c|uA_{25}g|gA_{25}u$

$A_{24} \rightarrow aA_{24}|uA_{24}|cA_{24}|gA_{24}|A_{24}a|A_{24}u|A_{24}c|A_{24}g|A_{34}A_{35}$

$A_{25} \rightarrow aA_{26}u|uA_{26}a|cA_{26}g|gA_{26}c|uA_{26}g|gA_{26}u$

$A_{25} \rightarrow aA_{25}|uA_{25}|cA_{25}|gA_{25}|A_{25}a|A_{25}u|A_{25}c|A_{25}g|A_{34}A_{35}$

$A_{26} \rightarrow aA_{27}u|uA_{27}a|cA_{27}g|gA_{27}c|uA_{27}g|gA_{27}u$

$A_{26} \rightarrow aA_{26}|uA_{26}|cA_{26}|gA_{26}|A_{26}a|A_{26}u|A_{26}c|A_{26}g|A_{34}A_{35}$

$A_{27} \rightarrow aA_{28}u|uA_{28}a|cA_{28}g|gA_{28}c|uA_{28}g|gA_{28}u$

$A_{27} \rightarrow aA_{27}|uA_{27}|cA_{27}|gA_{27}|A_{27}a|A_{27}u|A_{27}c|A_{27}g|A_{34}A_{35}$

$A_{28} \rightarrow aA_{29}u|uA_{29}a|cA_{29}g|gA_{29}c|uA_{29}g|gA_{29}u$

$A_{28} \rightarrow aA_{28}|uA_{28}|cA_{28}|gA_{28}|A_{28}a|A_{28}u|A_{28}c|A_{28}g|A_{34}A_{35}$

$A_{29} \rightarrow aA_{30}u|uA_{30}a|cA_{30}g|gA_{30}c|uA_{30}g|gA_{30}u$

$A_{29} \rightarrow aA_{29}|uA_{29}|cA_{29}|gA_{29}|A_{29}a|A_{29}u|A_{29}c|A_{29}g|A_{34}A_{35}$

$A_{30} \rightarrow aA_{31}u|uA_{31}a|cA_{31}g|gA_{31}c|uA_{31}g|gA_{31}u$

$A_{30} \rightarrow aA_{30}|uA_{30}|cA_{30}|gA_{30}|A_{30}a|A_{30}u|A_{30}c|A_{30}g|A_{34}A_{35}$

$A_{31} \rightarrow aA_{32}u|uA_{32}a|cA_{32}g|gA_{32}c|uA_{32}g|gA_{32}u$

$A_{31} \rightarrow aA_{31}|uA_{31}|cA_{31}|gA_{31}|A_{31}a|A_{31}u|A_{31}c|A_{31}g|A_{34}A_{35}$

$A_{32} \rightarrow aA_{33}u|uA_{33}a|cA_{33}g|gA_{33}c|uA_{33}g|gA_{33}u$

$A_{32} \rightarrow aA_{32}|uA_{32}|cA_{32}|gA_{32}|A_{32}a|A_{32}u|A_{32}c|A_{32}g|A_{34}A_{35}$

$A_{33} \rightarrow aA_{33}u|uA_{33}a|cA_{33}g|gA_{33}c|uA_{33}g|gA_{33}u$

$A_{33} \rightarrow aA_{33}|uA_{33}|cA_{33}|gA_{33}|A_{33}a|A_{33}u|A_{33}c|A_{33}g|A_{34}A_{35}$

$A_{34} \rightarrow a|u|g|c$

$A_{35} \rightarrow A_{34}A_{35}|A_{34}A_{34}$

Figure A.6: Productions for square bracket stem and loop

$A_{36} \rightarrow a|u|g|c$

$A_{37} \rightarrow A_{36}A_{38}|A_{36}A_{36}$

$A_{38} \rightarrow A_{36}A_{39}|A_{36}A_{36}$

$A_{39} \rightarrow A_{36}A_{40}|A_{36}A_{36}$

$A_{40} \rightarrow A_{36}A_{40}|A_{36}A_{36}$

Figure A.7: Productions for round bracket prefix

Bibliography

T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104(1–3):45–62, 2000.

P. Bildi and S. Brunak. *Bioinformatics: The Machine Learning Approach*. The MIT Press, 2001.

I. Brierley, S. Pennell, and R. J. C. Gilbert. Viral RNA pseudoknots: versatile motifs in gene expression and replication. *Nature Reviews Microbiology*, 5:598–610, August 2007.

L. Cai, R. Malmberg, and Y. Wu. Stochastic model of RNA pseudoknotted structures: a grammatical approach. *Bioinformatics*, 19(1):66–73, 2003.

R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge University Press, 1998.

S. Eddy and R. Durbin. RNA sequence analysis using covariance models. *Nucleic Acids Research*, 22(11):2079–2088, 1994.

X.-Y. Fang, Z.-G. Luo, and Z.-H. Wang. Predicting RNA secondary structure us-

ing profile stochastic context-free grammars and phylogenic analysis. *Journal of Computer Science and Technology*, 23(4):582–589, 2008.

H. Jabbari, A. Condon, and S. Zhao. Novel and efficient RNA secondary structure prediction using hierarachical folding. *Journal of Computational Biology*, 15(2): 139–163, 2008.

Y. Kato, T. Akutsu, and H. Seki. A grammatical approach to RNARNA interaction prediction. *Pattern Recognition*, 42(4):531–538, 2006a.

Y. Kato, H. Seki, and T. Kasami. RNA pseudoknotted structure prediction using stochastic multiple context-free grammar. *IPSJ Digital Courier*, 2:655–664, 2006b.

P. Linz. *An Introduction to Formal Languages and Automata*. 2006.

H. Matsui, K. Sato, and Y. Sakakibara. Pair stochastic tree adjoining grammars for aligning and predicting pseudoknot RNA structures. *Bioinformatics*, 21(11): 2611–2617, 2005.

J. Nowakowski and I. Tinoco, Jr. RNA structure and stability. *Seminars in Virology*, 8(3):153–165, 1997.

A. Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

S. Rajasekaran, S. A. Seesi, and R. A. Ammar. Improved algorithms for parsing ESLTAGs: A grammatical model suitable for RNA pseudoknots. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(4):619–627, 2010.

E. Rivas and S. R. Eddy. A dynamic programming algorithm for rna structure prediction including pseudoknots. *Journal of Molecular Biology*, 285:2053–2068, 1999.

E. Rivas and S. R. Eddy. The language of RNA: a formal grammar that includes pseudoknots. *Bioinformatics*, 16(4):334–340, 2000.

Y. Sakakibara. Grammatical inference in bioinformatics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1051–1062, 2005.

Y. Sakakibara, M. Brown, R. C. Underwood, I. S. Mian, and D. Haussler. Stochastic context-free grammars for modeling RNA. *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences: Biotechnology Computing*, 5:284–293, 1994.

M. Taufer, A. Licon, R. Araiza, D. Mireles, F. H. D. van Batenburg, A. P. Gultyaev, and M.-Y. Leung. Pseudobase++: an extension of pseudobase for easy searching, formatting and visualization of pseudoknots. *PseudoBase++: an extension of PseudoBase for easy searching, formatting and visualization of pseudoknots*, 37: D127–D134, 2009.

Y. Uemura, A. Hasegawa, S. Kobayashi, and T. Yokomori. Tree adjoining grammars for RNA structure prediction. *Theoretical Computer Science*, 210:277–303, 1999.

J. D. Watson and F. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

B.-J. Yoon and P. P. Vaidyanathan. Structural alignment of RNAs using profile-

csHMMs and its application to RNA homology search: Overview and new results. *IEEE Transactions on Automatic Control*, (Special Issue):10–25, 2008.