Introduction to Stochastic Context Free Grammars

Robert Giegerich

Abstract Stochastic context free grammars are a formalism which plays a prominent role in RNA secondary structure analysis. This chapter provides the theoretical background on stochastic context free grammars. We recall the general definitions and study the basic properties, virtues and shortcomings of stochastic context free grammars. We then introduce two ways in which they are used in RNA secondary structure analysis, secondary structure prediction and RNA family modeling. This prepares for the discussion of applications of stochastic context free grammars in the chapters on *Rfam*, *Infernal*, and *Pfold*.

Key words: RNA structure prediction, stochastic grammars, RNA family models

Version of June 8, 2011

Robert Giegerich
Faculty of Technology and Center of Biotechnology, Bielefeld University, Bielefeld, Germany, e-mail: robert@techfak.uni-bielefeld.de

1 Stochastic context free grammars – definitions

Stochastic context free grammars (SCFGs) were introduced in bioinformatics for the purpose of modeling RNA secondary structure, the original early references being [7, 15]. In computer science, stochastic grammars have a longer tradition, and were studied and used mainly in the field of natural language processing. Early references to that literature are [2, 12]. Both lines of work have evolved quite independently, and the terminology is not always the same. We will adhere to the terminology and notational conventions used in bioinformatics. Let us start with a one-sentence summary of what SCFGs are.

For our readers who are familiar with stochastic modeling and have worked with Hidden Markov Models (HMMs):

> SCFGs are HMMs where the linear path of state transitions is replaced by a tree of states.

For our readers with a background in computer science and formal language theory:

> SCFGs are context free grammars augmented with a probabilistic scoring scheme.

For our readers with a background of neither type:

> Don't worry, this chapter assumes neither kind of previous experience, but starts from first principles.

1.1 Context free grammars

Grammars, languages and derivations

Let $\mathscr{A}$ be a finite set of symbols, called the *alphabet*. $\mathscr{A}^*$ denotes the set of all finite strings of symbols from $\mathscr{A}$, including the empty string, which has no symbols at all. To make it visible, we write ε for the empty string. $|x|$ denotes the length of the string x. x^{-1} denotes the reverse of x. A *formal language* L is simply a subset of $\mathscr{A}^*$. Elements of L are called "words" in formal language theory, "phrases" or "sentences" in linguistics, or "sequences" in bioinformatics.

Formal languages can be described in many ways – the most popular one is the use of grammars.

Definition 1. A *context-free grammar* G is a formal system that generates a language $L(G) \subseteq \mathscr{A}^*$. It uses a set V of *nonterminal symbols*, one of which is designated as the *axiom*, and a set of *derivation rules* (also called *productions*) that have the form $X \to \alpha$, where $X \in V$ and $\alpha \in (V \cup \mathscr{A})^*$.

Grammar *DotPar*.
$\mathscr{A} = \{(,),.\}$, $V = \{S\}$, axiom is S.

production rule			rule name
S	$\to$	ε	*end*
	$\mid$	$.S$	*dot*
	$\mid$	$(S)S$	*pairsplit*

Grammar *Pali*.
$\mathscr{A} = \{a,b\}$, $V = \{P,T\}$, axiom is P.

production rule			rule name
P	$\to$	T	*turn*
	$\mid$	xPy	$pair_{xy}$, for $x,y \in \mathscr{A}$ and $x = y$
T	$\to$	ε	*end*
	$\mid$	xT	add_x, for $x \in \mathscr{A}$

Grammar *Ali*.
$\mathscr{A} = \{a,c,g,t,\$\}$, $V = \{A\}$, axiom is A.

production rule			rule name	
A	$\to$	xAy	$match_{xy}$,	for $x,y \in \mathscr{A} \setminus \{\$\}$ and $x = y$
	$\mid$	xAy	rep_{xy},	for $x,y \in \mathscr{A} \setminus \{\$\}$ and $x \neq y$
	$\mid$	xA	del_x,	for $x \in \mathscr{A} \setminus \{\$\}$
	$\mid$	Ax	ins_x,	for $x \in \mathscr{A} \setminus \{\$\}$
	$\mid$	ε	*end*	

Fig. 1 Three context free grammars. *DotPar* describes RNA structure in dot-parenthesis notation, *Pali* describes palindromes, and *Ali* DNA sequence alignments, allowing for simple insertions and deletions, matches and replacements. Production rules have been named for later reference. Grammars *Pali* and *Ali* use abbreviation. For example in grammar *Ali*, the rule $A \to xAy$ named $match_{xy}$ actually stands for four distinct rules for the indicated choices of x and y with $x = y$, and the rule rep_{xy} has 12 variants.

We shall use upper-case letters for non-terminal symbols and lower-case for symbols from $\mathscr{A}$. The productions serve to derive the words of the language, starting from the axiom. This will be defined more precisely in a moment. In contrast to the non-terminal symbols from V, the symbols from $\mathscr{A}$ are called terminal symbols, because once generated in the cause of a derivation, they are never replaced.

It is customary to collect all rules for the same nonterminal symbol in a rule with a single lefthand side, and several alternatives on the right, separated by a '|'. Hence, $\{A \to \alpha_1, A \to \alpha_2, A \to \alpha_3\}$ becomes $\{A \to \alpha_1 \mid \alpha_2 \mid \alpha_3\}$.

Figure 1.1 shows three simple grammars. There, we have associated names with all productions, which is not foreseen in the standard definition of CFGs, but will be convenient for later reference.

Next we make precise the notion of a derivation:

Definition 2. A derivation of a word $w \in L(G)$ starts with the axiom symbol, and in each step, replaces one of the nonterminal symbols in the emerging string according to one of the productions: When a derivation has already produced (say) xXy, and $X \to \alpha$ is a production of G, we may rewrite xXy to $x\alpha y$.

A derivation can be represented uniquely in the form of a tree. Figure 1.1 shows several derivation trees for the grammars of Figure 1.1. The inner nodes of the tree are labeled with the production names. Terminal symbols are leaf nodes in the derivation tree, and considering just these leaves in left-to-right order, we obtain the string $w \in L(G)$ produced by the derivation. We can also write derivations as formulas: Consider grammar *Ali* with its derivation rules

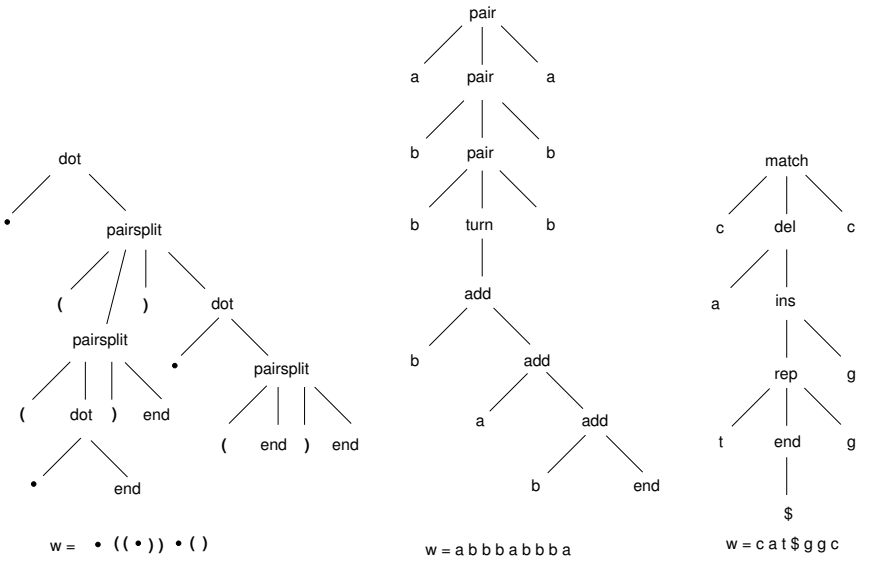

Fig. 2 Three derivation trees for the grammars of Figure 1.1. The *DotPar* tree (left) makes explicit the base pairing in the RNA structure representation $w =$ `".((.)).()"`: matching parenthesis, denoting base pairs, are jointly produced by an application of the *pairsplit* rule. This is the only derivation tree for this w. The *Pali* tree (middle) indicates a palindromic structure of $w = abbbabbba$, where the innermost three letters are considered part of the turn – this is one out of five different derivation trees for this string. The *Ali* tree describes one particular alignment (out of many) for `cat` and `cgg`, written as input $w =$ `cat$ggc`. This particular alignment uses each edit operation once.

$match, del, ins, rep, end$. Interpreting the production names as functions which actually construct trees, we can write a derivation tree as an expression, such as $match(c, del(a, ins(rep(t, end(\$), g), g)), c)$ for the *Ali*-tree in Figure 1.1.

Which are the languages described by the grammars in Figure 1.1?

- Grammar *DotPar* allows to derive all strings which denote RNA secondary structures in the familiar dot-bracket notation, where a dot means an unpaired base and matching parentheses denote two bases forming a base pair.
- Grammar *Pali* allows to derive palindromic strings of the form $p = uvu^{-1}$, which allows for a "turn" v of any length in the middle, where the characters do not have to match up.
- Grammar *Ali* allows to derive two strings from the DNA alphabet, separated by a '$'. Thus, $L(Ali)$ by itself does not appear very interesting, and could be described by an even simpler grammar. But consider *how Ali* derives these strings: A derivation of $x\$y$ actually models an alignment of x and y^{-1}, allowing for base matches and replacements, insertions and deletions. This is made apparent by the production names we have chosen.

Grammar normal forms

The standard definition allows CFGs to contain certain elements of "junk": There could be non-terminal symbols which are useless because they can never be reached in a derivation starting from the axiom. A rule like $D \to BD$, when other, alternative rules for D are lacking, would allow for endless, unproductive derivations, which never get rid of the D non-terminal symbol. And there could be unproductive derivation cycles $C \to B \to D \to C$ that do not produce any terminal symbols. This would blow up, without need, the number of possible derivations for any string that is derived using C. We call a grammar *clean* when it does not have any of these features. Grammars *DotPar*, *Pali*, and *Ali* are clean. Our discussion in the sequel will tacitly assume that grammars are clean.

Even with clean grammars, there are many ways to describe the same formal language with different grammars, i.e. $L(G_1) = L(G_2)$. This is important, as some grammars are more convenient algorithmically than others. In particular, there is always the following normal form, which has especially simple production rules:

Definition 3. A grammar is said to be in *Chomsky Normal Form*[1], if each righthand side holds no more than two symbols from $V \cup \mathscr{A}$.

Chomsky Normal Form of a grammar can always be achieved without changing the language. For example, the rule

$$S_1 \to aS_2bS_1S_3c$$

can be replaced by

$$S_1 \to aA,\ A \to S_2B,\ B \to bC,\ C \to S_1D,\ D \to S_3c$$

where A, B, C, D are new non-terminal symbols. Such a transformation of the productions does not affect the language $L(G)$, but makes the grammar more handsome for the issue we study next: parsing.

Parsing and ambiguity

Given $w \in \mathscr{A}^*$, we want to solve the *word problem*, also known as *syntax checking*: Is $w \in L(G)$? This question is answered by constructing a derivation tree for x, or by showing that no such tree exists.

Definition 4. The construction of a derivation tree for given $w \in \mathscr{A}^*$ is called *parsing*, and the derivation tree (if any) is called a *parse tree* in this context. A CFG G is syntactically *ambiguous*, if there is more than one parse tree for some w. Let $T_G(w)$ denote the set of all parse trees for w.

[1] Our definition is a bit more relaxed than the one found in the formal language literature.

Syntactic ambiguity of context free grammars is sometimes welcome, sometimes a nuisance. For example, when grammars are used to define the syntax of programming languages, ambiguity is to be avoided. If (say) a Java program could be parsed in two ways, machine code implementing different algorithms might be generated from it! Therefore, programming language research mainly studies non-ambiguous grammars, which also allow for very efficient parsing algorithms. An algorithm for checking ambiguity of grammars was intensively sought for – until this problem was proved to be algorithmically undecidable in the 1960s [11].

The situation is completely different in natural language processing, as well as in bioinformatics. Syntactic correctness, $w \in L(G)$, is taken for granted. Here, grammars are typically ambiguous. We have a large number of parse trees for any given word, and we want to select from them a most plausible one, based on a suitable scoring scheme and objective function. Hence, with ambiguous grammars, parsing naturally generalizes to an optimization problem. Adding a probabilistic scoring scheme to CFGs will take us to SCFGs in the next section. Scoring instead with a thermodynamic energy model takes us to RNA folding, as discussed in the Chapter by S. Sukosd et al. in this book.

Considering our example grammars, we note the following:

- Grammar *DotPar* is non-ambiguous. This is easy to check: Given any word $w \in L(DotPar)$, generate a derivation for it, producing w from left to right. In each step, exactly one of the three rules applies, so there are never two different derivation trees for any w.
- With grammar *Pali*, when deriving, for example, *abbbabbba*, we find that there are five different derivations, depending whether we choose $w = uvu^{-1}$ with $u =$ *abbb* and $v = a$, $u =$ *abb* and $v =$ *bab*, $u =$ *ab* and $v =$ *bbabb*, $u = a$ and $v =$ *bbbabbb*, or $u = \varepsilon$ and $v =$ *abbbabbba*. You may argue that the first choice is the "most palindromic" one, but this is sort of your personal opinion. The grammar, per se, does not make such preference.
- Grammar *Ali*, finally, has as many derivations for $x\$y$ as there are alignments of x and y^{-1}. No preference is expressed within the grammar itself, but of course, we can add a suitable scoring scheme and search for an optimal alignment.

CYK parsing

A simple parsing algorithm for ambiguous CFGs, which finds all parses for a word, was independently suggested by Cocke, Younger and Kasami in the 1960s. It is commonly refered to as the CYK-algorithm in bioinformatics.[2]

To decide if (and how) a string w can be parsed according to a particular production, CYK splits w into as many parts as there are symbols on the righthand side.

[2] Amusingly, the natural language processing community consistently refers to it as the CKY algorithm. The reason for such confusion is that there is no joint paper by these three authors. [1] tells the story behind CYK. This classical textbook presents CYK because of its "intuitive simplicity", but remains "doubtful, however, that it will find practical use". Those were the days when a state-of-the-art computer had 65K bytes of memory.

The split is done in all possible ways, and the i-th substring in a split is derived (if possible) from the corresponding i-th symbol in the righthand side. Using our production names as tree construction functions, the logic of a CYK algorithm for a grammar is easily written as a recursive function: $parse_S(w)$ constructs all derivation trees for a word w, starting from non-terminal symbol S. For grammar *DotPar* this leads to the recurrence

$$\begin{aligned} parse_S(w) = & \{end \mid w = \varepsilon\} \\ & \cup \{dot('.',x) \mid w =' .'v,\ x \in parse_S(v)\} \\ & \cup \{pairsplit('(',x,')',y) \mid w =' ('u')'v,\ x \in parse_S(u),\ y \in parse_S(v)\} \end{aligned}$$

In the *pairsplit* case, the split of w must be done in all possible ways, but only the split which chooses the matching '(' and ')' will be successful and deliver a parse tree. In general, a CYK parser has a parser $parse_X$ for each $X \in V$, and the parser has as many cases to distinguish as there are alternative rules for X. The axiom parser called with input w produces all parse trees for w , i.e. $T_G(w)$.

Here it is important that the grammar is clean, otherwise the recursion in the parser may not terminate. Our $parse_X$ function constructs a derivation in a top-down fashion. The order of computation may also be reversed, computing parses proceeding from smaller to larger subwords. This is normally done in CYK parsing. To achieve good efficiency, CYK parsers tabulate their results for each subword on which they are called, to avoid recomputation in the case of another call on the same subword. When $|w| = n$, there are at most $O(n^2)$ subwords. If the grammar is in Chomsky Normal Form, a righthand side requires no more than $|w|$ different splits, and the CYK algorithm runs in $O(n^3)$ time and $O(n^2)$ space.

However, the idea behind CYK is not restricted to grammars in Chomsky Normal Form. A production of form $A \to BCD$, for example, will increase runtime to $O(n^4)$. Two extra, nested loops are required to iterate over all internal splits between BC and CD.

In the case of *DotPar*, since the grammar is non-ambiguous, at most one of the sets on the righthand side of $parse_S$ contains a derivation tree. For ambiguous grammars, several sets can hold multiple trees, and with productions like $A \to BC$, the numbers of parses for B and C multiply when considering all parses for A. In general with an ambiguous grammar, the number of parses grows exponentially with the length of the input string.

1.2 Stochastic CFGs

To make a rational choice between multiple parses, a CYK parser is typically equipped with a scoring scheme. Parse trees are scored and selected based on their scores on-the-fly – this combination of parsing, scoring and choice is commonly

known as dynamic programming [10]. If the scoring is based on a probabilistic model, this brings us to SCFGs.

Definition 5. A *stochastic* context-free grammar G is a CFG which associates with each production rule r, a transition probability π_r. For all $A \in V$, with $A \to \alpha_1 \mid ... \mid \alpha_k$, named $r_1, \ldots, r_k$, $\sum_{i=1}^{k} \pi_{r_i} = 1$ must hold. The *probability* $P_G(t)$ *of a derivation* (or parse tree) t is the product of the π_{r_i} for all uses of productions r_i in t. The *word probability* w assigned by grammar G is defined as $P_G(w) = \sum_{t \in T_G(w)} P_G(t)$.

Note that by this definition, $P_G(w) = 0$ if $w \notin L(G)$. $P(t)$ is sometimes called the *joint probability* of parse t and word w, to emphasize that t includes w as its string of leaves.

In Figure 1.2 we associate probabilities with the rules of grammars *DotPar*, *Pali*, and *Ali*.

The relationship between parses and their probability can be expressed in a neat way by considering a parse tree t as a formula, say $t = dot(a, pairsplit(c, end, g, end))$. By re-interpreting the production names as scoring functions

$$\begin{aligned} dot(x,s) &= \pi_{dot_x} \cdot s \\ pairsplit(x,s,y,s') &= \pi_{pairsplit_{xy}} \cdot s \cdot s' \\ end() &= \pi_{end} \end{aligned}$$

we obtain

$$P(t) = dot(a, pairsplit(c, end, g, end)) = \pi_{dot_a} \cdot (\pi_{pairsplit_{cg}} \cdot \pi_{end} \cdot \pi_{end}) = 0.002$$

under the above assignment of probabilities.

Probabilites assigned to *DotPar*

$\pi_{\text{end}} = 0.1$
$\pi_{\text{dot}} = 0.5$
$\pi_{\text{pairsplit}} = 0.4$

Probabilites assigned to *Pali*

$\pi_{\text{turn}} = 0.1$
$\pi_{\text{pair } aa} = 0.7$
$\pi_{\text{pair } bb} = 0.2$
$\pi_{\text{end}} = 0.1$
$\pi_{\text{add } a} = 0.4$
$\pi_{\text{add } b} = 0.6$

Probabilites assigned to *Ali*

$\pi_{\text{match } xx}$	$= 0.2$	for 4 choices of x
$\pi_{\text{replace } xy}$	$= 0.033$	for 12 choices of x, y
$\pi_{\text{del } x}$	$= 0.025$	for 4 choices of x
$\pi_{\text{ins } x}$	$= 0.025$	for 4 choices of x
π_{end}	$= 0.0004$	

Fig. 3 Parameters assigned to grammars *DotPar*, *Pali*, and *Ali*. With these assignments, the trees in Figure 1.1 have probabilities $\pi^4_{end} \cdot \pi^3_{dot} \cdot \pi^3_{pairsplit} = 8 \cdot 10^{-7}$, $\pi^1_{pair\ aa} \cdot \pi^2_{pair\ bb} \cdot \pi^1_{turn} \cdot \pi^1_{add\ a} \cdot \pi^2_{add\ b} \cdot \pi^1_{end} = 0.4032 \cdot 10^{-3}$, and $\pi_{match\ cc} \cdot \pi_{del\ a} \cdot \pi_{ins\ g} \cdot \pi_{rep\ tg} \cdot \pi_{end} = 1.65 \cdot 10^{-9}$, respectively.

Under mild conditions, an SCFG defines a probability distribution on $L(G)$, i.e. $\sum_{w\in L(G)} P_G(w) = 1$. Here, it is important that the grammar is clean. In a clean grammar, the sum defining $P_G(w)$ is finite, and no probability mass gets lost by entering unproductive derivations.

A probability measure defined in this way has a number of properties one should be aware of. For example, since multiplication is a commutative and associative operation, $P_G(t)$ can always be re-factored as $\pi_{r_1}^{n_1} \cdot \pi_{r_2}^{n_2} \cdot \ldots \cdot \pi_{r_k}^{n_k}$, where n_i denotes the number of times rule r_i is applied in t. Hence, all derivations which use the same rule the same number of times have the same probability, no matter of their arrangement in the derivation.

Looking more closely at *DotPar*, we find that a derivation tree with k *pairsplit*-nodes must have $k+1$ *end*-nodes. Hence, for a structure w with k base pairs and l unpaired bases, we have $P_{DotPar}(w) = \pi_{dot}^{l} \cdot (\pi_{pairsplit} \cdot \pi_{end})^k \cdot \pi_{end}$. This stochastic model simply scores the number of base pairs against the number of unpaired bases! It is insensitive to their arrangement, which makes it certainly a rather crude structure model. Larger grammars can capture more structural features, but also have more parameters and require more data to derive concrete parameters from. This balance must be carefully chosen. Grammar *DotPar* is an extreme choice: it is well-suited for expository purposes, because it is so small, but not useful for practical modeling.

Another property of this type of model is that, since all rule probabilities multiply over a derivation, and all $\pi_r < 1$, longer words tend to have small probability. In fact, for any $\varepsilon > 0$, there is a limit N such that for all "long" words with $|w| > N$, we have $P(w) < \varepsilon$. See [2] for a detailed discussion of such issues. Should we intend to study probabilities assigned to words of varying length, we should normalize scores with respect to word length. This can be achieved, for example, by dividing the probability $P_G(w)$ by the probability of generating the *sequence* w from a background distribution.

Definition 6. Three important algorithms are encountered with SCFGs:

- The most-likely-parse algorithm or "Viterbi-Algorithm": Given w, compute $p^* = max_{t\in T_G(w)} P(t)$, and also some or all t^* such that $P(t^*) = p^*$. In words: t^* is the most likely parse, and p^* is its probability.
- The "Inside-Algorithm": Given w, compute the word probability $P(w)$, as defined above as the sum over all parse trees of w.
- The "Outside Algorithm": Given $w = xyz$, compute $\sum_{t\in T_G(w)} P(t[y])$ where $t[y]$ is a derivation tree for xyz *excluding* the subtree which derives subword y.

The name "Viterbi" for the most-likely-parse algorithm is borrowed from HMM terminology. Both the Viterbi and the Inside algorithms are commonly based on a CYK parsing algorithm, equipped with a different handling of multiple parses. The Viterbi algorithm only persues the most likely parse at each point, while the Inside algorithm takes the probability sum over all parses. In the bioinformatics literature, one occasionally finds the name CYK used with the meaning of most-likely-parse. At least for the present chapter, we avoid this confusion. The outside algorithm is relevant for parameter training and uses a recursion scheme different from CYK [6].

An SCFG G can always brought into Chomsky Normal Form while preserving the probability distribution it defines on $L(G)$. Let there be a rule named r, which is not in Chomsky Normal Form, say

$$r: \quad A \to BCD \quad : \pi_r$$

We can savely replace it by two rules

$$r_1: \quad A \to BX \quad : \pi_{r_1} = \pi_r$$

$$r_2: \quad X \to CD \quad : \pi_{r_2} = 1$$

(Remember that X is a new non-terminal symbol with no other rules.) While the original rule would cause a runtime of $O(n^4)$ of the CYK parser, the transformed grammar only requires $O(n^3)$. Since this transformation is always possible[3], feel free to use the most natural form when you design an SCFG, and use Chomsky Normal Form only for the implementation.

1.3 Connecting SCFG to HMM terminology

Hidden Markov Models (HMMs) are stochastic models based on state transitions. Each (hidden) state from a state set S emits an observable symbol from alphabet $\mathscr{A}$ and enters a subsequent state. A series of transitions, also called a state path, thus emits a sequence of symbols. Emissions and transitions are made with a certain probability, and these probabilities multiply along a state path. HMMs are often depicted as state transition diagrams, or as transition probability matrices (assuming any state can transit into any state) and emission probability vectors (assuming any state can emit any $a \in \mathscr{A}$). However, the grammar view does also apply: A combination of emission and transition can be written as a rule

$$S_1 \to aS_2$$

indicating that a symbol a is emitted upon transit from state S_1 to S_2. This resembles a production rule, where states become the non-terminal symbols. In fact, this rule has a very simple form, as it holds only one non-terminal symbol on the righthand side. Grammars under this restriction are called *regular grammars* in formal language theory, and from this point of view, HMMs are SCFGs where the underlying grammar is regular. In the frequent case where any state can transit into any state and emit any symbol, albeit with different probability, the language generated by this grammar is trivially $\mathscr{A}^*$. This is why formal language terminology is not very useful with HMMs.

[3] When other types of scoring schemes are associated with G, such an efficiency improving transformation may not be possible. This is known as the yield parsing paradox in dynamic programming [10].

The generalization towards SCFGs, formulated in HMM terminology, allows each state to generate not a single successor state, but any (fixed) number of immediate successors. For example, a transition could be

$$S_1 \rightarrow aS_2bS_1cS_3d$$

with several symbols a,b,c,d emitted simultaneously and successor states S_2,S_1,S_3 created. The stochastic process branches after such a transition, and further transitions from the generated states S_1,S_2,S_3 proceed independently. The view of a state "path" breaks down, as the transitions branch off into a tree. Transition diagrams become unreadable, and transition matrices are no longer convenient. This is why the HMM terminology is not very useful with SCFGs.

Considering the HMM tradition of separating state transition and symbol emission, we note that SCFGs, per se, do not need to distinguish between emission and transition probabilities. Nor are they restricted to emit one symbol at a time. However, we can make a grammar more HMM-like by rewriting it a bit (without changing the language). We can transform it such that some non-terminal symbols produce only terminal symbols – the "emissions", and others only produce nonterminal symbols – the "transitions", and re-factor the probabilities accordingly. The above rules

$$S_1 \rightarrow aS_2 \text{ for } a \in \mathscr{A}$$

would be rewritten to

$$S_1 \rightarrow AS_2$$

$$A \rightarrow a \mid b \mid c \mid \ldots$$

and the original probabilities re-assigned as a transition probability with the first rule, and as emission probabilities with the second. But note that any probability for two (or more) symbols emitted jointly as in $S \rightarrow aSb$ must remain associated with the production that does the joint emission.

The HMM counterparts of the most-likely-parse, Inside and Outside algorithms are the Viterbi, Forward and Backward algorithms in HMM terminology. In fact, when the grammar underlying our SCFG is a regular grammar, practically the same computation is performed.

1.4 Semantics of stochastic models

So far, we have discussed SCFGs purely as a formal device. An SCFG G assigns a probability to a $P_G(w)$ to a word $w \in L(G)$, and a probability $P_G(t)$ to each of its derivations t. When it comes to using stochastic models for the analysis of real-world objects or phenomena, it must be clearly specified what the computed probabilities mean in reality. The stochastic model must be given a semantics. We will do this twice in the second part of this chapter, where we use SCFGs to derive a most likely structure for a single RNA sequence, and to compute the likelihood of

the sequence belonging to a particular RNA family for which a SCFG family model has been created.

2 Analyzing RNA secondary structure with SCFGs

The use of SCFGs for RNA structure analysis has two forms: An SCFG can be used to assign a structure to an RNA sequence, and it can be used to build a model of an RNA sequence family with a conserved structure. The present section introduces both kinds of use, and then links to the applied chapters of this book where these scenarios are treated in detail.

2.1 SCFGs modeling secondary structure of a single RNA sequence

SCFG parses indicating RNA secondary structure

For secondary structure prediction, we use grammars which generate RNA *sequences*, and whose parses designate potential secondary *structures* for these sequences. Let $F(w)$ be the folding space of sequence w, i.e. the set of all structures w can fold into according to the chosen rules for base pairing, and let there be a semantic mapping $\mu : T_G(w) \to F(w)$, which relates parses to structures. Most desirable, this mapping is bijective: Surjectivity of μ ensures that the CYK parser actually evaluates all possible foldings. Injectivity of μ guarantees that the most likely parse also denotes the most likely structure under the given parameters.

Recall grammar *DotPar*. It generates dot-bracket strings, each of which denotes an RNA secondary structure, independent of a particular sequence of bases. As we have seen, each dot-bracket string has a unique parse tree. If we replace the terminal alphabet and make the grammar generate RNA sequences instead, each parse with this grammar will indicate a secondary structure for that sequence. In this way, we obtain the grammar *MiniRNA* – see Figure 2.1.

Grammar *MiniRNA*.
$\mathscr{A} = \{a,c,g,u\}$, $V = \{S\}$, axiom is S.

production rule		rule name
S	$\to \varepsilon$	*end*
	$\mid xS$	dot_x for $x \in \mathscr{A}$
	$\mid xSyS$	$pairsplit_{xy}$ for $x,y \in \mathscr{A}$

Fig. 4 Grammar *MiniRNA* is modeled after grammar *DotPar*. Note that rule *pairsplit* generates arbitrary pairs of bases. Stochastic parameters trained from real data will associate high probabilities with canonical base pairs (a-u, c-g, g-u), but allow non-standard base pairs with a small probability.

Grammar *MinRNA* is shaped after *DotPar*, using isomorphic rules, but deriving an RNA sequence rather than a dot-bracked string. For a parse tree $t \in T_{MiniRNA}(w)$,

let $\hat{t}$ be the isomorphic tree of grammar *Dot par*. $\hat{t}$ derives a dot bracket string s, and this s is the structure assigned to w by t – we define $\mu(t) = s$. Since we already know that any dot-bracket string s has only one parse tree in *DotPar*, in our case $\hat{t}$, the parse tree t is the only parse tree of grammar *miniRNA* with $\mu(t) = s$. This means that with *MiniRNA*, the most likely parse tree t^* indicates the most likely structure.

Semantic (non-)ambiguity

What if μ is not injective, i.e. several parse trees indicate the same secondary structure? This situation is called *semantic ambiguity*. This is a subtle pitfall with SCFGs, which, when ignored, may render parts of your SCFG work invalid.

Let us cast this in general terms, following [8].

Definition 7. Let G be an SCFG and $\mathscr{M}$ a set of objects of interest, called meanings. A *semantics* for G is a mapping $\mu : \{T_G(w) \mid w \in L(G)\} \to \mathscr{M}$. The probability of object $m \in M$ given w is defined as $P(m) = \sum\{P(t) \mid t \in T_G(w), \mu(t) = m\}$. G is *semantically ambiguous*, if there is a $w \in L(G)$ with $t, t' \in T_G(w)$ such that $t \neq t'$ and $\mu(t) = \mu(t')$.

With semantic ambiguity, the probability of any object m is distributed over all the parses t where $\mu(t) = m$. When interested in the most likely object m^*, it does not help to compute the most likely parse t^*, since generally, $\mu(t^*) \neq m^*$. $P(m^*) > P(t^*)$ can occur, with several parses t' contributing to $P(m^*)$, all having $p(t') < p(t^*)$. In fact, mistaking $\mu(t^*)$ for m^* can be strongly misleading, as has been evaluated empirically in [5]. The Viterbi algorithm is not meaningful with a grammar that is semantically ambiguous.

Semantic ambiguity is discussed in HMM literature under the name "path labeling problem", where a path labeling corresponds to our meaning function μ, and maps the state paths of an HMM to more abstract objects of interest. In [4] it was shown that the problem of computing the optimal path labeling (or in our terms: the optimal meaning) is NP-hard in general. As SCFGs properly subsume HMMs, this also applies to SCFGs, *in general*. It has not been shown whether this holds *in particular* for SCFGs with μ mapping parses to structures, but can be taken as a warning that computing $P(m^*)$ from a semantically ambiguous SCFG may be intractable. Therefore, it seems advisable to avoid semantic ambiguity altogether.

As with syntactic grammar ambiguity, there is no algorithm which, given a grammar G and a meaning function μ, can decide whether G is semantically ambiguous with respect to μ. This problem was shown to be undecidable in [14]. However, we shall point out a pragmatic approach to semantic ambiguity checking below.

Grammar design

Grammar *MiniRNA* cannot be expected to be a very useful modeling device. Remember what we observed about *DotPar*: it merely weights base pairs against unpaired bases. *MiniRNA* has different parameters for each base or base pair, but else has little extra distinctive power. For example, all unpaired residues are treated alike, no matter whether they reside in a hairpin loop, in a bulge, or in the external loop. All C-G pairs contribute the same score, independent of their structural context, and so on. Let us create a grammar which gives us more control.

Grammar *RNAFeatures* is designed to explicitly designate the different structural features which humans refer to when speaking about secondary structure. See Figure 2.1.

Grammar *RNAFeatures*.
$\mathscr{A} = \{a, c, g, u\}$,
$V = \{Externalloop, \ldots, MLComponents\}$, axiom is *ExternalLoop*.

production	rule		rule name	
ExternalLoop	$\rightarrow$	ε	el_1	
	$\mid$	x ExternalLoop	$el_{2,x}$	for $x \in \mathscr{A}$
	$\mid$	Stack ExternalLoop	el_3	
Stack	$\rightarrow$	x Stack y	$st_{1,xy}$	for $x, y \in \mathscr{A}$
	$\mid$	x Weak y	$st_{2,xy}$	for $x, y \in \mathscr{A}$
Weak	$\rightarrow$	HairpinLoop	wk_1	
	$\mid$	InternalLoop	wk_2	
	$\mid$	BulgeLeft	wk_3	
	$\mid$	BulgeRight	wk_4	
	$\mid$	MultiLoop	wk_5	
HairpinLoop	$\rightarrow$	x SingleStrand y	hl_{xy}	for $x, y \in \mathscr{A}$
InternalLoop	$\rightarrow$	x SingleStrand Stack SingleStrand y	il_{xy}	for $x, y \in \mathscr{A}$
BulgeLeft	$\rightarrow$	x SingleStrand Stack y	bl_{xy}	for $x, y \in \mathscr{A}$
BulgeRight	$\rightarrow$	x Stack SingleStrand y	br_{xy}	for $x, y \in \mathscr{A}$
MultiLoops	$\rightarrow$	x ML_Component ML_Components y	ml_{xy}	for $x, y \in \mathscr{A}$
ML_Component	$\rightarrow$	SingleStrand Stack	co_1	
	$\mid$	Stack	co_2	
ML_Components	$\rightarrow$	ML_Component ML_Components	cs_1	
	$\mid$	ML_Component SingleStrand	cs_2	
	$\mid$	ML_Component	cs_3	
SingleStrand	$\rightarrow$	x SingleStrand	$ss_{1,x}$	for $x \in \mathscr{A}$
	$\mid$	x	$ss_{2,x}$	for $x \in \mathscr{A}$

Fig. 5 Grammar *RNAfeatures* which explicitly identifies structural components.

Grammar *RNAFeatures* uses more non-terminal symbols and rules than *MiniRNA* to explicitly designate multiloops, bulges, internal loops, and so on. If a structure contains two multiloops, it is because the rule named multiloop is used twice, and the same holds for internal loops, bulges, and hairpin loops. This allows the respective probabilities to reflect the statistics of structural features. *RNAFeatures* also enforces the convention that a structure should not have "lonely pairs" (i.e. base pairs not stacked onto an adjacent pair). This is achieved by the use of two non-terminal symbols: *Weak* derives substructures "weakly" closed by a single base pair, while *Stack* derives substructures closed by two or more base pairs. Since *Weak* substructures can only be embedded in larger parse trees via the rule $Stack \rightarrow x\ Weak\ y$, there is no way that a structure derived from the axiom $Externalloop$ can have an isolated base pair. (If you prefer to allow lonely pairs, just identify *Weak* and *Stack*, and remove the redundant production rule that results from this merge.)

Another interesting point is the handling of multiloops. First of all, substructures inside a multiloop are produced by different rules than substructures in the external loop. All unpaired bases in the external loop are produced via rules $el_{2,x}$, while all others stem from rules $ss_{1,x}$ abd $ss_{2,x}$. This allows to assign independent probabilities to them. Second, care has been taken that inside the multiloop, there are at least two closed substructures. This is important, since a multiloop with a single stem inside would rather be considered an internal loop, which is already modeled by the appropriate rule. Ignoring this fact would make the grammar semantically ambiguous. Third, the grammar takes care not to derive two adjacent *SingleStrand* non-terminal symbols. This would happen, for example, if the rule for *ML_Component* was written in the more natural, symmetric way, as in $ML\_Component \rightarrow SingleStrand\ Stack\ Singlestrand$. With such a rule, two helices branching from a multiloop would lead to a derivation via $SingleStrand\ Stack\ Singlestrand\ SingleStrand\ Stack\ Singlestrand$, and unpaired bases between two stacks could be ambiguously derived from the two adjacent *SingleStrand* non-terminal symbols.

We leave it to the reader to define the semantic mapping μ and show that this grammar is semantically non-ambiguous.

Grammar design trade-offs

Let us consider the trade-offs when designing a (stochastic) grammar modeling RNA structure. Grammar *MiniRNA* has only three rules and only 20 parameters to be trained from the data – π_{end}, 4 for π_{dot_x}, and 16 for $\pi_{pairsplit_{xy}}$, wich makes 21 - 1, since they must sum up to 1. In particular, the *pairsplit* rule combines both the generation of a base pair and a potential branch in the structure. Since continuous stacking pairs are much more frequent than branching structures, this is a unfortunate coupling of situations. They should rather be modeled independently. Grammar *MiniRNA* has been reported to have "abysmal" practical performance [5] in modeling RNA structure.

Grammar *RNAFeatures*, on the other hand, has more than 120 parameters. Each structural feature is governed by an extra parameter for its special rule. If structures in our training data have lots of bulges, but few internal loops, the grammar will be trained to reflect this. Even the base pairs enclosing an internal, bulge or multiloop are assigned separate parameters, which may come out different in training from data. In practice, one may want to tie some of these parameters to each other, to avoid overfitting.

Grammars that constitute a compromise between *MiniRNA* and *RNAFeatures* are used, for example, in *Pfold* and *Infernal*.

A stochastic model will always reflect the statistical properties of the training data set, including structural feature frequency, sequence length, and base composition. There are two sides to this coin:

- A general model requires a large data set, "typical" of all RNA structures. This is hard to achieve. An extreme case is a grammar presented in [13], which reaches the sophistication of the thermodynamic parameter space. This grammar requires 17 non-terminal symbols and 41 rules.
- If you use training data from a specific RNA family, the model can adapt to family properties to a certain extent, again in terms of base composition, sequence length, and frequency of structural features. It will not be able to capture the specific arrangement of structural components. This is why we must move on from general RNA folding grammars to grammars specialized to specific structures for RNA family modeling.

Avoiding semantic ambiguity

Most likely, when working with SCFGs, you will use one of the established tools, which have been carefully designed, and the issue of semantic ambiguity does not arise. Should you ever design your own grammar, here is some guidance for avoiding semantic ambiguity.

- Stay away from rules like $S \to S\ S$, as with this rule, distinguishing multiple occurences of S by superscripts, $S^1S^2S^3$ can be derived in two ways, $S \to S^1S \to S^1S^2S^3$ and $S \to SS^3 \to S^1S^2S^3$.
- Use two non-terminal symbols instead to generate sequences of substructures, e.g. $T \to S\ T \mid S$.
- Finally, do not use rules like $S \to aS \mid Sa \mid T$, as the phrase aTa has two derivations.

In all of these cases, the competing derivations make no difference for the assigned structure under the semantic mapping μ, and lead to semantic ambiguity.

Still, avoiding ambiguity can be subtle, and can lead to grammars with more rules and hence, more parameters to train. A simple check against semantic ambiguity is the following. Modify your scoring functions such that they score each candidate by 1, and sum up the scores as in the Inside algorithm. This gives you the size of your grammar's search space, $|T_G(w)|$, for any test sequence w. Compare this to

$|T_{MiniRNA}(w)|$, which should give the same result. (Instead of MiniRNA, you can use any other grammar whose semantic non-ambiguity you trust in, but make sure it uses the same conventions on lonely pairs, minimal hairpin loop sizes etc. as your grammar G does. The number of structures is very large (use long integers!), and if the numbers coincide, this is strong evidence that the search space of G does not have redundant candidates. Still, it is only a test.

In spite of the general undecidability of semantic ambiguity checking, there is a quite powerful method to do it for grammars modeling RNA structure [9]: Given grammar G, replace each rule generating unpaired bases $x \in \{a,c,g,u\}$ by a single rule generating a '.' instead, and each rule generating possible base pairs by a single rule generating '(' and ')' instead. This turns the grammar into a grammar $\hat{G}$ which derives dot-bracket strings, i.e. $L(\hat{G}) = L(DotPar)$. In fact, $\widehat{MiniRNA} = DotPar$. Then, submit the grammar $\hat{G}$ to a syntactic ambiguity checker for context free grammars, such as the ACLA server described in [3]. If it proves that $\hat{G}$ is syntactically non-ambiguous, then your grammar G is semantically non-ambiguous.

Implementing your SCFG

When it comes to implementing your own SCFG, you must produce code for the Viterbi and/or Inside algorithms. The chapter by Sukusd et al. in this book provides some guidance by specifying low-level pseudo-code for these algorithms, which needs to be adapted to your grammar. However, hand-programming dynamic programming recurrences is error-prone and their debugging is tedious. The recent Bellman's GAP programming system [**?**] supports dynamic programming over sequence data in general, which subsumes the implementation of SCFGs. It allows you to specify a grammar and one or more scoring schemes separately, and generates efficient code from these declarative constituents. This by-passes all subscript-fiddling on your side, and the resulting programs are easy to modify when your ideas about the grammar or the scoring scheme evolve.

2.2 SCFGs modeling structural RNA families

A structural family of RNAs is defined as a set of RNA sequences which fold into a consensus structure, either with good free energy in the thermodynamic model, or with high probability in a stochastic model. However, the consensus structure need not be realized exactly by a sequence in order to fit the model. The sequence may have more or fewer residues than the consensus, and it should achieve a good number of the base pairs in the consensus, but not necessarily all of them.

Definition 8. A *family model grammar* is an SCFG whose parse trees encode alignments of a *query* sequence to a *consensus* structure, allowing for insertions and deletions.

Parameters for a family model grammar M are typically derived from a set of aligned "seed" sequences, and an explicitly given consensus structure. As before, M assigns a probability $P_M(q)$ to any query sequence q. When this probability exceeds a model-specific threshold, q is accepted as a member of the family modeled by M.

To construct a family model M, we can build on our grammars introduced earlier. Since *DotPar* encodes structures, *MiniRNA* assigns structures to sequences, and *Ali* models sequence alignments, a suitable combination of these three grammars will give an SCFG that serves as a family model. We will describe the construction of the family model grammar using a small example. Let our consensus structure be $c =$ `" . ( ( . ) ) . ( ) "`, shown as a *DotPar* tree in Figure 1.1.

A family model does not allow the query to fold into arbitrary structures. Only structures are allowed which are formed by a subset of the base pairs in the consensus. Hence, the model specializes grammar *MiniRNA* with respect to c by using several copies of its rules and non-terminal symbols, indexed by the positions in the consensus to which they correspond. Let us number the positions in c as

$$\cdot_1(_2(_3\cdot_4)_5)_6\cdot_7(_8)_9$$

Specializing *MiniRNA* to c will yield, as an intermediate step, a grammar *StrictConsensus* (see Figure 2.2), which folds each sequence *strictly* into the consensus, with no base pairs omitted or bases inserted.. We proceed as follows:

Grammar *StrictConsensus*.
$\mathscr{A} = \{a,c,g,u\}$, $V = \{S_1,...,S_{10}\}$, axiom is S_1.

production rule		rule name (subscripts omitted)
S_1	$\to xS_2$	dot_x for $x \in \mathscr{A}$
S_2	$\to xS_3yS_7$	$pairsplit_{xy}$ for $x,y \in \mathscr{A}$
S_3	$\to xS_4yS_6$	$pairsplit_{xy}$ for $x,y \in \mathscr{A}$
S_4	$\to xS_5$	dot_x for $x \in \mathscr{A}$
S_5	$\to \varepsilon$	*end*
S_6	$\to \varepsilon$	*end*
S_7	$\to xS_8$	dot_x for $x \in \mathscr{A}$
S_8	$\to xS_9yS_{10}$	$pairsplit_{xy}$ for $x,y \in \mathscr{A}$
S_9	$\to \varepsilon$	*end*
S_{10}	$\to \varepsilon$	*end*

Fig. 6 Grammar *StrictConsensus* obtained specializing *MiniRNA* to $c =$ `" . ( ( . ) ) . ( ) "`.

The length of c is 9. We make 10 copies of *MiniRNA*, renaming the non-terminal symbol S into $S_1,\ldots,S_{10}$. For each position, we retain the rule alternative required to derive c in that place, and delete the other two alternatives. We start with $S_1 \to xS_2$, since the position 1 in c is unpaired. We continue with $s_2 \to xS_3yS_7$, as x and y correspond to the matching brackets at consensus positions 2 and 6. Similarly, we get $S_3 \to xS_4yS_6$. For S_6, we create the rule $S_6 \to \varepsilon$, since the base at position 6 is already generated as y by S_2. And so on.

The grammar *StrictConsensus* is most rigid. It will parse any query sequence of length $|c|$ into the consensus structure, and find no parse for any other sequence. What we want to compute are alignments like shown in Figure 2.2.

```
.((.)).(--)      .((.)).-()      .((.)).-()
-cguc-aucca      c-uca-cuua      cu-c-acuua
    (1)              (2)             (3)
 cgucaucca        cucacuua        cucacuua
 .(.).(..)        .(.)..()        .(.)..()
```

Fig. 7 Top row: Three alignments of a query sequence to the family consensus structure. Bottom row: The structure assigned to the query by the respective alignment. Due to deletions and insertions in the alignments, all assigned structures in (1) - (3) are different from the consensus.

In order to allow for deletions and insertions with respect to the consensus, let us incorporate what we have learnt from grammar *Ali*. *Ali* aligns two sequences – here we want to align the query sequence to the consensus. As the consensus is already encoded in the grammar *StrictConsensus*, we do not have to treat the consensus as a second sequence. But we have to allow (1) for residues in the query, which are not aligned to residues in the model, and (2) for positions in the consensus structure, which are not matched by bases in the query. Point (1) is handled by adding a rule alternative $S_i \rightarrow xS_i$ for $x \in \mathscr{A}$. Point (2) requires, for each rule alternative which generates query residues, an alternative to make the same transitions without generating a symbol. For $S_i \rightarrow xS_j$, we add the alternative $S_i \rightarrow S_j$ – without the x, which means that no symbol in the query corresponds to position i in the consensus. For $S_i \rightarrow xS_jyS_k$, we add the alternatives $S_i \rightarrow S_jyS_k \mid xS_jS_k \mid S_jS_k$, which allow the left, the right, and both partners of a consensus base pair to be missing in the query. This completes our construction of the family model grammar; the result is shown in Figure 2.2.

Grammar *FamilyModel*.
$x,y \in \mathscr{A} = \{a,c,g,u\}$, $V = \{S_1,...,S_{10}\}$, axiom is S_1.

production rule			rule names omitted			
S_1	$\rightarrow xS_1$	$\mid xS_2$	$\mid S_2$			
S_2	$\rightarrow xS_2$	$\mid xS_3yS_7$	$\mid S_3yS_7$	$\mid xS_3S_7$	$\mid S_3S_7$	
S_3	$\rightarrow xS_3$	$\mid xS_4yS_6$	$\mid S_4yS_6$	$\mid xS_4S_6$	$\mid S_4S_6$	
S_4	$\rightarrow xS_4$	$\mid xS_5$	$\mid S_5$			
S_5	$\rightarrow xS_5$	$\mid \varepsilon$				
S_6	$\rightarrow xS_6$	$\mid \varepsilon$				
S_7	$\rightarrow xS_7$	$\mid xS_8$	$\mid S_8$			
S_8	$\rightarrow xS_8$	$\mid xS_9yS_{10}$	$\mid S_9yS_{10}$	$\mid xS_9S_{10}$	$\mid S_9S_{10}$	
S_9	$\rightarrow xS_9$	$\mid \varepsilon$				
S_{10}	$\rightarrow xS_{10}$	$\mid \varepsilon$				

Fig. 8 Grammar *FamilyModel* obtained by extending *StrictConsensus* with rules for insertions and deletions.

We have used grammar *MiniRNA* as the prototype grammar. It was specialized with respect to a consensus structure, and then extended to provide for query/consensus alignments. It can be shown that, when proceeding in this systematic fashion from a prototype grammar which is semantically non-ambiguous, this property also holds for the derived family model grammar: Each parse tree $t \in T_{FamiliModel}(w)$ uniquely encodes a query/consensus alignment. It is not strictly necessary to proceed this way: *Infernal*, for example, starts from an ambiguous prototype grammar, but still ensures non-ambiguity of the family model by a more refined generation process.

What can we compute once we trained the parameters for our family model grammar? The Viterbi algorithm computes the most likely alignment of the query to the model. This alignment can be used, for example, to extend the seed sequence alignment, from which the model was generated, by adding a high scoring query as a bona-fide family member. The Inside algorithm computes the overall probability of the query with respect to the model, $\sum\{P(t)|t \in T_{FamilyModel}(w)\}$.

Another interesting piece of information would be (the probability of) the most likely structure assigned to the query by a family model grammar M. All base pairs produced by the rules $S_i \rightarrow xS_jyS_k$ make up the secondary structure assigned to w, irrespective how gaps and unpaired bases are placed. Here, the semantic mapping μ maps query/consensus alignments to structures of the query. In Figure 2.2, the same structure is assigned to the query in case (2) and (3), by *different* alignments to the consensus. Hence, the most likely structure would be the query structure s^* which maximizes $\sum\{P_M(t)|\mu(t) = s\}$. However, at the point of this writing, it is not known how to compute this information efficiently.

3 Further reading

Applications of stochastic context free grammars are treated in three further chapters of this book:

- An introduction to RNA databases, by Marc Hoeppner and Paul Gardner,
- SCFGs for RNA structure prediciton, by Zsuzsanna Sukosd, Ebbe S. Andersen, Paula Tataru, and Rune Lyngsoe,
- Annotating ncRNAs in genomes with *Infernal*, by Eric Nawrocki.

These chapters cover the most important uses of SCFGs in RNA structure analysis at the time of this writing. For further reading, please consult the literature given therein.

Acknowledgement Thanks go to Jan Reinkensmeier for a careful reading of this manuscript.

References

[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, NJ, 1973. I and II.

[2] T L Booth and R A Thompson. Applying probability measures to abstract languages. *IEEE transactions on Computers*, 1973.

[3] Claus Braband, Robert Giegerich, and Anders Møller. Analyzing Ambiguity of Context-Free Grammars. *Science of Computer Programming*, 75(3):176–191, March 2010. Earlier version in Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07, Springer-Verlag LNCS vol. 4783.

[4] B Brejová, DG Brown, and T Vinař. The most probable annotation problem in HMMs and its application to bioinformatics. *Journal of Computer and System Sciences*, 73(7):1060–1077, March 2007.

[5] R D Dowell and S R Eddy. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, 5:71–71, Jun 2004.

[6] R Durbin, S Eddy, A Krogh, and G Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 2006 edition, 1998.

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

[8] R Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2000.

[9] R Giegerich and C Höner zu Siederdissen. Semantics and ambiguity of stochastic rna family models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(2):499–516, 2011.

[10] R Giegerich, C Meyer, and P Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, June 2004.

[11] J E Hopcroft and J D Ullman. *Formal languages and their relation to automata*. Addison-Wesely, 1969.

[12] Baker J K. Trainable grammars for speech recognition. *Journal of the Acoustic Society of America*, pages 54–550, 1979.

[13] M Nebel and A Scheid. Analysis of the free energy in a stochastic RNA secondary structure model. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, to appear.

[14] J Reeder, P Steffen, and R Giegerich. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6(153), 2005.

[15] Y Sakakibara, M Brown, R Hughey, I S Mian, K Sjölander, R C Underwood, and D Haussler. Stochastic context-free grammars for tRNA modeling. *Nucleic Acids Res*, 22(23):5112–5120, Nov 1994.