Contents

Chapter 1

Overview

This work presents original contributions to the field of biological RNA modeling using stochastic context-free grammars (SCFGs). These original contributions include the application of SCFGs to RNA modeling [Sakakibara et al., 1994a], a theory of SCFG constraints that allows the application of large grammars to large strings, the application of SCFGs to small subunit ribosomal RNA modeling, an SCFG approximation to the modeling of RNA pseudoknotted structures [Brown and Wilson, 1995], the extension of Fisher information feature vectors in a support vector machine classification framework to SCFGs, as well as a suite of software tools that perform RNA modeling using SCFGs: RNACAD.

This work stems from my earlier work in biological sequence modeling using hidden Markov models(HMMs) [Krogh et al., 1994a]. HMMs probabilistically encode the information contained in a multiple alignment in which interactions between multiple alignment columns are not modeled. HMMs model the evolutionary processes of mutation, insertion, and deletion in their state structure. HMMs correspond to regular grammars and are not a powerful enough language model to capture the language of RNA basepairs that is balanced and complementary. The more powerful language model of context-free grammars are exploited in their probabilistic form, SCFGs, to model the basepairing structure of RNA. My previous HMM work also included parameter estimation from small sample sizes using a Bayesian prior of mixtures of Dirichlets [Sjölander et al., 1996].

The first application of SCFGs to RNA modeling was t-RNA modeling [Sakakibara et al., 1994a]. The state structure of SCFGs in this work is an extension of profile HMM state structures. Like HMMs, SCFGs encode the mutation, insertion, and deletion operations in their state structure but are generalized to a tree of pairwise interactions so that basepair interactions of RNA molecules can be modeled. SCFG modeling requires great computational burdens in order to compute solutions. The algorithms for SCFGs have a running time of $O(ML^3)$ and storage requirements of $O(ML^2)$ where M is the number of grammar nonterminals (states) and L is the length of the input string. For molecules such as small subunit ribosomal RNA, which has an average length of about 2000, applying full SCFG modeling requires at least 16 billion bytes of storage. This is not feasible for today's computers.

I introduce a system of constraints to allow SCFGs to be applied to large RNA structures. The system is based on maximum posterior decodes using HMMs and consists of constraint generation and propagation. With this constraint system, computational resources for large problems are reduced by a factor of order 10,000.

The introduction of constraints allows SCFGs to be applied to small subunit ribosomal RNA (SSU rRNA) modeling. SSU rRNA is an important molecule that is used in phylogenetic tree construction, antibiotic drug design, and large scale crystallographic projects in the search for the ribosome's three dimensional high resolution crystal structure. I show that SCFGs can automatically generate multiple alignments of quality close to hand-edited ones.

Although SCFGs are useful in RNA modeling they are unable to model RNA pseudoknot structures. A pseudoknot is a basepair interaction that is not well-nested with respect to other basepairs. Just as HMMs are unable to represent basepairing interactions, SCFGs are unable to correctly represent RNA pseudoknotted structures. I introduced an approximation using two SCFGs to model a single pseudoknot to help model an *in vitro* selected RNA structure that binds biotin [Brown and Wilson, 1995].

Probabilistic generative models such as SCFGs can be used in a classification framework to perform tasks such as database search. Following previous work in the domain of microarray gene expression classification using support vector machines [Brown et al., 1999] and other work in the domain of protein super-family classification using Fisher information vector features from hidden Markov models [Jaakkola et al., 1999], I introduce a procedure for constructing SCFG feature vectors and using them in a support vector machine classification framework in order to perform phylogenetic classification.

The RNA modeling software package, RNACAD, is introduced. This set of software tools allows SCFGs to be applied to RNA modeling.

This work is given in 6 Chapters including this one. Each chapter is designed to be self-contained and readable independently of the other chapters.

Chapter 2 **Biosequence Analysis** gives an introduction to the fundamental ideas of biosequence analysis. This section serves gives terminology and key concepts in the areas of biology, sequence similarity, machine learning, and formal language but does not introduce any original ideas.

Chapter 3 **Efficient Biological Sequence Modeling Using Stochastic Context-Free Grammars With Constraints** gives the theory of applying SCFGs to RNA modeling including the relation between SCFGs and hidden Markov models, how SCFGs are applied to RNA modeling, the parsing problem, a theory of SCFG constraints, and efficiency results applying SCFGs to small subunit ribosomal RNA.

Chapter 4 **The RNACAD Stochastic Context-Free Grammar Modeling System** presents the RNACAD software modeling suite. It gives a concrete overview of the mechanics of applying SCFGs to RNA modeling. It presents data formats of the RNACAD system for concreteness.

Chapter 5 **Small Subunit Ribosomal RNA Modeling Using Stochastic Context-Free Grammars** applies SCFG modeling techniques to a important large RNA molecule, small subunit ribosomal RNA(SSU rRNA). The biological importance of SSU rRNA is discussed, including its importance to phylogenetic tree construction. SCFGs and several competing methods are applied to the problem of creating a multiple alignment of SSU rRNA. It is shown that SCFGs have the highest accuracy, close to the level of hand-edited alignments. Results of the SCFG model are discussed. A method of performing phylogenetic discrimination based on SSU rRNA sequences using support vector machines and SCFG feature vectors is introduced.

Chapter 6 **Biotin Binding RNA: Pseudoknot Modeling Using Stochastic Context-Free Grammars** presents a method of modeling RNA pseudoknotted structures using a approximation involving two SCFGs. RNA pseudoknots involve sequence structures that are more complex than allowed by SCFG modeling. The approximation is applied to an *in vitro* selected RNA molecule that binds biotin.

Appendix A **RNACAD Programs** gives manual pages for programs in the RNACAD modeling suite.

The field of RNA research is very exciting for both basic research and application. The SCFG modeling paradigm presented here has many strong theoretical foundations and extends the theory in new directions. The application of SCFGs also presents new directions for their intended problem domain, RNA structure modeling. The following chapters report on progress made in these areas.

Chapter 2

Biosequence Analysis

High-throughput sequencing efforts have produced entire genomic sequences for several organisms. This has been made possible by both technological innovation and scientific realization that genomic sequencing is fundamental to the study of life on earth. Industrial high-throughput sequencing centers and massive funding initiatives by both government and private organizations have finally allowed biology to study entire organisms not simply as black boxes that respond to certain stimuli but rather as complex machines composed of millions of interacting parts in which the identity of every single part is known through the genomic sequence. Because of these technologies science can start to apply engineering methods to the study of biology. Unlike physics and chemistry that study systems having underlying truths that do not change over time, biology is an enormously complex system that has evolved over time through the process of evolution. This makes a mathematical engineering characterization of biology a dauntingly difficult and exciting task.

A discussion of biosequence analysis requires background in several key areas. First is an introduction to biology. While most problems in biosequence analysis can be mathematically abstracted away from their biological origins, presenting a mathematical solution that does not relate back to the underlying biological problem will not have much practical utility. Second is an introduction to the mathematical concepts frequently used in algorithms for biosequence analysis. These include dynamic programming, probability theory, and formal language theory.

This chapter is intended to present fundamental ideas and terminology. No original work is presented here.

2.1 Biology and the Sequencing Revolution

2.1.1 DNA

DNA is the primary information carrier in biological systems. DNA sequencing has become one of most studied areas of science.

DNA is composed of four nucleic acids: Adenine, Guanine, Cytosine, and Thymine [Watson et al., 1987]. Each nucleic acid has about 50 atoms and consists of a 5 carbon sugar with a phosphate group located at the 5' carbon of the sugar and a flat aromatic molecule located at the 1' carbon of the sugar. The flat aromatic molecule can either be a double-ringed purine (A,G) or a single-ringed pyrimidine (C,T). Nucleic acids are strung together by phosphodiester linkages or covalent bonds linking the 3' hydroxyl group of one nucleic acid to the phosphate group at the 5' carbon of the sugar of the next nucleic acid. Thus DNA sequences

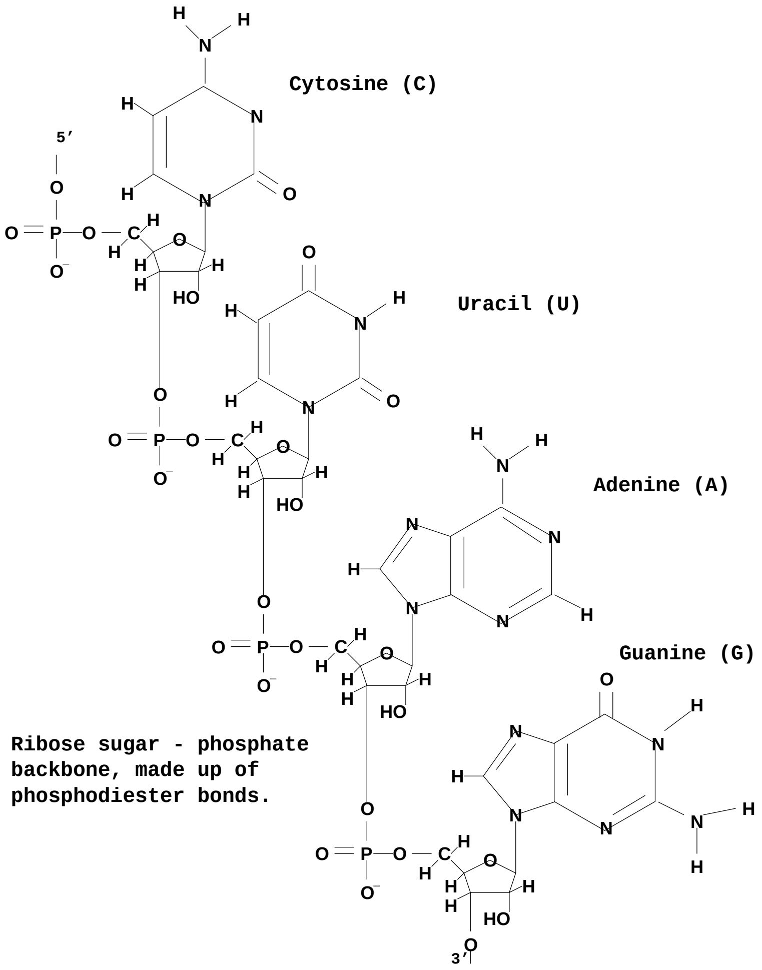


Figure 2.1: The chemical nature of an RNA chain CUAG. The sugars and phosphate groups are to the left.

form a linear chain and have a direction specified by the 5' and 3' ends. Most DNA processing occurs in a linear manner starting from the 5' nucleic acid and continues to the 3' end.

Nucleic acids can interact with each other by forming hydrogen bonds. DNA usually occurs in nature as a double stranded helix in one of several conformations in which the one strand interacts with another complementary strand by forming Watson-Crick basepair interactions (A pairs with T and G pairs with C).

2.1.2 RNA

RNA is very similar to DNA. Chemically there is a slight modification to the sugar and the nucleic acid base Thymine is replaced with a chemically similar base Uracil. Like DNA, RNA forms linear chains. See Figure 2.1 for the chemical nature of an RNA chain.

The exact sequence of bases in a RNA molecule determines its function. It might code for a protein (through mRNA), make up part of the ribosome (rRNA), be a t-RNA, or bind to a complex such as biotin. RNA is one of nature's most complex machines. It can carry information in the sequence of bases and it can

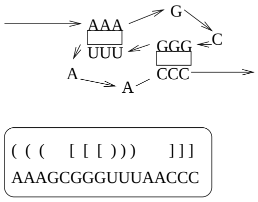


Figure 2.2: An RNA Pseudoknot. The basepairs formed by the A-U helix and the G-C helix are not well-nested. Bases within the hairpin loop of each helix interact with bases that are outside the loop. Abstractly this pairing is represented by "((([[[)))]]]" where "()" and "[]" represent two types of basepair interactions. A well-nested interaction is "((([[[]]])))".

perform certain functions based on how it folds in space, which is dictated by the sequence of bases.

When RNA folds in space it can interact with itself. Because of the chemical nature of the bases, basepairs can form. A basepair is a state in which several hydrogen bonds form between two bases. Watson and Crick showed that the basepairing nature of DNA allowed genetic information to be carried accurately[Watson and Crick, 1953]. The strength of a basepairs is not as strong as the covalent bonds stringing the sequence together and therefore the ordering of the bases in the sequence remains invariant regardless of the basepairs that form. The most common basepair interactions are Watson-Crick (G pairs with C and A with U). Wobble pairs (G pairs with U) also occur in nature. The secondary structure of RNA is usually characterized by specifying which bases interact. When basepairs form, the sequence is divided into helixes, which are the basepairing regions, and loops which are the non-basepairing regions.

A RNA pseudoknot is formed when bases outside a hairpin structure pair with bases within the hairpin loop to create a second stem and loop structure [Chen et al., 1992]. See Figure 2.2. Pseudoknotted RNA structures occur in virtually all classes of RNA and are involved in a number of important functions [ten Dam et al., 1992].

2.1.3 Database Search

The wealth of sequence data has prompted the use of fast and efficient algorithms for searching through it. Database search involves finding new sequences in a database that are by some measure "similar" to a model that represents what is being sought. Sometimes this model is a single sequence and a simple model of evolution as is the case with the popular tool, BLAST. The database search then tries to find other sequences that are "close" to the query sequence. Other models can involve a set of sequences and a probabilistic model of them or an abstracted model of physics and chemistry.

At the heart of database search is discrimination, the task of classifying a given sequence as belonging to a particular class or not. To decide whether a sequence in a database contains a t-RNA molecule, for example, one must describe the class of all strings that contain a t-RNA molecule and the class of all strings that do not contain a t-RNA molecule. Then, given a database sequence, one can check which class, the t-RNA-containing class or the non-t-RNA-containing class, better describes the sequence, and answer whether the sequence contains t-RNA. Probabilistic machine learning and Bayes decision theory address this issue of discrimination in a formal manner [Duda and Hart, 1973].

As a philosophical sidenote, there is a paradox associated with database search whose crux lies in the fact that database search must define a query that describes what is being sought. Meno's dilemma from Plato's dialogs exemplifies this paradox in the question: how can we learn anything new? To find something that is truly new we must find something that we didn't have any knowledge of, but if we had no knowledge of what we were looking for then how could we start searching for it? This is relevant to biological sequence analysis

```
   [     ]  <     D-domain    > <   Anticodon   >< Extra ><    T-domain   >[     ]
   (((((((  ((((           )))) (((((  ===  )))))         (((((       ))))))))))))
1 -GCCAAGGTGGCAGAGTTCGGCCTAACGCGGCGGCCTGCAGAGCCGCTC----ATCGCCGGTTCAAATCCGGCCCTTGGCT---
2 -GGGCGTGTGGCGTAGTC-GGT--AGCGCGCTCCCTTAGCATGGGAGAG----GTCTCCGGTTCGATTCCGGACTCGTCCA---
3 --GCCCCATCGTCTAGA--GGCCTAGGACACCTCCCTTTCACGGAGGCG----A-CGGGGATTCGAATTCCCCTGGGGGTA---
4 -GGCGGCATAGCCAAGC--GGT--AAGGCCGTGGATTGCAAATCCTCTA----TTCCCCAGTTCAAATCTGGGTGCCGCCT---
5 -GTCTGATTAGCGCAACT-GGC--AGAGCAACTGACTCTTAATCAGTGG----GTTGTGGGTTCGATTCCCACATCAGGCACCA
6 -GGGCGAATAGTGTCAGC-GGG--AGCACACCAGACTTGCAATCTGGTA----G-GGAGGGTTCGAGTCCCTCTTTGTCCACCA
7 -GGGGCTATAGTTTAACT-GGT--AAAACGGCGATTTTGCATATCGTTA----T-TTCAGGATCGAGTCCTGATAACTCCA---
8 -AGCTTTGTAGTTTATGTG-----AAAATGCTTGTTTGTGATATGAGTGAAAT--------------------TGGAGCTT---
```

Figure 2.3: A multiple alignment of t-RNA. There are eight sequences in this alignment listed vertically. The sequences themselves are listed horizontally. The sequences contain insert/delete characters '-' to help align bases in specific columns. The first two lines represent the secondary structure of the molecule and are not present in multiple alignments that do not involve secondary structure. A left parenthesis represents a 5' base of a basepair and a right parenthesis represents a 3' base of a basepair.

because sequence databases can over-represent certain molecules. If a certain molecule is well characterized, then programs can search the databases and find new instances of that molecule, adding more data, and building a better characterization in a positive feedback loop. However, less well characterized molecules are neglected. This paradox is somewhat solved by full genome sequencing projects. Here *all* sequences are found not just well characterized ones. New things are found because *everything* is considered. New classes of sequences are defined to account for clusters of similar sequences that do not fall into any known classes.

2.1.4 Multiple Alignment

A multiple alignment is a tool used in biology to show the correspondence between a set of sequences. This correspondence is shown by aligning in columns the portions of the sequences that are similar. Specific columns in a multiple alignment group together the bases from the set of sequences that perform the same function. In globin proteins, for example, there is an amino acid, histidine, that helps bind the heme group. In a multiple alignment of several globin proteins, an occurrence of the heme binding histidine will be isolated in a specific column for all the globin sequences indicating this amino acid plays the same role in all sequences. In statistical sense, a column in a multiple alignment can be thought of as defining a distribution of bases in that position of the alignment. See Figure 2.3 for an example multiple alignment. Multiple alignment is important because it allows sequences to be viewed structurally despite the "noise" injected by evolutionary mutations. This analysis allows important regions of the sequences to be determined and might lead to a better chemical understanding or to a better evolutionary understanding. The theory of formal languages helps to address the problem of identifying the set of subsequences or structural elements that are placed together in a hierarchy by providing a theoretical framework in which to discuss such problems.

Gutell [Gutell, 1993] believes there is much room for improvement in computer aided alignment:

> Although there have been significant advances in computer generated alignment in recent years, the most precise method is still a manual one... not just sequence similarity. Homology is best defined in the context of larger units of homologous structure and function, and involves higher order structure.

2.2 Sequence Similarity and Homology

2.2.1 Biological View

Sequence similarity is important in biological modeling because of the concept of homology. Homology groups together different objects that are descended from a common ancestor. This is important because molecules that are homologous usually share a common function. Unfortunately, determining whether two sequences are homologous is difficult. To answer the question fully one would have to know the complete evolutionary ancestry of both molecules back to a common ancestor, which is usually impossible to know. For this reason, the question of homology and functionality is usually answered by addressing the problem of computing sequence similarity.

Sequence similarity is important because the function of polymers such as proteins and RNAs is determined by structure, and structure is largely determined by the identity of the primary sequence of bases making up the polymer. Molecules such as proteins might differ in primary sequence because of evolutionary events such as mutation, insertion, and deletion, but still share a common function. Selection pressure weeds out destructive changes and encourages changes that improve function.

If the primary sequences of two protein molecules are the same then one can infer that they have the same function. If two sequences share 95% identity in primary sequence then one can usually infer functional similarity. When primary sequence identity becomes less well conserved, functional inference becomes more difficult and the models for computing sequence similarity must be fairly sophisticated and use what little knowledge is available in the sequences as best they can to determine similarity.

One would like to infer functional similarity and possibly homology of two molecules by determining whether they share primary sequence similarity. This is routinely done throughout the biology community.

It is important to note that homology implies some degree of sequence similarity but sequence similarity does not necessarily imply homology because of mechanisms such as convergence. A simple example is that of bats and birds. Both have wings but this does not imply that they are both descended from a common winged ancestor. The true answer is they evolved wings independently.

2.2.2 Computer Science View

Computing sequence similarity was first solved as the computer science problem of approximate string matching [Baase, 1988]. The problem here is to search a database of text for a search string considering that misspellings might occur. If the search is for PIPELINE, then the database words PIELINE or PIPELIME might be allowed with only one misspelled or deleted letter. Depending on how many misspellings and indels (insertions or deletions) are allowed, the search can return only exact matches to words, or words having great deviations in spelling. Any word can be interpreted as a misspelling of any other word given a sufficient number of allowable spelling errors. A dynamic programming algorithm can perform these string searches allowing mismatch, insertion, and deletion in $O(sd)$ time where s is the size of the search string and d is the size of the text database. For fixed search string length, this time is linear in the database size.

In the world of biosequence analysis, one of the first algorithms for computing sequence similarity was the Needleman–Wunsch algorithm [Needleman and Wunsch, 1970]. In essence, this algorithm is the approximate string matching dynamic programming algorithm but with a cost structure that reflects biological rules about mutating and deleting characters of the sequences. The algorithm defines a distance matrix, $d(a, b)$, that represents the cost of a mutation of residue a into b and a gap cost, $g(a)$, that represents the cost of a inserting or deleting the residue a. Given two sequences a dynamic programming algorithm can be constructed that finds an alignment with minimum cost in time proportional to the product of the sizes of the strings, which is approximately $O(n^2)$ if each sequence has n letters. Extensions of this algorithm to optimally align r sequences run in time $O(2^r n^r)$, which is very prohibitive computationally.

There are common approximations to this algorithm that use heuristics to dramatically reduce the running time. BLAST is a very widely used approximation [Altschul et al., 1990]. BLAST uses the heuristic that an alignment with a good overall score usually contains regions that have *very* good scores or are identical. BLAST searches for these short regions of very high scores, which can be done very quickly because the regions are so short. It then uses these high scoring regions as starting points for the alignment and tries to extend them until the extension creates a poor overall score.

Gribskov and others extended sequence-to-sequence alignment by considering sequence-to-profile alignment [Gribskov et al., 1990]. A profile is a position-specific scoring table. The profile gives a score for the ith residue matching the jth column of the profile. There is not simply a monolithic distance matrix that is used at all positions in the sequence as in the Needleman–Wunsch algorithm but rather a distance matrix is constructed in which different positions in the sequence are assumed to have different statistical identities. This captures the idea that a certain position, j, in a molecule might contain G or C equally likely while another position, $(j + 2)$, might contain A or T equally . A monolithic distance matrix would be unable to represent these disparate behaviors.

A profile is calculated, in essence, by counting the number of times the ith residue occurs in the jth column of a multiple alignment. For example, a multiple alignment with a column containing a conserved G would generate a corresponding profile position with high penalties for A,C, and T and a very low penalty for G. In a probabilistic framework, this position would have a very low entropy distribution. Aligning a sequence to a profile allowing gaps is essentially the same as the Needleman–Wunsch algorithm and takes time $O(nm)$ where n is the length of the sequence and m is the length of the profile.

The sequence alignment problem can be generalized to probabilistic machine models that try to capture the information of evolutionary sequence mutation structure. These approaches have received considerable attention. They are based upon language-accepting machines of formal language theory and benefit from the rich mathematical underpinnings of probability theory. Two models are considered in this work: hidden Markov models, which are based on regular languages, and stochastic context-free grammars, which are based on context-free languages.

2.3 Probabilistic Machine Learning Theory

This work addresses sequence modeling in a rigorous framework of probability and formal language theory. Probability allows us to perform operations such as estimation and hypothesis testing. Formal language theory allows us to describe the complexity of interactions in a rigorous way.

2.3.1 Syntactic or Structural Pattern Recognition

Syntactic pattern recognition research addresses the problem of classifying objects in a structural or syntactic framework [Fu, 1982]. Unlike the decision-theoretic framework, in which features extracted from the objects determine assignment to a class, the syntactic framework not only assigns an object to a class but describes aspects of the object that make it ineligible for another class. It describes complex patterns in objects in terms of a hierarchical composition of simpler subpatterns. This approach makes sense when the number of patterns is very large and having each pattern defining a class would be impractical. Instead classes are described by how simpler subpatterns can be structurally arranged to make a whole class of superpatterns.

This approach greatly reduces the size of the specification of the pattern. Instead of enumerating all patterns, a structural meta-pattern is described which elucidates how subpatterns can be arranged to give the class of all patterns. This is the power of language: a set of words and a grammar describing how words can be put together is much more expressive given its size than a simple set of primitives in which each new concept must be described by a new word. This power comes at a cost. Unlike a simple set of primitives in which the meaning of the primitive is a directly mapped to a concept, in the structural framework a string of primitives must be parsed using a grammar to determine its structure and meaning.

The language metaphor is used in biological sequence modeling when a language is constructed that describes how mutational events occur. This language is an abstraction of the real biological world but nevertheless tries to capture the essence of it without becoming overly cumbersome. This language can decompose and classify portions of a biological sequence into structural elements much in the same way English grammar can decompose and classify an English sentence into elements such as nouns and direct objects. These biological structural elements can be used to construct biological multiple alignments that align structural elements of different sequences into multiple alignment columns.

2.3.2 Bayes Decision Theory

Database search is one of the primary tasks in sequence modeling. At the heart of a database search is the question of discriminating and classifying an arbitrary sequence as being a member of a class or not. When searching for t-RNA sequences, for example, a database search should answer for each sequence in the database whether that sequence can be classified as a t-RNA or not. Discrimination can be addressed through decision theory [Duda and Hart, 1973, Rissanen, 1989, Jefferys and Berger, 1992].

Consider the case in which a scientist is trying to explain an event using a hypothesis which comes from a predetermined set of hypotheses. Associated with each one of these hypotheses is a prior probability, which reflects a degree of belief as to whether that hypothesis is correct or not. Given this set of hypotheses and prior probabilities, the scientist dutifully performs an experiment and collects some data. The scientist would like to use this data to more accurately choose the "correct" hypothesis for describing the data. This situation is best dealt with using Bayes decision theory.

Let the hypotheses be H_i, the priors be $P(H_i)$, and the data be D.

One way to choose the best hypothesis is by examining the likelihood of each hypothesis and choosing the best one, the maximum likelihood choice,

$$\mathrm{argmax}_{H_i} P(D|H_i) \tag{2.1}$$

However in a Bayesian setting, what the scientist would like is to move from *prior* probabilities, $P(H_i)$, to *posterior* probabilities, $P(H_i|D)$, using the data, D. Bayes rule provides just that,

$$P(H_i|D) = \frac{P(D|H_i)P(H_i)}{P(D)} \tag{2.2}$$

where $P(D|H_i)$ is referred to as the likelihood and $P(D)$ is $\sum_i P(D|H_i)P(H_i)$, a normalizing constant.

The "best" hypothesis for the data in a Bayesian setting is the hypothesis that maximizes $P(H_i|D)$ or in other words is the *maximum a posteriori* choice. This is also equivalent to choosing the hypothesis that minimizes $-\log P(H_i|D)$ or equivalently minimizes $-\log P(D|H_i) - \log P(H_i)$ where the $\log P(D)$ term has been dropped because it does not depend on H_i. This latter formulation has a Minimum Description Length interpretation [Rissanen, 1989]. The term $-\log P(D|H_i)$ can be interpreted as the number of bits needed to encode the data given the ith hypothesis while $-\log P(H_i)$ can be interpreted as the number of bits needed to encode the ith hypothesis. The sum of these two quantities gives the total information needed to communicate the data in a two-stage description process in which the hypothesis is transmitted first and then the data is transmitted given the hypothesis.

Consider the restricted case in which there are only two hypotheses and the scientist is trying to choose the best one. Usually the competing hypotheses are a constructed model and random noise, or the null hypothesis. In this case a discrimination function can be constructed,

$$g(D) = \log \frac{P(H_1|D)}{P(H_2|D)} \tag{2.3}$$

This is known as the log posterior odds. It is equal to

$$g(D) = \log \frac{P(D|H_1)}{P(D|H_2)} + \log \frac{P(H_1)}{P(H_2)} \tag{2.4}$$

which a sum of the log likelihood ratio and the log prior odds.

This function is constructed so that if $g(D)$ is bigger than zero then H_1 is the "best" hypothesis otherwise H_2 is "better". The function indicates which hypothesis is better and can be applied to database search. Let H_1 be a true model and H_2 be a null model against which the true model must compete. Also assume that by prior knowledge there is approximately one true sequence in a database of 2^{20} sequences. This transforms the previous equation to

$$g(D) = \log \frac{P(D|model)}{P(D|null)} - 20 \tag{2.5}$$

Thus the data, D, will only be classified as true if the model encodes D with at least 20 fewer bits than the null model or in other words it has a savings of 20 bits or more over the null model. We use bits because we assume the logarithm is base 2.

Bayes decision theory also addresses the issue of parameter estimation. Let θ represent all the parameters of a model. Let the observed data, D, be a set composed of n observations, d_i. The conditional probability, $P(d|D)$ must be computed where d is a new observation, not part of the given set of observations $D = \{d_i\}$. The form of $P(d|\theta)$ is known but its parameters are not. Initial knowledge of θ is contained in $P(\theta)$. The remainder of knowledge is contained in the data, D.

We can start with the equation

$$P(D|\theta) = \prod_k P(d_k|\theta) \tag{2.6}$$

This equation is the likelihood of θ with respect to the data. It assumes that each of the observations, d_i, are independent by using a product. Sometimes this is a very wrong assumption. In biological sequence analysis most sequence databases are not constructed using a random sample and therefore might be heavily biased toward a group of related organisms. Estimations that do not take into account these correlations in the data set can do poorly. If the observations are not independent, some parameterized model of dependence can be made [Altschul et al., 1989].

The Maximum Likelihood Estimate (MLE), θ, is chosen to maximize the likelihood, $P(D|\theta)$, ignoring the prior, $P(\theta)$. Let $\hat{\theta}$ be the MLE of θ then $P(d|D)$ is estimated as $P(d|\hat{\theta})$. It is sometimes useful to consider

$$-\log P(D|\theta) = \sum_k -\log P(d_k|\theta) \tag{2.7}$$

Minimizing this negative log likelihood equation over θ is equivalent to finding the Maximum Likelihood Estimate.

In Bayesian estimation, as opposed to maximum likelihood, the distribution $P(d|D)$ is sought explicitly using the prior, $P(\theta)$, and is computed by averaging over all possible values of the parameters.

$$P(d|D) = \int P(d|\theta)P(\theta|D)d\theta \tag{2.8}$$

By Bayes rule

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{\int P(D|\theta)P(\theta)d\theta} \tag{2.9}$$

This is known as the posterior distribution over the parameter, θ given the data, D. Note here that no single estimate $\hat{\theta}$ of θ is made. Rather one averages over all possible $\hat{\theta}$ according to the posterior distribution given the data, D. One can choose a single estimate, $\hat{\theta}$ that maximizes the posterior, the maximum a posteriori (MAP) estimate. One can also choose the estimate that is the average with respect to the posterior, the posterior mean estimate (PME).

2.3.3 Information Theory

There are several information theoretic quantities that will be referred to later [Cover, 1991].

Entropy of a distribution is defined to be

$$h(p) = -\sum_{i}^{k} p_i \log p_i \tag{2.10}$$

where p is the discrete probability distribution. The entropy ranges from zero to $\log k$ where the units are bits if the log base is 2 or nats if the log base is e.

Information is defined to be the difference in entropies before and after an event occurs. For example, consider a DNA base and assume that the prior distribution for the identity of this base is uniform over A,G,C,T. The entropy of this four possible outcome distribution is 2 bits. After estimation, the distribution shows the base A occurs with probability one. The entropy of this posterior distribution is 0 bits. The information conveyed by this position is therefore 2 bits.

The Kullback-Liebler divergence or K.L. distance for two distributions, p and q, is defined to be

$$h(p||q) = \sum^{k} p_i \log \frac{p_i}{q_i} \tag{2.11}$$

This quantity is always greater than or equal to zero, which can be shown using the fact that $\log x < x - 1$.

Mutual information is defined for a joint distribution, p_{ij}, and singleton distributions p_i, p_j (the marginals: $p_i = \sum_j p_{ij}$, $p_j = \sum_i p_{ij}$) to be

$$\sum_{i=1}^{k} \sum_{j=1}^{k} p_{ij} \log \frac{p_{ij}}{p_i p_j} \tag{2.12}$$

and ranges from zero to $\log k$ bits if the base of the log is 2. Mutual information can be written in terms of entropy $h(p_i) + h(p_j) - h(p_{ij})$

2.3.4 Expectation Maximization

Parameter estimation is defined in a framework in which a set of data observations is used to estimate the parameters of the model such that the likelihood of the model given the data is maximized (maximum likelihood estimation). However there are many cases in which only incomplete data is given and some data are missing. When complete data are known it can be possible to write down a closed form maximum likelihood estimate given a likelihood function, $P(x|\theta)$, where x is a complete data specification and $P(x|\theta)$ is the likelihood of a model, θ. However when the data is incomplete, a closed form maximum likelihood estimate usually does not exist. Let y be an incomplete data specification. There is a many-to-one mapping $M : x \to y(x)$ from complete data specifications to incomplete data specifications indicating that there are many complete data specifications that can give rise to the same incomplete data specification. Let the likelihood of a model, θ, given incomplete data, y, be given by $g(y|\theta)$. The relation of g to P is given by

$$g(y|\theta) = \int_{M^{-1}(y)} P(x|\theta)dx \tag{2.13}$$

This equation gives that the incomplete data likelihood given incomplete data y is an integral over all complete data specifications, x, that give rise to y. The Expectation Maximization algorithm describes a class of algorithms that solve this maximum likelihood estimation problem with "missing data" using an iterative procedure [Dempster et al., 1977].

Practically, the EM algorithm can be thought of as the following procedure

1. Start with initial model parameters, θ.
2. Estimate sufficient statistics for the complete data, $t(x)$, given the incomplete data, y, and θ. **E step**
3. Calculate new model parameters, $\bar{\theta}$, to maximize the likelihood as though the estimated statistics for the complete data were actually observed. **M step**
4. Set $\theta = \bar{\theta}$ and iterate to step 2 until some stopping criteria is met.

It can be shown that this procedure increases the likelihood at each step [Dempster et al., 1977].

There are algorithms that rely on EM to estimate parameters of models. Of special importance to this work are the algorithms for hidden Markov model(HMM) and stochastic context-free grammar(SCFG) parameter estimation. In the case of HMM parameter estimation, there is a well known Baum-Welch algorithm [Rabiner, 1989]. SCFGs have an analog of the Baum-Welch algorithm called inside-outside [Lari and Young, 1990].

2.3.5 Priors and Dirichlet Estimation

Estimation formulas are based on frequencies of occurrence of various events, such as the number of times there was a transition from state i to j in a sequence model. In the limit as the number of sequences, or data points, grows large, these frequencies will approach the true probabilities. However when training sets are small, the influence of the Bayesian prior is very important [Brown et al., 1993]. As a practical matter, a Bayesian prior can be thought of a way from keeping probabilities from becoming zero even though no occurrences of an event were observed. For example, even though no bulges are observed in a helix for several examples of different t-RNA molecules, the true probability of a bulge is most likely not zero. The Bayesian prior helps to estimate the probability of the bulge even though no occurrences were observed by calculating the Maximum Posterior Estimate(MPE) instead of the Maximum Likelihood Estimate(MLE).

We assume Dirichlet priors in our modeling [Sjölander et al., 1996]. Dirichlet priors have many attractive features that stem from the fact that they are conjugate distributions to multinomial likelihoods that are used extensively in our models [Winkler, 1972]. Having conjugate distributions leads to very elegant estimation formulae. The prior and posterior have the same form when examining mean posterior estimates, for example.

Simple Dirichlets can be extended to mixtures of Dirichlets in a straightforward manner. The parameters of the Dirichlet can be estimated in a maximum likelihood sense from a set of data [Sjölander et al., 1996].

2.4 Formal Language Theory

2.4.1 The Chomsky Language Hierarchy

Formal language theory deals with sets of strings called languages and different mechanisms for generating and recognizing them [Harrison, 1978, Hopcroft and Ullman, 1979]. It is a formalization that is abstractly based on human language. A language is a set of strings over some alphabet. A string is a sequence of letters of the alphabet. For a given language there will be a set of strings which are contained in the language and a set of strings which are not in the language. A language might be the set of all even length strings or the set of all strings in which the number of as equals the number of bs, for example. It is one of the great results of theoretical computer science and linguistics that languages form a definite hierarchy called the Chomsky hierarchy after one of the primary originators of formal language theory, Noam Chomsky. There are some languages which are simpler than others. This hierarchy of language complexity is determined by characterizing different languages in general by the complexity of the machine models that are required to represent them.

This hierarchy is important in sequence modeling because of complexity of interactions in sequence structure. Simple models cannot represent some types of sequence structure. It is impossible to represent balanced parenthesis with a regular language, for example[Hopcroft and Ullman, 1979]. Depending on the level of complexity of interactions, an appropriate machine model can be chosen.

When describing strings some notation is useful. The notation, a^j, means that the symbol, a, is repeated j times ($a^3 = aaa$). The Kleene closure, a^*, equals $\cup_{i=0}^{\infty} a^i$ which means zero or more occurrences of the symbol a. The alphabet is denoted by the set Σ. In the case of binary symbols the alphabet, Σ, is the set $\{0, 1\}$.

Regular languages are the simplest and are representable by any of the other more complex language models. An example of a regular language is $L = \{\Sigma^n | n \text{ is even}\}$, the set of all even-length strings. Deterministic Finite Automata (DFA) can recognize regular languages. DFAs are characterized by a finite number of states and transitions between them that generate symbols.

Context-free languages are a superset of regular languages. An example of a context-free language is $L = \{ww^R\}$, the set of all strings consisting of a string w concatenated with its reverse. Push-Down Automata(PDA) recognize context-free languages. PDAs extend DFAs by incorporating a stack that can push and pop stack symbols. In addition to the finite state structure DFAs have, PDAs can store information on an infinite stack and access the top symbol using simple push and pop operations. The reverse language is easily recognized by a PDA, it simply pushes symbols onto the stack until the end of w is reached and then the symbols are popped off and checked against w^R to make sure they match. DFAs cannot model this. For arbitrarily long w, there is no way for a DFA to keep track of what symbols in w have been seen in order to check them against w^R. If we assume the length of w is bounded as in $L_k = \{ww^R : |w| \leq k\}$ where $|w|$ denotes the length of w, then a DFA can recognize the language. However this leads to a DFA with a huge number of states. This is somewhat uninteresting because all finite languages are regular (a DFA simply has to enumerate all strings in the finite language in its finite state structure to accept it).

Context-sensitive languages are a superset of context-free languages. An example is $L = \{a^i b^i c^i\}$, an equal number of as followed by bs followed by cs. Linear Bounded Automata(LBA) recognize context-sensitive languages. LBA are characterized by a finite state structure and a two-way tape that has bounded length. LBAs are more complex than PDAs because, rather than storing information in a simple stack, LBAs have a more complex tape in which symbols can be read and written anywhere on the tape.

Universal languages contain all computable languages and represent computation in general. Turing Machines (TMs) recognize Universal Languages. TMs are characterized by a finite state structure and an unbounded tape. Any computable problem can be reduced to a language accepted by a TM.

There are also uncomputable languages that are not representable by TMs. These languages would lead to contractions if they were, in fact, computable. The halting problem leads to a language in which strings are descriptions of Turing Machines that halt when run. It can be shown that this and many related languages are not computable.

2.4.2 Context-Free Grammars

This work develops stochastic context-free grammars, which are a probabilistic extension of context-free grammars. A context-free grammar, G, is formally defined as a 4-tuple: $G = (V, \Sigma, P, S)$ [Hopcroft and Ullman, 1979], where V is a set of nonterminals, Σ is the terminal alphabet, S is the start symbol, and P is a set of productions of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

A context-free grammar generates strings through a rewrite process starting from the start symbol in which productions are used to rewrite nonterminal symbols with strings of nonterminal and terminal symbols. This type of formal language was proposed as a model for human language in which grammar productions model language rules such as a noun phrase is defined to be either a noun, a noun preceded by a determiner, a noun preceded by an adjective, etc..

Consider the nested parenthesis grammar. This grammar is defined as $V = \{T\}$, $\Sigma = \{(,)\}$, $S = \{T\}$, and P. The set of productions, P, contains three productions $(\mathbf{a}, \mathbf{b}, \mathbf{c})$:

$$\begin{aligned} \mathbf{a} : T &\rightarrow TT \\ \mathbf{b} : T &\rightarrow (T) \\ \mathbf{c} : T &\rightarrow \epsilon \end{aligned}$$

Here ϵ denotes the empty string.The grammar can be used to derive the string $x = ((())(()))$ in the following way using the indicated productions $(\mathbf{a}, \mathbf{b}, \mathbf{c})$:

$$\begin{array}{rcl} \mathbf{b} : T & \Rightarrow & (T) \\ \mathbf{a} : & \Rightarrow & (TT) \\ \mathbf{b} : & \Rightarrow & ((T)T) \\ \mathbf{b} : & \Rightarrow & (((T))T) \\ \mathbf{c} : & \Rightarrow & ((())T) \\ \mathbf{b} : & \Rightarrow & ((())(T)) \\ \mathbf{b} : & \Rightarrow & ((())((T))) \\ \mathbf{c} : & \Rightarrow & ((())(())) \end{array}$$

A parsing algorithm can determine if an arbitrary string w can be derived using a grammar G in polynomial-time [Hopcroft and Ullman, 1979]. If it can, the parsing algorithm returns true and the strings legal derivation (the sequence of productions that derive w), otherwise it returns false. Thus a parsing algorithm given x= $((())(()))$ and the parenthesis grammar will return true and b, a, b, b, c, b, b, c. Given y= $())(()$ and the parenthesis grammar, the parser will return false because it is impossible to derive y using the grammar. Parsing algorithms are used to parse computer languages to determine whether they have a syntactically correct form and how they should be interpreted.

Associated with every derivation is a derivation tree. Every node of the tree is labeled with a nonterminal or terminal symbol. If a series of nodes labeled $A_1, A_2, ..., A_n$ are direct descendants of a node labeled A then the production $A \rightarrow A_1 A_2 ... A_n$ is used in the derivation of the string. The frontier of the tree is a string constructed by concatenating the labels of the leaves of the tree left to right. See Figure 2.4 for a derivation tree.

A grammar is unambiguous if for two derivation trees, T and T', if the frontier of T equals the frontier of T' then $T = T'$. Equivalently, a grammar is ambiguous if there exist two distinct derivation trees whose frontiers are equal. Ambiguous grammars can derive a single string in more than one way.

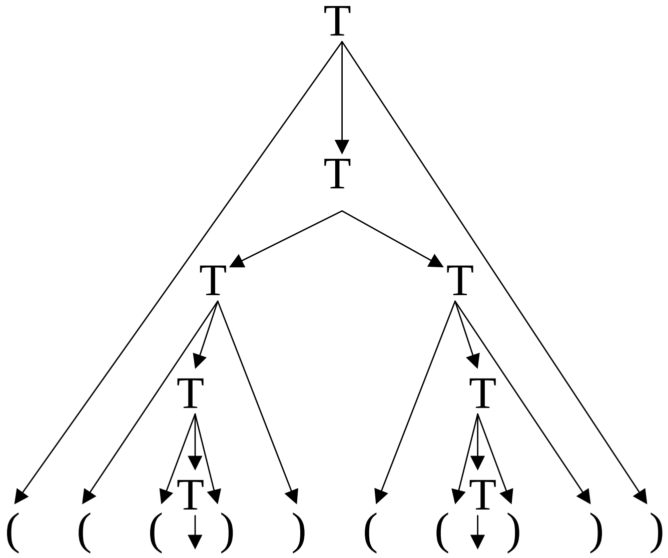


Figure 2.4: The derivation tree of the string ((())(())) under the parenthesis grammar described in the text. The single nonterminal is T and the alphabet is { (,) }. The frontier of the tree is at the bottom of the figure. Arrows denote direct derivation.

Chapter 3

Efficient Biological Sequence Modeling Using Stochastic Context-Free Grammars With Constraints

Methods for performing sequence alignment have proliferated, largely as a result of the abundance of biological sequence data from large scale genomic sequencing efforts. Most sequence alignment methods can be reduced to simple dynamic programming algorithms that assume regular sequence models and simple cost structures. These simple dynamic programming algorithms usually have a time complexity that is linear in the length the string being processed. A more recent development has been the extension of sequence models to handle non-regular, or more specifically context-free, sequence structures. These higher order models allow modeling of structures such as pairwise basepairing interactions in RNA sequence alignments. While these context-free sequence alignment methods also reduce to polynomial-time dynamic programming algorithms, their complexity is several orders of magnitude greater than simple regular sequence models. This work examines stochastic context-free grammars and their use in biological sequence modeling including algorithm enhancements using constraints that allow them to run efficiently when presented with large problems.

We start with an introduction to probabilistic sequence models. We then introduce hidden Markov models (HMMs) as a probabilistic model of primary sequence information. Stochastic context-free grammars (SCFGs) are then presented as an extension of HMMs that allow pairwise interactions. We relate SCFGs to HMMs, point out some interesting properties of SCFGs, and describe how SCFGs relate to other methods. The problem of parsing is then treated with several solutions. We introduce constraints as a natural solution to the complexity of SCFG parsing. We show how constraints can be generated and efficiently used by a SCFG parser. Finally we present the results of a practical application of SCFGs to RNA modeling.

3.1 Probabilistic Sequence Modeling

Mathematical models of strings and biological sequences are rooted in both the mathematical theory of string editing [Baase, 1988] and biological considerations of evolution [Brock et al., 1994]. In string editing, the basic operations of match, insert, and delete define the dynamic programming algorithm that computes fuzzy string matches. These operations mimic the natural process of DNA evolution in which sequence positions can be mutated and skipped, and extra bases can be inserted. Traditional sequence alignment algorithms produce optimal alignments given costs associated with insertions, deletions, mismatches, and matches and are defined in terms of a dynamic programming algorithm [Waterman, 1989]. A recent development has been the specification of probabilistic machine models for string matching that capture these string editing operations in a cohesive framework.

In one of the first characterizations of a probabilistic machine model for string matching, hidden Markov models [Krogh et al., 1994a, Baldi and Chauvin, 1994] use an explicit state structure that embodies the match, insert, and delete operations. This state structure contains a number of explicit match states for the modeling of the core positions of the molecule. In addition to these match states there are other states that allow the deletion of any of the match states and insertion of any number of bases before and after any of the match state positions. The use of hidden Markov models for modeling protein and DNA structures has flourished [Krogh et al., 1994b, Krogh et al., 1994a, Baldi et al., 1994b, Baldi and Chauvin, 1994, Baldi et al., 1994a, Tanaka et al., 1993, Raman and Overton, 1994, Fujiwara et al., 1994, Sonnhammer et al., 1997, Amtai, 1998, Park et al., 1998, Baldo and McClure, 1999, Bertilsson et al., 1998].

HMMs are probabilistic generative models that output strings by moving through a discrete state structure using Markov decisions indexed by time. At each point in time, i, the current state, π_i, generates symbols according to some probabilistic rule. The states can be given semantic meaning by giving them biological interpretation such as characterizing the position in globin proteins that binds the heme group using a histidine amino acid.

HMMs are practical because they can construct multiple alignments and perform database searches efficiently. Following the review of Rabiner [Rabiner, 1989, Rabiner and Juang, 1986] in which hidden Markov models (HMMs) are presented with applications to another problem domain, speech recognition, we define three major problems as described above and give algorithms for their solution:

1. **Evaluation**: find the probability of a string given a model.
2. **Decoding**: find the most likely way a model generated a string.
3. **Learning**: find parameter estimates of the model that maximize the probability of a set of strings.

The next section gives a characterization of hidden Markov models. This is followed by a characterization of stochastic context-free grammars(SCFGs). Like HMMs, SCFGs have a finite state structure that is traversed using time-indexed Markov decisions. However SCFGs generate output symbols in a strictly more complex way because output symbols can be generated in a branching process. The class of all HMMs are shown to be a strict subset of stochastic context-free grammars.

Much of the notation adopted here comes from the excellent treatment given in the book *Biological Sequence Analysis* [Durbin et al., 1998].

3.2 Hidden Markov Models (HMMs)

3.2.1 Definition

As mentioned previously, HMMs are probabilistic generative models that output strings by moving through a discrete state structure using Markov decisions indexed by time. At each point in time, i, the current state, π_i, generates symbols according to some probabilistic rule.

An HMM is mathematically defined rather simply as

1. M: the finite number of states in the model.

2. Σ: the alphabet or the set of all output symbols. For DNA this might be defined to be $\{A, C, G, T\}$.
3. $a_{kl} = P(\pi_i = l|\pi_{i-1} = k)$: the Markov state transition probability distribution. a_{kl} gives the probability of moving from state k at time $(i-1)$ to state l at time i.
4. $e_k(b) = P(b|\pi_i = k)$: the probability that symbol b is generated by state k.
5. S: the start state

3.2.2 Algorithms

Evaluation

Given an observation sequence of length L, $x = x_1 \ldots x_L$, and an HMM, M, finding the probability of the sequence given the model, $P(x|M)$, is done by defining a forward variable, $f_k(i)$. The forward variable is the probability of observing the partial observation sequence, $x_1 \ldots x_i$ and being in state k at time i given the model M

$$f_k(i) = P(x_1 \ldots x_t, \pi_i = k|M) \tag{3.1}$$

The forward variable is defined recursively

$$f_l(i+1) = [\sum_k f_k(i) a_{kl}] e_l(x_{i+1}) \tag{3.2}$$

The forward variables can be computed using dynamic programming in $O(LM^2)$ time in the worst case where L is the length of the string and M is the number of states in the HMM. This assumes the HMM's states are fully connected. If the in-degree of states is bounded by a constant then the time is $O(LM)$.

The full probability of the sequence $x = x_1 \ldots x_L$ is a sum of forward variables across all states after generating the entire L length string : $P(x|M) = \sum_{i=1}^{M} f_i(L)$.

Decoding

The most likely path (or Viterbi path) is defined to be the path, π, that maximizes the probability of the sequence

$$\text{argmax}_\pi P(x, \pi|M) \tag{3.3}$$

This can also be found in $O(LM^2)$ time, where L is the length of the string and M is the number of states in the HMM using Equation 3.2 and replacing the sum with the maximum term.

$$f_l(i+1) = [\max_k f_k(i) a_{kl}] e_l(x_{i+1}) \tag{3.4}$$

At each point in the dynamic programming, the subproblem is maximized. Each subproblem is independent given the computation of the previous time point. Therefore the ending solution is maximized. Because each maximization corresponds to a specific state, k, backtracking from the final solution through the maximal state structure gives the most likely path.

Learning

In order to estimate HMM parameters from a set of sequences one would like to compute the maximum likelihood estimate. Unfortunately a closed form solution for estimated parameters does not exist and instead an Expectation-Maximization algorithm is used. The EM algorithm calculates the expected frequencies of model events such as a state generating a terminal symbol or a transition occuring between two states given the current values of the model parameters. It then uses these expected frequencies to estimate maximum likelihood parameters given the expected frequencies (Section 2.3.4).

In order to find posterior probabilities such as $P(\pi_i = k|x, M)$ it is useful to consider $P(x, \pi_i = k|M)$, the probability of observing the entire string x and having the ith symbol being produced by state k. This probability can be computed using the previously described forward variable, along with the backward variable, $b_k(i)$. The backward variable is defined to be the probability of observing the string $x_{i+1} \dots x_L$ given that the ith symbol is produced by state k

$$b_k(i) = P(x_{i+1} \dots x_L | \pi_i = k, M) \tag{3.5}$$

This backward variable, like the forward variable, is defined recursively

$$b_k(i) = \sum_l a_{kl} e_l(x_{i+1}) b_l(i+1) \tag{3.6}$$

The product of forward and backward variables gives the desired probability, $P(x, \pi_i = k|M) = f_k(i)b_k(i)$. The posterior can now be written as

$$P(\pi_i = k|x, M) = f_k(i)b_k(i)/P(x|M) \tag{3.7}$$

These posterior counts can be used for parameter estimation. Note that counts from multiple sequences are usually simply added together assuming the sequences are independent.

Viterbi training is an approximation to this method. Instead of computing the posterior, a simple count of one is used if the event occurs in the generation of the Viterbi path of a sequence. Viterbi training serves as a good approximation in most HMM applications.

3.2.3 HMM Time Complexity is Linear, $O(L)$

The above algorithms run in linear time with respect to the length of sequence, L. Linear time algorithms run in time proportional to the time it takes to scan the sequence from left to right. However the constants for HMM algorithms might not be good enough for very large database searches and computational strategies must be used to speed the search. The popular alignment program BLAST uses constant-time table look up on fixed length strings to help speed its searches [Altschul et al., 1990]. These constant time sequence look-ups are then used as alignment seeds and are extended until the score falls below a cut off.

3.2.4 Biological Sequence Modeling Using HMMs

HMMs capture the match, insert, and delete structure of string editing that is an approximation to evolutionary processes. In multiple alignments of biological sequences there are insertions, deletions, mismatches, and matches. An HMM captures these events in its state structure. See Figure 3.1 for the structure of a HMM protein model [Krogh et al., 1994a, Karchin and Hughey, 1998]. The boxes represent core matching positions in the protein that output a base according to a multinomial distribution. The diamonds represent insertions between core positions. The circles represent deletions of core positions. This type of model is composed of a number of superstates connected together in a left-to-right manner. The superstates contain

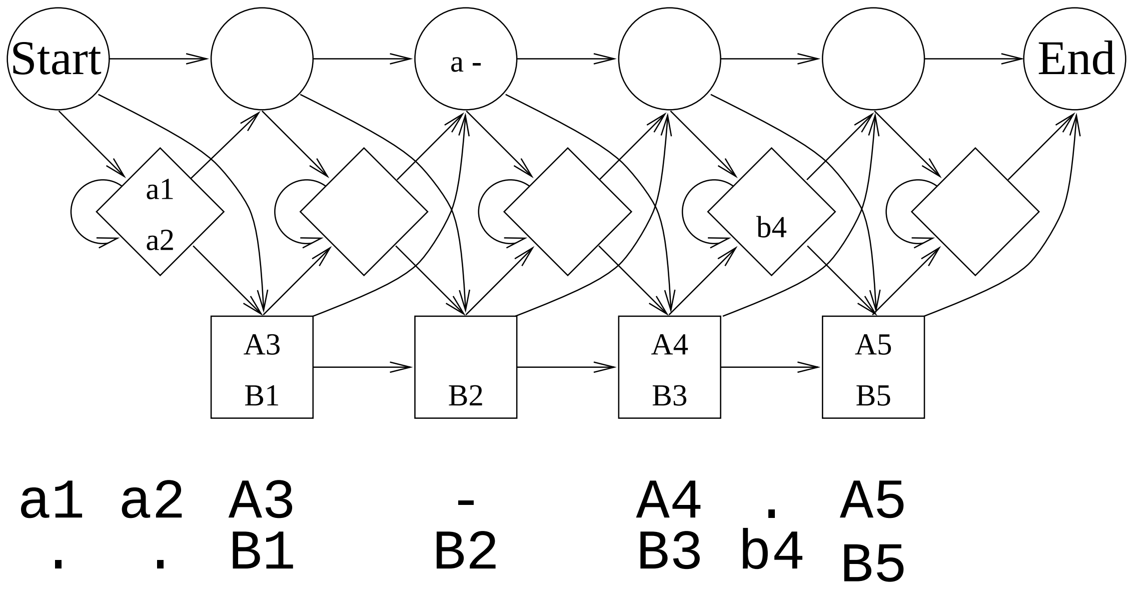


Figure 3.1: The profile HMM protein model. This HMM models the match, insert, and delete operations of string editing that is an approximation to evolutionary processes. Boxes represent match positions or positions that define the core positions of a biological string. Circles represent the deletion of a corresponding match position. Diamonds represent inserts that can occur between any match position. Lines represent transitions between the states. Also shown is most likely path of two sequences, A and B, through the HMM. These paths are used to contsruct the alignment at the bottom of the figure. Sequence A inserts two bases, a1 and a2, and deletes match position two. Sequence B inserts one base, b4.

several states connected in a predefined structure: one state for match, one state for delete, and one state for insert. Decomposing a model into superstates allows for an economical description of the model because one can simply define the number of superstates and not the complete description of how all substates are connected.

HMMs having this type of model structure have the property that any path through them will visit either the match or delete state of every superstate exactly once and the order of the match/delete states is invariant. This type of modeling is useful because it describes all strings in terms of matches to, deletions from, and insertions to a platonic ideal sequence. The platonic ideal sequence is represented by a left-to-right ordering of match states each generating the most likely base defined by their distribution. This type of modeling will be generalized to stochastic context-free grammars in the next section.

Given this type of HMM and the algorithms described above, biological results can be computed in linear time with respect to the length of the sequence being examined.

An HMM can perform a database search by finding the probabilities of all the sequences in the database given the model using $P(X|M)$ as calculated above where X is a sequence and M is a model. Given these probabilities a Bayes decision function can classify sequences as being members of a class or not.

An HMM can construct a multiple alignment by constructing a Viterbi path of a sequence through the model. This will give a mapping from each base in the sequence to a single model state. After this mapping is obtained, a multiple alignment is trivially constructed by placing in respective columns the bases that map to the core matching positions of the model. If a sequence uses a delete state, a dash is placed in the corresponding column. Inserts are placed between core matching columns. Bases placed in insert regions are not aligned from one sequence to another in such multiple alignments.

An HMM can estimate a model given a set of sequences, aligned or unaligned. It does this by computing posterior counts and estimating a model in an EM estimation.

HMMs have been applied to a wide range of sequence modeling problems including globin and EF-hand proteins [Krogh et al., 1994a], construction of a database of protein alignments Pfam [Sonnhammer et al., 1997], and general applications to biopolymers [Amtai, 1998]. Work has shown that HMMs excel at finding remote homologs in proteins [Park et al., 1998].

3.3 Stochastic Context-Free Grammars (SCFGs)

Stochastic context-free grammars are an extension of hidden Markov models. Like HMMs, SCFGs have a finite state structure that is traversed using time-indexed Markov decisions. However SCFGs generate output symbols in a strictly more complex way than HMMs, because output symbols can be generated in a branching process.

3.3.1 Definition

Stochastic context-free grammars are a probabilistic extension of context-free grammars. A SCFG is defined to be:

1. V: a finite set of nonterminal symbols.
2. Σ: a finite set of terminal symbols.
3. P: a set of productions of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.
4. ρ: a mapping from elements in P to their associated probability. For each nonterminal, B, the sum of probabilities for all productions of the form $B \rightarrow \alpha$ is one where $\alpha \in (V \cup \Sigma)^*$.
5. S: the start symbol which is an element of V.

A stochastic context-free grammar generates strings through a rewrite process in which productions are used to rewrite nonterminal symbols with strings of nonterminal and terminal symbols starting from the start symbol. The sequence of rewrite productions used to derive a terminal string from the start symbol is called the string's derivation. The probability of a derivation is equal to the product of the probabilities of the productions used in the derivation.

SCFGs are an extension of HMMs. There is a direct correspondence between HMM model states and SCFG nonterminals and between the HMM start state and the SCFG start nonterminal. A SCFG's productions and their associated probabilities can represent an HMM's transition and output structure.

3.3.2 Algorithms

For purposes of this section, we introduce notation that involves grammars in Chomsky Normal Form in which grammar productions are of the form $W_v \rightarrow W_y W_z$ or $W_v \rightarrow a$ where $W_v, W_y, and W_z$ are nonterminals and a is a terminal symbol. This restriction allows the statement of algorithms to be done economically. There is a simple algorithm for converting an arbitrary context-free grammar into one that is in Chomsky Normal Form [Hopcroft and Ullman, 1979]. While Chomsky Normal Form is used to introduce the algorithms, the actual implementations handle grammars in a more arbitrary form.

Let there be M nonterminal symbols with the start nonterminal labeled W_1 and all other nonterminals labeled consecutively until W_M. Define the production probabilities, ρ, to be composed of two parts: For productions of the form $W_v \rightarrow W_y W_z$, let $t_v(y,z)$ be the probability of that production. For productions of the form $W_v \rightarrow a$ let $e_v(a)$ be the probability of that production.

Evaluation

Given an observation sequence of length L, $x = x_1 \ldots x_L$, and a SCFG, G, finding the probability of the sequence given the grammar, $P(x|G)$, is done by defining an inside variable, $\alpha(i,j,v)$ [Lari and Young, 1990]. The inside variable is the probability of a subtree rooted with the nonterminal v deriving the subword $(x_i \ldots x_j)$:

$$\alpha(i,j,v) = P(v \Rightarrow^* x_i \ldots x_j|G) \tag{3.8}$$

The symbols, $\Rightarrow^*$, indicate that the lefthand symbol after some number of grammar production applications derives the righthand string. The inside variable is analogous to the HMM's forward variable where the nonterminal t exactly replaces the HMM's state k and the subword $x_i \ldots x_j$ generalizes the HMM's partial sequence $x_1 \ldots x_i$. This generalization is necessary to account for the SCFG's greater representational capability over HMMs. The inside variable is defined recursively in a bottom up fashion starting with subwords of length one.

$$\alpha(i,j,v) = \sum_{y=1}^{M} \sum_{z=1}^{M} \sum_{k=i}^{j-1} \alpha(i,k,y)\alpha(k+1,j,z)t_v(y,z) \tag{3.9}$$

The sum over k examines all split-points between i and j of breaking the string into two parts and having the nonterminal W_y derive the subword (i,k) and the nonterminal W_z derive the subword $(k+1,j)$. This quantity is computed using a dynamic programming algorithm called parsing. The computational complexity of parsing is $O(M^3L^3)$ time in the worst case where L is the length of the string and M is the number of nonterminals. Note that if the number of productions having a specific nonterminal on the lefthand side is bounded by a constant for every nonterminal then the time reduces to $O(ML^3)$ because the sums involving y and z in Equation (3.9) only sum over a constant number of non-zero probability productions. The space complexity is $O(ML^2)$. This is significantly more complex than the HMM and has important consequences for efficient implementation of SCFG parsing algorithms. This will be discussed extensively in later sections.

Decoding

The most likely or Viterbi path is defined to be the derivation, π, that maximizes the probability of the sequence

$$\text{argmax}_{\pi} P(x,\pi|G) \tag{3.10}$$

This can also be found in $O(M^3L^3)$ time where L is the length of the string and M is the number of nonterminals by modifying the inside algorithm to replaces sums with maximums as was done for the HMM forward variable.

Learning

SCFGs and HMMs have similar algorithms for parameter estimation. Instead of HMM's forward and backward variables, SCFGs have inside and outside variables. The outside probability, $\beta(i,j,v)$ is the probability that starting from the start nonterminal the nonterminal v is generated and the string not dominated by it is $(x_1 \ldots x_{i-1})$ to the left and $(x_{j+1} \ldots x_L)$ to the right

$$\beta(i,j,v) = P(S \Rightarrow^* x_1 \ldots x_{i-1} v x_{j+1} \ldots x_L | G) \tag{3.11}$$

That outside variable can be computed in a recursive manner starting with the largest excluded subsequence

$$\beta(i,j,v) = \sum_y \sum_z \sum_{k=1}^{i-1} \alpha(k, i-1, z)\beta(k,j,y)t_y(z,v) + \sum_y \sum_z \sum_{k=j+1}^{L} \alpha(j+1,k,z)\beta(i,k,y)t_y(v,z) \tag{3.12}$$

Initially $\beta(1,L,1) = 1$ and $\beta(1,L,v) = 0$ for $v = 2 \ldots M$ indicating that only the start symbol dominates the entire derivation. The first triple sum examines possible ways of the derivation deriving W_y, W_y deriving $W_z W_v$, W_v deriving the subword (i,j) and W_z deriving $(k, i-1)$. The second triple sum examines possible ways of the derivation deriving W_y, W_y deriving $W_v W_z$, W_v deriving the subword (i,j) and W_z deriving $(j+1,k)$.

The probability that a nonterminal v derives the subword (i,j) is given as

$$\alpha(i,j,v)\beta(i,j,v)/P(x|G) \tag{3.13}$$

In HMMs, the Baum–Welsh algorithm relies on computing *expectations* over all values of the missing variables. Similarly, SCFGs use inside/outside variables that compute expectations by summing over all possible values. These expectations are then used in estimation formulae in the general EM framework. However sometimes the most likely or Viterbi path is of more interest than these expectations using sums over all paths. This might be true because only the most likely alignments are being considered or perhaps the algorithm for computing the most likely path is more tractable than summing over all paths. In any case, the counts from a Viterbi path can be used in a general EM framework just as expected counts are used in the Baum–Welsh algorithm. The EM theory does not treat the case of computing Viterbi paths because the theory relies on computing full expectations. However, using Viterbi paths does well in practice and there are supporting arguments for this method [Neal and Hinton, 1993]. Theoretical bounds and empirical results have been computed relating Viterbi to inside/outside training for stochastic context-free grammars and have shown that the two methods are similar [Sańchez et al., 1996]. The system developed in this work uses Viterbi training instead of the full inside/outside calculations.

3.3.3 Biological Sequence Modeling Using SCFGs

Pairwise Modeling

The primary application of SCFGs is RNA modeling. Modeling RNA involves modeling the primary sequence structure as well as the secondary structure. The primary sequence is defined by the left-to-right (5' to 3') ordering of ribonucleic acids that make up the RNA sequence. The chemical interactions involved in the primary sequence structure are the strong covalent bonds that hold together the string of ribonucleic acids. The secondary structure is defined by the set of pairs of ribonucleic acids in the primary sequence that interact with each other. These interactions usually involve Watson-Crick or wobble basepairing and occur through the formation of weaker hydrogen bonds between interacting ribonucleic acids. The specification of the secondary structure describes helices (regions of the sequence that basepair with each other) and loops (regions of the sequence that do not basepair). SCFGs model loop positions by a multinomial distribution over the ribonucleic acids and helix positions by a joint distribution over pairs of ribonucleic acids. In this way basepairing positions are characterized by their full pairwise joint probability.

A correspondence can be made between a SCFG derivation and RNA secondary structure [Sakakibara et al., 1994a]. See Figure 3.2. As the figure shows, there are nonterminals which generate basepairs, loop positions, and branch productions. By assigning an interpretation the nonterminals, they can be thought of as insert states, match states, or delete states just as is the case in HMM biological sequence modeling.

STG SCFGs: A Generalization of Profile HMMs

Just as with HMMs, SCFGs can be used to perform database searches, construct multiple alignments, and estimate models from a set of sequences. The fundamental difference is that SCFGs model pairwise interactions. This means that multiple alignments not only give an alignment of the sequences but also a list of alignment columns that are modeled as being basepaired or having pairwise interactions.

In profile HMMs, the string of match states constructed by traversing the HMM superstate structure from left-to-right captures the essence of the modeled string because all paths through the HMM are insertions to or deletions from this platonic match state ideal. This work examines single tree grammar (STG) SCFGs that are a generalization of this type of HMM modeling. Profile HMMs are a single linear left-to-right ordering of superstates where each superstate consists of a match, insert, and delete state that operate on single bases (See Figure 3.1). STG SCFGs extend this to a single tree of superstates that allows pairwise interactions. STG SCFGs are composed of three types of superstates: loop, helix, and branch. The loop superstate models unpaired bases exactly as a profile HMM superstate does. The helix superstate is a generalization of the loop superstate to pairwise interactions. The helix superstate consists of three states: match, insert, and delete. In contrast to loop positions which output single bases, the helix match state outputs a pair of bases, the helix delete state deletes a pair, and the helix insert inserts a number of bases on the left and a number of bases on the right in a pairwise fashion. The branch superstate is used to layout the structure of the tree that defines the topology of the helix and loop regions of the structure. See Figure 3.3 for an illustration of this type of SCFG. See Section 4.2 for a more detailed explanation of the structure. STG SCFG modeling ensures that for *any* derivation of any sequence under the grammar, every superstate is visited once using either the match or delete state, the ordering of the superstates is invariant, and this ordering is defined by an in-order traversal of the grammar in which the match state of every superstate is visited. The STG grammar structure is used in construction of multiple alignments (as is done for HMMs) and in grammar constraint propagation, which will be discussed later.

SCFG modeling is strictly more powerful than HMM modeling. HMMs correspond to regular grammars, a proper subset of context-free grammars. Consider a state S in a HMM generating the symbol x with probability $b_S(x)$ and moving to state T with probability, a_{ST}. This can be coded in a grammar with the following production, $S \rightarrow xT$ having production probability $b_S(x)a_{ST}$. In this way any HMM can be encoded with a SCFG. HMMs are not a powerful enough language model to model the language of RNA basepairing structure in which the structure is balanced and complementary. The language of balanced parenthesis is context-free but not regular and consequently SCFGs can model RNA basepairing and HMMs cannot [Hopcroft and Ullman, 1979].

Structural Model Estimation

As a consequence of pairwise modeling, model estimation is made more difficult. In the case of HMMs the length of the model (or number of superstates) is the only structural model feature that can be manipulated. However SCFGs allow specification of structural pairwise interactions. There are order L^2 pairs in a string of length L. Choosing the structure of a SCFG involves specifying a subset of these L^2 pairs that should be modeled. While a large number of these pairs are illegal in the sense that a SCFG would be unable to generate them, there is still a very large number of legal pairs from which to choose.

Most solutions to finding which positions should be paired involve computing metrics like mutual information on pairs of columns in an alignment [Eddy and Durbin, 1994, Cary and Stormo, 1995, Juan and Wilson, 1999, Grate et al., 1994, Grate, 1999]. Those pairs of positions that have high mutual information are identified as candidates for pairwise modeling. This work side steps the difficultly of estimating the structure of pairwise

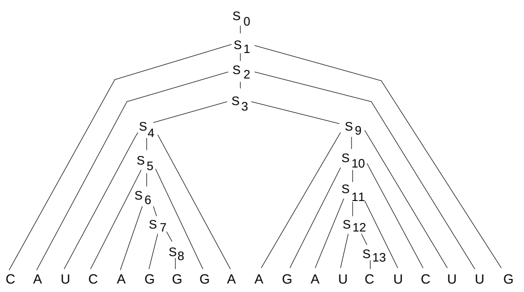
a. Productions

$$
P \;=\; \{ \quad
\begin{array}{ll}
S_0 \to S_1, & S_7 \to \mathtt{G}\ S_8, \\
S_1 \to \mathtt{C}\ S_2\ \mathtt{G}, & S_8 \to \mathtt{G}, \\
S_2 \to \mathtt{A}\ S_3\ \mathtt{U}, & S_9 \to \mathtt{A}\ S_{10}\ \mathtt{U}, \\
S_3 \to S_4\ \ S_9, & S_{10} \to \mathtt{G}\ S_{11}\ \mathtt{C}, \\
S_4 \to \mathtt{U}\ S_5\ \mathtt{A}, & S_{11} \to \mathtt{A}\ S_{12}\ \mathtt{U}, \\
S_5 \to \mathtt{C}\ S_6\ \mathtt{G}, & S_{12} \to \mathtt{U}\ S_{13}, \\
S_6 \to \mathtt{A}\ S_7, & S_{13} \to \mathtt{C}
\end{array}
\quad \}
$$

b. Derivation

$$
\begin{array}{ll}
S_0 & \Rightarrow S_1 \Rightarrow \mathtt{C}S_2\mathtt{G} \Rightarrow \mathtt{CA}S_3\mathtt{UG} \\
& \Rightarrow \mathtt{CA}S_4S_9\mathtt{UG} \Rightarrow \mathtt{CAU}S_5\mathtt{A}S_9\mathtt{UG} \\
& \Rightarrow \mathtt{CAUC}S_6\mathtt{GA}S_9\mathtt{UG} \\
& \Rightarrow \mathtt{CAUCA}S_7\mathtt{GA}S_9\mathtt{UG} \\
& \Rightarrow \mathtt{CAUCAG}S_8\mathtt{GA}S_9\mathtt{UG} \\
& \Rightarrow \mathtt{CAUCAGGGA}S_9\mathtt{UG} \\
& \Rightarrow \mathtt{CAUCAGGGAA}S_{10}\mathtt{UUG} \\
& \Rightarrow \mathtt{CAUCAGGGAAG}S_{11}\mathtt{CUUG} \\
& \Rightarrow \mathtt{CAUCAGGGAAGA}S_{12}\mathtt{UCUUG} \\
& \Rightarrow \mathtt{CAUCAGGGAAGAU}S_{13}\mathtt{UCUUG} \\
& \Rightarrow \mathtt{CAUCAGGGAAGAUCUCUUG}.
\end{array}
$$

c. Parse tree



d. Secondary Structure

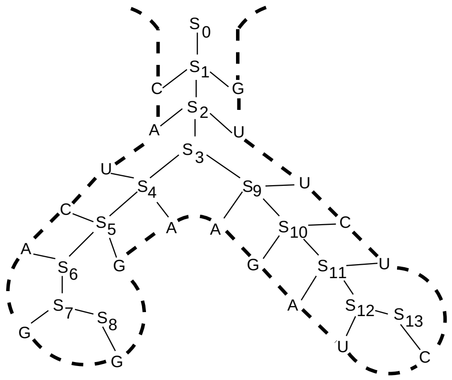


Figure 3.2: A simple context-free grammar which may be used to derive a set of RNA molecules including the specific example illustrated here, CAUCAGGGAAGAUCUCUUG. *a.* A set of productions P which generates RNA sequences with a certain restricted structure. S_0 (start symbol), $S_1, \ldots, S_{13}$ are nonterminals; A, U, G and C are terminals representing the four nucleotides. *b.* Application of the productions P could generate the given sequence by the derivation indicated. For example, if the production $S_1 \to CS_2G$ is selected, the string CS_2G replaces S_1 and the derivation step is written $S_1 \Rightarrow CS_2G$. *c.* The derivation in *b* may be arranged in a tree structure called a *parse* or *derivation tree*. *d.* The physical secondary structure of the RNA sequence is a reflection of the parse tree (or syntactic structure).

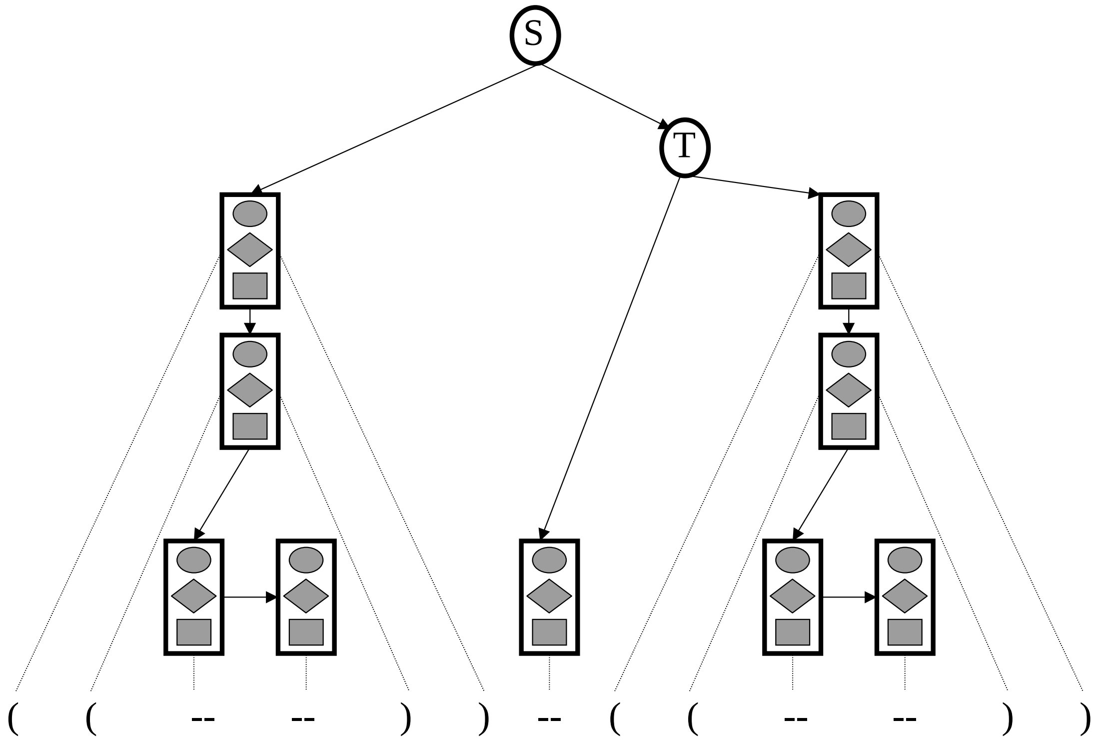


Figure 3.3: Illustration of a single tree grammar(STG). This STG grammar derives a consensus sequence with five loop positions and four helix pairs represented by "((- -)) - ((- -))" where "-" is a loop position, "(" is a lefthand side of a helix, and ")" is a righthand side of a helix. Boxes containing a circle, diamond, and square represent superstates and are able to match, insert, or delete a position in the sequence. Lines without arrowheads indicate positions in the consensus sequence that are derived by the match states. Superstates with two lines are helix superstates. Superstates with one line are loop superstates. Large ovals represent branch superstates. Arrows indicate the single tree structure. Complete transitions between all states are not shown. This structure is represented by the high level grammar, "S:> H1:(L1 H1:) T; T:> L2 H2:(L3 H2:); H1:=HELIX 2; L1:=LOOP 2; L2:=LOOP 1; H2:=HELIX 2; L3:=LOOP 2" (See Section 4.3.1).

interactions by assuming it is given in the form of a list of basepairs for one sequence in the training set. Any method can be used to estimate the pairwise structure before modeling is done with SCFGs.

Application

SCFGs have been used for modeling t-RNA [Sakakibara et al., 1994a, Eddy and Durbin, 1994], snRNAs [Underwood, 1994], a pseudoknotted biotin binder [Brown and Wilson, 1995], snoRNA [Lowe and Eddy, 1999] and small subunit ribosomal RNA (this work). SCFG methods were developed independently by Eddy and Durbin [Eddy and Durbin, 1994] and Sakakibara, Brown , Hughey, Mian, Sjölander, Underwood, and Haussler [Sakakibara et al., 1994a].

Because of their computational complexity, SCFG modeling packages are not widely available or supported.

Thermodynamic Folding of RNA

Thermodynamic-based algorithmic folding of RNA is related to SCFG modeling [Jaeger et al., 1990, Zuker, 1989, Zuker and Sankoff, 1984, Zuker and Stiegler, 1981, Mathews et al., 1999, Zuker et al., 1999]. This thermodynamic method uses a dynamic programming algorithm to find the minimal free energy structure of an RNA sequence by examining different structures composed of basepaired, stacking basepaired, bulged, looped, and multi-branched substructures. The energies of these substructures are measured experimentally. The free energy of a complete structure is the sum of the free energies of its substructures. A dynamic program explores this structure space to find the optimal free energy.

This dynamic programming algorithm corresponds to a context-free grammar with a relatively small number of states [Durbin et al., 1998]. The parameters of the grammar are derived from the free energies that are measured from physical experiments. The structure of the grammar corresponds to the different substructures that can be used in composing the final structure. The relatively small number of states used by the thermodynamic method contrasts to the SCFG method developed in this work which can have a very large number of states. The thermodynamic method tries to capture rules of RNA folding and has only a few "universal" rules that apply to RNA folding in general. The SCFG method developed in this work essentially models each position in a specific RNA structure as having its own rule as manifested through the distribution of bases in a structural multiple alignment column. Consequently the thermodynamic model has a small number of states and tries to capture RNA folding in general while the SCFG method developed here can have a large number of states and tries to capture the intricacies of the specific molecule for which the SCFG was intended to model.

Most of the SCFG algorithm enhancements described later in the text most likely would not work well for the thermodynamic models. The algorithm enhancements described later rely on the fact that a SCFG model state can be mapped to a single position in the sequence with high probability. Because the thermodynamic models try to capture RNA folding in general with a small number of states, each state accounts for many positions in the model. The thermodynamic model state representing basepairs will map to all positions in the sequence that are basepaired, for example. Consequently a single high confidence mapping from single model state to single sequence position usually cannot be made for thermodynamic models.

3.4 Parsing Stochastic Context-Free Grammars

Parsing is the mechanism for determining the probability of a sequence given a stochastic context-free grammar model and is the essential operation in SCFG modeling.

3.4.1 Parsing's Worst Case Time Complexity is $O(L^3)$

The CYK algorithm exemplifies most of the issues in parsing [Hopcroft and Ullman, 1979]. The CYK algorithm uses a dynamic programming table, $T[s, i, j]$, that stores the probability of nonterminal s deriving the subsequence, $x_i \ldots x_j$, where x is the sequence being parsed. The size of this table is $NL(L-1)/2$ where N is the number of nonterminals and L is the length of the string. The CYK dynamic programming algorithm can be used to find the probability of the start symbol deriving the entire string by filling out this table. This is done in a bottom-up fashion by first solving for all the probabilities for subwords of length one. It then uses these results to find the probabilities for subwords of length two. This continues until the probability of the entire sequence is found. For a fixed grammar, the time complexity of this dynamic program is $O(L^3)$ where L is the length of the string [Hopcroft and Ullman, 1979]. The size of dynamic programming table is $O(NL^2)$ entries, which can be quite large. For small subunit ribosomal RNA having average length about 2000 bases, a full SCFG model with 8000 states will store about 16 billion dynamic programming table entries when parsing using the CYK algorithm. This is not feasible for today's computers.

Other parsing methods that have been extended to stochastic context-free grammars such as Early parsing [Stolcke, 1995] and generalized LR(k) parsing [Tomita, 1991] are able to exploit the structure of the grammar to parse faster. However the worst-case time complexity for parsing arbitrary grammars with these methods is still $O(L^3)$.

There is correspondence between parsing and matrix operations based on associativity and communitivity properties [Harrison, 1978]. Algorithms can reduce the complexity of matrix operations from $O(n^3)$ to $O(n^{3-\epsilon})$. Strassen showed that products of 2x2 matrices can be computed in seven multiplies and this result can be recursively extended to perform matrix multiplies in $O(n^{2.807})$ time [Strassen, 1969]. The best currently known time complexity for matrix multiplication is $O(n^{2.376})$ [Coppersmith and Winograd, 1990]. Therefore, by solving the parsing problem using matrix operations, it can be shown that the true worst case complexity for parsing is less than $O(n^3)$ [Valiant, 1975]. However, the constant factors in these methods can be large and can diminish performance significantly. Also the conversion of an arbitrary context-free grammar into the matrix operation framework can be cumbersome.

3.4.2 Top-down Parsing

One of the simplest parsing techniques is top-down parsing. Top-down parsing computes the probability that the nonterminal S derives the subword (i, j) by constructing queries of the form (S, i, j). Top-down parsing is a simple tree search technique like depth-first search. It starts by querying the start nonterminal to derive the examined string. This query generates several other queries that are constructed from the set of all grammar productions that have the start symbol on the left hand side. In a recursive fashion these queries generate other queries that depend on the structure of the grammar. Once a query has been solved by reaching a terminating condition such as deriving the null string, that recursion ends and its probability value is stored in a memo table so that its value can be recalled in constant time. This is important because if that query is made again the table lookup saves the computation needed to compute it. Top-down parsing with memoization has the same worst-case running time as the CYK algorithm, $O(L^3)$ where L is the length of the string and the grammar is fixed.

In order to use a top-down parser, there must be a restriction on the grammar to ensure that top-down parsing terminates: the grammar must not contain any null cycles. Null cycles are cycles in the grammar that do not increase the length of the derived string. This restriction ensures that all explored paths eventually increase the length of the string. Because the length of the examined string is finite and any explored path must increase the derived string, top-down parsing is guaranteed to terminate. Without this restriction the top-down parser would be able to loop infinitely. Null cycles can be removed from stochastic context-free grammars preserving the probabilistic structure of the grammar [Stolcke, 1995].

3.4.3 Probabilistic Search

Artificial Intelligence search methods such as best-first and A^* can be used to find optimal parses in a top-down parsing framework [Bender, 1996]. As with any search strategy, the evaluation function must be good in order to find the solution quickly.

There has been some work in which knowledge about the probabilities of the SCFG is used to analyze the computation of parsing through island parsing [Corazza et al., 1994, Corazza et al., 1991]. In island parsing the problem domain is natural language and the problem is to compute the most likely sentence derivation given that only some words are known to occur in the sentence. These known words form islands and island parsing tries to efficiently use knowledge of these islands and the probabilistic structure of the grammar to construct the most likely sentence derivation around these islands. Key to the computation of island parsing are the off-line computations of such quantities as the upper-bound on the probability that a derivation tree rooted with nonterminal H_i yields a string of length n and upper-bounds on island probabilities or that a derivation tree is rooted with nonterminal H_i and whose yield is H_j surrounded by two terminal strings to the left and right.

A best-first branch-and-bound top-down parser was implemented for stochastic context-free grammars that use the probabilistic structure of the grammar. A best-first parser is similar to a top-down parser (Section 3.4.2), but employs a priority queue to *prioritize* the queries that it makes based on an admissible heuristic that is computed from upper-bounds. The method computes upper-bounds based on the grammar alone and uses these bounds to perform a best-first search in order to find the best parse when presented with a string. The upper-bound, U is the maximum probability of a nonterminal, T, deriving a string of length l,

$$U(T,l) = \max_{w:\ \text{string } w \text{ has length } l} \text{Prob}(T \rightarrow w)$$

Upper-bounds are computed for every nonterminal and every length between zero and some maximum length. These bounds can be computed efficiently in a bottom up fashion using an $O(L^2)$ algorithm where L is the maximum length. The algorithm simply returns the maximum probability choice at every point. The upper bounds are used as an admissible heuristic to guide a best-first search in order to find the best parse. This heuristic is admissible because the values are upper bounds and always over-estimate the true probability of the query.

The best-first parser makes a distinction between estimated probabilities which are based on the admissible heuristic and computed probabilities which are true probabilities that result from terminating conditions on queries. The best-first parser branch and bounds the queries using a query of the form $(S, i, j, bound)$ which represents the probability that S derives the subword (i, j) whose probability is greater than *bound*. Therefore if the estimated probability of a derivation falls below the *bound* then the recursion is terminated, the current best estimate of the probability is stored, and control is returned to the point in the recursion that generated the *bound* so that alternate queries with higher upper-bound probability estimates will be explored. This is a best-first search with an admissible heuristic. As with top-down parsing, this recursion will terminate. However unlike top-down parsing it is possible that not all queries will be explored. Because the admissible heuristic is used to prioritize the query process, some queries whose estimated probabilities based on upper-bounds are less than the true probability will never be explored and consequently will never be stored in the dynamic programming table.

The best-first parser was applied to t-RNA parsing. Parses for a mitochondrial t-RNA sequences under a t-RNA grammar(MY10CY10) [Sakakibara et al., 1994a] were computed. The number of stored dynamic programming (DP) entries for the best-first parser was computed and compared to the the number of DP entries required by a CYK parser (Section 3.4.1). The CYK DP table size serves as an upper bound. The best-first parser requires a DP table whose size is 85% of the CYK algorithm's DP table size. This is an improvement but is still not good enough to handle structures such as small subunit ribosomal RNA where a 15% improvement still leads to infeasible computing resources.

One possible improvement is the computation of the upper-bounds. The bounds based on length of string might be too loose and better bounds might solve the problem. To test this, a new upper bound was constructed. The new upper bound was equal to $(\alpha T + (1-\alpha)L)$ where T is the true computed probability of the query and L is the upper bound based on length as described above. An α equal to zero corresponds exactly to the experiment described above and α equal to one corresponds to a bound that has perfect information. When α was set to one, the best-first DP table size was 0.017% of the CYK DP table size. The best-first parser simply had to check that the upper bounds were indeed correct and requires only minimal storage. However when α was set to a little less than one, 0.99, performance degraded significantly. In this case the best-first DP table size was 20% of the CYK DP table size. Thus only when the upper-bounds are perfect does best-first parsing attain the performance that is required for parsing large strings with large grammars.

Other improvements such as adding a factor to the bound so that recursion is not terminated until the estimate is less than a multiplicative factor of *bound* instead of simply *bound*, changing the structure of the grammar so there are as few branch productions ($S \to TU$) as possible, or computing lower bounds for subwords (by taking the computed upper bound derivation for a string of length n and substituting the frontier of that derivation with the n long subword being examined by the query) and terminating the query and returning the lower bound if the upper bound was within ϵ of the lower bound did not lead to any significant performance enhancements.

Thus practical work has shown that a branch-and-bound top-down parser using length-based upper-bounds as an admissible heuristic does not prune the search space efficiently enough to handle large structures. The best-first parser with bounds based on length of string saves about 15% of the memory that would be needed by a CYK parser. Improving the bounds will most likely not help significantly because any bound even slightly different from perfect information will lead to performance that is only slightly better. Therefore best-first parsing is impractical when parsing large structures. To efficiently prune the space, the probabilistic structure of the grammar must have features that are derived with very high probability. If a certain feature *must* be derived in the string, then much of the space can be pruned. Unfortunately, SCFGs usually allow *any* deviation with some probability. This generality gives HMMs and SCFGs the ability to find remote homologs [Park et al., 1998]. The generality that SCFGs allow in modeling disallows the strong pruning which would allow them to run efficiently using best-first branch-and-bound top-down parsing methods.

3.5 Parsing Constraints and the Importance of Additional Information

Although computation using SCFGs is efficient in the sense that it is a polynomial-time algorithm, using SCFGs for large models and large database searches becomes infeasible because of the computational demands that are required. Even modeling structures, such as t-RNA, that are fairly short (about 76 bases) has high computational costs when performing large database searches. Sean Eddy estimated that it would take at least nine years of computer time to find t-RNAs in the human genome using SCFGs [Lowe and Eddy, 1997]. Parsing larger structures such as small subunit ribosomal RNA with length of approximately 2000 bases with a full stochastic context-free grammar requires a conservative estimate of 16 billion bytes of memory simply to hold the dynamic programming tables. This is not feasible for today's computers.

Previous approaches to solve this problem were to preprocess the data with a simple filter. The program tRNAscan-SE uses this method when it searches for t-RNAs [Lowe and Eddy, 1997]. It should be noted that in this method the preprocess does not influence the SCFG's work. It simply presents likely strings to the SCFG but does not change the computation done by the SCFG on this string. This method works for small structures like t-RNA in which the preprocess simply eliminates much of the database and allows only likely sections to be analyzed by the SCFG. It fails for larger structures like SSU rRNA, because the computational requirements for analyzing even a single SSU rRNA structure are impractically large.

Context-free grammar parsers have a worst-case time complexity of $O(L^3)$ when required to parse a string with an arbitrary context-free grammar where L is the length of the string and the size of the grammar is not dependent on the length of string (Section 3.4.1). This running time comes from the fact that parsing algorithms must examine all possible ways of each nonterminal deriving each subword and then use this information in a dynamic programming algorithm to construct the best possible way of the start symbol deriving the terminal string.

Parsing a string of length L given a grammar can be made more efficient through the use of constraints. This additional knowledge, in the form of constraints, allows the parsing problem to be carried out on subproblems of length m that are smaller than the original L length problem. Because time and space complexities for parsing are greater than $O(L)$ this leads to a speed up. Constraints fall naturally in the context-free grammar framework because they provide a hierarchical decomposition of the pattern they are describing.

A SCFG constraint is defined to be 5-tuple: (S, a, b, c, d) where S is a nonterminal and (a, b, c, d) are indices in which $1 \leq a \leq b \leq L$ and $1 \leq c \leq d \leq L$ where L is the length of the string being parsed. The constraint is interpreted to mean that if the nonterminal S is used in the most likely parse of the sequence then S must derive a substring (i, j) in which $a \leq i \leq b$ and $c \leq j \leq d$. The zero-knowledge constraint, $(S, 1, L, 1, L)$, enforces that S can derive any substring beginning and ending anywhere in the string of length L. The four indices allow uncertainty in the range that the nonterminal derives.

SCFG grammars are usually highly ambiguous in a formal language sense and constraints that are given at the start of the parsing operation limit the ambiguity by disallowing certain derivations that conflict with the given constraints. This will be shown algorithmically in the next section.

In biological modeling, constraints allow additional knowledge to be incorporated into the parse. If, for example, a basepair is known to form between bases 9 and 25 at a certain position in a helix then a constraint can be formulated for the nonterminal that outputs that basepair: $(helixNT, 9, 9, 25, 25)$. There is no uncertainty for this nonterminal, $helixNT$, given this constraint because the constraint specifies the begin and end ranges as having length exactly equal to one.

Constraints of another type have been analyzed in the case of bracketed strings [Pereira and Schabes, 1992]. In this framework, a bracket of a string of length L is a set of spans (i, j) where $1 \leq i \leq j \leq L$ which represent the substring w_{i+1} to w_j. The spans in the bracket do not overlap. In the estimation of inside and outside, all terms not consistent with the bracket are discarded. The brackets delineate where nonterminals can begin and end their derivations without labeling the nonterminal. For example $w = aabb$ and the bracket $(0, 2), (2, 4)$ represents the bracketing $(aa)(bb)$. Any nonterminal which tried to derive ab would be discarded because it would violate the bracketing. Bracketing was shown to lead to faster convergence and a better modeling of hierarchical structure for several datasets. This bracket formulation of constraints is less specific than SCFG constraints because it does not allow assignment of a nonterminal to a bracket. Also, uncertainty in the position of the bracket is not allowed.

3.6 Top-down Parsing With Constraints

The top-down parser takes a grammar, a string, and a list of constraints and finds the most likely parse of the string that satisfies the constraints given the grammar.

A top-down parsing algorithm is exemplified in Figure 3.4. Initially, the call $\text{prob}(S, 1, L)$ is made where S is the grammar's start symbol and L is the length of the string. Recursion terminates when $\text{prob}(A, j+1, j)$

```
prob(N, i, j)
{
   // returns the Viterbi inside probability of the nonterminal N
   // deriving the subword i,j

   if (i,j) violates N's constraints then return 0;

   if (i,j) stored in table then return that result;

   m = maximum over all productions Q with N on the left hand side
       of (probability of production Q times the probability of Q
       deriving the subword (i,j))

   table(N,i,j) = m;
   return m;
}
```

Figure 3.4: Pseudocode for a probabilistic top-down parser. The `table` structure stores previously computed results. If constraints are violated by the query, then a probability of zero is returned. The query subqueries all possible productions with `N` on the lefthand side to find the maximum.

is made for any nonterminal A. This call is equal to the probability that α derives the null string λ and is initially computed from the grammar using a depth-first search.

The different productions, `Q`, are the righthand sides of grammar productions and represent the different ways of deriving single positions, paired positions, etc. with their associated probability. See Section 4.3.2 for a description of the different productions for the RNACAD system. The different productions add a cost to the parse and then recursively descend. Certain productions, like $S \rightarrow TU$, iterate through many different recursions and add computational time complexity.

Constraints are used to reduce work by not exploring all queries that might possibly be generated by the parsing process. If a top-down query requires the computation of a nonterminal deriving a subword (i, j) and (i, j) conflicts with the nonterminal's given constraint then that query can be terminated and a probability of zero returned. Of all the subword queries that would be generated for the nonterminal S by the parser, only those lying within the constraints are allowed. Constraints also limit the amount of memory needed to store results. With no constraints, a nonterminal must store results for all subwords of the string having length L yielding $O(L^2)$ entries. In a constrained parse, only the space dictated by the constraints needs to be allocated. Therefore the space required for a nonterminal with no constraints is $[(L-1+1)(L-1+1)]$ while the space required for a nonterminal with constraints (a, b, c, d) is $[(b-a+1)(d-c+1)]$ which can be significantly smaller.

In order to perform top-down parsing with constraints, two preprocessing events are executed. First, as mentioned before, the most probable path of each nonterminal deriving the null string is calculated. This is computed using a simple depth-first search which runs in time linear in the size of the grammar. The second preprocessing event is constraint propagation. Here information from given constraints is propagated to all other nonterminals in order to make the parser more efficient.

3.7 Constraint Propagation

3.7.1 Motivation

Constraints reduce the time and space complexities of a top-down parsing algorithm. It is, however, possible that a top-down parser with constraints still generates queries that are illegal given the constraints and therefore the parser is not as efficient as possible. This happens because not all nonterminals have given constraints at the start of the parsing process. A simple example is the grammar having the productions: $S \to AB$ and $A \to C$, with given constraints on C. In a top-down parser, the nonterminal S tries all possible split points of the string into A and B. Some of these split points are illegal given the constraints because A directly derives C and C has given constraints that might disallow the derivations generated by S. These illegal queries keep the parser from being as efficient as possible.

Constraint propagation is the process of propagating information from nonterminals with given constraints to other nonterminals so parsing can be made move efficient. Constraint propagation can save time because certain queries will never be generated in the top-down parsing process. In the above example, after constraint propagation is performed, the parser will never generate queries on A that are later found to conflict with C's constraints. Constraint propagation can also save space. In the above example, the dynamic programming table for A can be made smaller than a full CYK table even though constraints are not given directly for A because constraints dictate that certain conflicting queries have probability zero and therefore do not need to be stored. Top-down parsing with given constraints can be done without constraint propagation, but this misses much of the constraint information implied by the hierarchical structure of the grammar.

Constraint propagation can be done efficiently in SCFGs. Strings are one-dimensional objects that define a total order on the string's terminals. Nonterminals derive these ordered terminals in a grammar derivation of the string. Given the structure of a grammar, certain hierarchical relationships exist between grammar nonterminals. These hierarchical relationships can be used to propagate constraint information from one nonterminal to other hierarchically related nonterminals. Thus constraint propagation takes a set of constraints and a grammar and propagates constraint information given the hierarchical relationships that exist between nonterminals.

3.7.2 Correct Constraint Propagation

In order to discuss constraint propagation we must rigorously define what it means to correctly propagate SCFG constraints. A Constraint Propagation Algorithm (CPA) takes a grammar, G, a string X, and a constraint instance, I_1, and produces a final constraint instance, I_*, in which constraint information is propagated to all nonterminals. A constraint instance is a set of constraints, one for each nonterminal in the grammar. Nonterminals that do not have a given constraint start with the zero-knowledge constraint $(1, L, 1, L)$ that indicates that the nonterminal can begin and end its derivation anywhere in the string.

Definition 1 *a* legal derivation *of a string x given a grammar G is a sequence of k applications of grammar productions $P_i \in G$'s productions starting with G's start symbol and ending with the terminal string x.*

Definition 2 *a* constraint-legal derivation *of a string x given a grammar G and a constraint instance I is a legal derivation that also satisfies the constraints, so that, if in the legal derivation nonterminal S derives the subword (i, j), then the constraint for S allows that derivation.*

Definition 3 *Two constraint instances, I and J, are* consistent *for a grammar G and string X if for any legal derivation, π: π is constraint-legal given G and I if and only if π is constraint-legal given G and J.*

Two constraint instances are consistent if the set of legal derivations under the two constraint instances are the same. The consistency condition will be used to ensure that the space of legal derivations is not changed by constraint propagation.

Definition 4 *The* size of a constraint instance*, I, is the sum of* $[(b_i - a_i + 1)(d_i - c_i + 1)]$ *over all constraints where the constraint for the ith nonterminal is given by* (a_i, b_i, c_i, d_i)*.*

The size of a constraint instance is how many entries must be stored in the memo table of a top-down parser.

Definition 5 *A constraint instance I is* contradictory *given a grammar G and string x if there exists a legal derivation of x given G but there does not exist a constraint-legal derivation of x given G and I.*

The constraint propagation algorithm is correct if:

1. The initial, I_1, and final constraints, I_*, are consistent.
2. The size of the final constraint instance is less than or equal to the size of the initial constraint instance.
3. Contradictions in the initial constraint instance for strings known to be derived using the grammar are identified as errors in the constraints before parsing begins.

3.7.3 The linearCP Algorithm for Constraint Propagation In STG Grammars

I give a simple algorithm named linearCP for performing constraint propagation in STG SCFGs. STG SCFGs (Section 3.3.3) are a restricted form of SCFGs that are composed of a number of superstates arranged in a single tree. Each superstate consists of a match state, an insert state, and a delete state. A superstate models either singleton loop positions or basepaired helix positions. A property of STG SCFGs is that for any derivation of any sequence, every superstate is visited exactly once using either the match or delete state, the ordering of the superstates is invariant, and this ordering is defined by an in-order traversal of the grammar in which the match state of every superstate is visited. This is important because constraint propagation relies on knowing the hierarchical relationships of nonterminals within the grammar.

The structure of a STG SCFG enforces that there are only four legal relations between the subword (i, j) derived by a match-state nonterminal S and the subword (k, l) derived by a match-state nonterminal T in any legal parse under a STG SCFG. The first relation is (i, j) is to the right of (k, l). This means that $j \leq k$. The second relation is (i, j) is to the left of (k, l). This means that $l \leq i$. The third relation is (i, j) contains (k, l). This means that $i \leq k$ and $l \leq j$. The fourth relation is (i, j) is dominated by (k, l). This means that $k \leq i$ and $j \leq l$. All other relations are not legal in a STG SCFG. The STG structure ensures that for any parse of any sequence, two match-state nonterminals present in the parse will always have the same relation to each other.

Three recursions are performed for constraint propagation in STG SCFGs that each run in linear time with respect to the number of match-state nonterminals. The algorithm starts with given constraints for certain match-state nonterminals and the zero-knowledge constraint for all other nonterminals that do not have given constraints. The algorithm then propagates constraint information to all match-state nonterminals using the three recursions. At the end of the recursions, constraint information is copied to the insert and delete states according to their relative position to neighboring match states. In the following description of the algorithm, let the constraints for a specific match-state nonterminal be identified by (a, b, c, d).

The first recursion maintains a runner variable that indicates the maximal index allowed by the constraints for which the closest hierarchically related nonterminal was able to start its begin range as indicated by a or start its end range as indicated by c. The recursion updates this runner variable as it visits nonterminals in the SCFG using an in-order traversal of the STG tree. See Figure 3.5 for pseudocode of this algorithm.

```
left_prop(runner)
{
   // the current constraints for this nonterminal are (A,B,C,D).

   // Find and set the maximum of the runner and A
   runner = max (runner,A);
   A = runner;

   // Recursively descend the tree using the nonterminal
   // production of the form S->TU.
   // Proceed left to right calling T first and then U
   if (the current nonterminal has children){
      call T.left_prop(runner);
      call U.left_prop(runner);
   }

   // Find and set the maximum of the runner and C
   runner = max (runner,C);
   C = runner;
}
```

Figure 3.5: The first recursion of the linearCP constraint propagation algorithm. The initial call is the start nonterminal's left_prop(1) indicating that the start nonterminal must start its derivation at the first terminal symbol. The recursion maximizes the A and C constraint indices as dictated by the constraints and the hierarchical relationships between nonterminals in a left-to-right manner.

The initial call to this recursion starts with the runner variable equal to one indicating that the start symbol must derive the sequence starting with the first terminal symbol. Subsequent recursions take the maximum of the runner variable and the a value of the current nonterminal's constraint, recursively descend the tree, and then take the maximum of the runner variable and the c value of the current nonterminal's constraint. This recursion occurs left-to-right.

The second recursion is similar to the first except that the runner variable for this recursion indicates the minimal index allowed by the constraints for which the closest hierarchically related nonterminal was able to finish its end range as indicated by d or finish its begin range as indicated by b. Nonterminals are visited in an in-order traversal of the STG tree. See Figure 3.6 for pseudocode of this algorithm. The initial call to this recursion starts with the runner variable equal to L indicating that the start symbol must derive the sequence ending with the last terminal symbol. Subsequent recursions take the minimum of the runner variable and the d value of the current nonterminal's constraint, recursively descend the tree, and then take the minimum of the runner variable and the b value of the current nonterminal's constraint. This recursion occurs right-to-left.

Finally, the third and last recursion simply checks for valid ranges, *i.e.* that $a \leq b$ and $c \leq d$ for every nonterminal's constraint. Initial given constraints that are contradictory will be identified in this last stage by invalid ranges.

```
right_prop(runner)
{
   // the current constraints for this nonterminal are (A,B,C,D).

   // Find and set the minimal of the runner and D
   runner = min (runner,D);
   D = runner;

   // Recursively descend the tree using the nonterminal
   // production of the form S->TU.
   // Proceed right to left calling U first and then T
   if (the current nonterminal has children){
      call U.right_prop(runner);
      call T.right_prop(runner);
   }

   // Find and set the minimum of the runner and B
   runner = min (runner,B);
   B = runner;
}
```

Figure 3.6: The second recursion of the linearCP constraint propagation algorithm. The initial call is the start nonterminal's right_prop(L) indicating that the start nonterminal must end its derivation at the last terminal symbol. The recursion minimizes the D and B constraint indices as dictated by the constrains and the hierarchical relationships between nonterminals in a right-to-left manner.

3.7.4 Correctness of linearCP

In order to understand the linearCP algorithm, the hierarchical relationships between nonterminals must be defined in more detail. We introduce the structure string to help identify these relationships.

Definition 6 *By F_i we denote the* structure sentential form *of the derived string in a legal derivation after i applications of grammar productions. To define the structure sentential form we must introduce new symbols to the alphabet. If A is a nonterminal then the new structure symbols $\dot{A}$ and $\ddot{A}$ are introduced. Let Γ denote the alphabet of these new structure symbols. F_i is defined recursively:*

Initially:

$F_0 = S$ *where S is the grammar's start symbol*

Recursion:

Let $F_i = \alpha A \beta$ where A is a nonterminal and α, β are in $\{V \cup \Gamma\}^*$

If the applied production is of the form $A \rightarrow BC$ then $F_{i+1} = \alpha \dot{A} BC \ddot{A} \beta$

If the applied production is of the form $A \rightarrow a$ then $F_{i+1} = \alpha \dot{A} \ddot{A} \beta$

Termination:

Termination occurs when the derived string contains no nonterminals. This is the final structure sentential form *or* structure string.

The structure string delineates, for a specific derivation, the hierarchical relationships of every nonterminal used in the derivation. This definition is given for grammars in Chomsky Normal Form but is easily extended to other forms.

Definition 7 *The* canonical structure string *for a STG SCFG is defined to be the structure string constructed from the derivation following the STG tree that uses only match states and no insert or delete states.*

We now define an ordering relation, $<^*$, on the structure symbols, Γ.

Definition 8 *For any $L, R \in \Gamma$ we define $L <^* R$ if L occurs to the left of R in the canonical structure string.*

Note that for $L, R \in \Gamma$, $L <^* R$ implies that L never occurs to the right of R in any structure string constructed from any derivation of any string under a STG SCFG.

Each structure symbol in Γ is associated with a range derived from the constraints on its nonterminal. When the nonterminal S has the constraint (S, a, b, c, d) then we will associate the range (a, b) with the structure symbol $\dot{S}$ and the range (c, d) with $\ddot{S}$.

Now consider two structure symbols, L and R, with $L <^* R$ where we do not assume that $L = \dot{S}$ and $R = \ddot{S}$ for some nonterminal S. Let the range (i, j) be associated with L and the range (k, l) be associated with R. These ranges are extracted from the given constraints.

Logically, the following relations for ranges associated with the two structure string symbols, L and R where $L <^* R$, must hold true: $(i \leq j)$, $(k \leq l)$, $(i \leq k)$, and $(j \leq l)$. The relations $i \leq j$ and $k \leq l$ are true because they represent ranges in which the starting index is less than or equal to the ending index. The relations $i \leq k$ and $j \leq l$ hold true because, given that $L <^* R$, L must always start before R. Together these four relations imply $(i \leq l)$. There is no relation between j and k because j can be less than, equal to, or greater than k and be logically consistent. If any of these relations does not hold true then there is a logical inconsistency and the constraints are contradictory. These relations are exploited in the linearCP algorithm as shown in Figure 3.7.

With the definition of the ordering relation and identification of the logical relationships that hold between structure symbols, establishing correctness of the linearCP algorithm is straightforward.

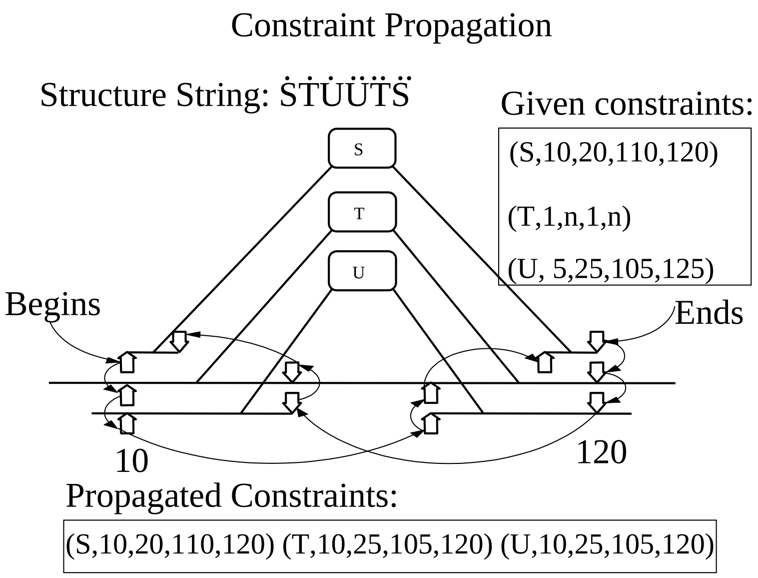


Figure 3.7: Constraint Propagation for the derivation ($S \Rightarrow \Sigma^*T\Sigma^* \Rightarrow \Sigma^*U\Sigma^*$). The structure string for this derivation is ($\dot{S}\dot{T}\dot{U}\ddot{U}\ddot{T}\ddot{S}$). The given constraints are $(S, 10, 20, 110, 120)$ and $(U, 5, 25, 105, 125)$. The nonterminal T has the zero-knowledge constraint. Constraint updates are made for the T and U nonterminals based on the given constraints and the logical relations between structure symbols given the derivation. The a value for nonterminal T can be updated from 1 to 10 in the constraint propagation because the nonterminal S starts its derivation to the left of the T (represented by the fact that the structure string contains the substring "$\dot{S}\dot{T}$") and S's derivation must start after position 10 as given by the constraint. Therefore T must begin its derivation at or after position 10.

The first condition for correctness is constraint consistency between the given constraints and the propagated constraints. At each point in the linearCP recursion, maximums and minimums of a given constraint and constraints to the immediate left and right are computed. This directly reflects the logical relationships that must hold true between structure symbols. Inconsistent constraint information is not created. If the current constraint is a maximum or minimum then its value is unchanged and therefore cannot be inconsistent. If the value is changed then the constraint from either the left or the right was stronger and more restrictive. This update is dictated by the hierarchical ordering of nonterminals and by the logical relations between structure symbols. Therefore, if the value is updated, then the constraints dictated its value and inconsistent constraint information is not created. If an update is made such that the resulting range is invalid as in the range (j, i) where $i < j$ then this reflects the fact that there is a logical inconsistency in the constraints and therefore there is a contradiction. This error will be identified in the last recursion of the linearCP algorithm.

The second condition for correctness is that the size of the propagated constraint instance must be smaller or equal to the size of the initial constraint instance. For a constraint, (a, b, c, d), the linearCP algorithm either makes a and c larger by taking maximum or b and d smaller by taking minimums. Because all indexes are positive, the size function, $(b - a + 1)(d - c + 1)$, is smaller if updates are made or equal if no updates are made.

The last condition for correctness is the identification of contradictory constraints. It is easy to see if the initial constraint instance was contradictory then the linearCP constraint propagation algorithm would cause updates that result in constraint ranges on the nonterminals to be invalid. The last recursion of the linearCP algorithm would identify these invalid ranges and signal a contradiction error in the initial constraint instance.

3.8 Constraint Generation Using Hidden Markov Models

Constraints limit the parsing computation but a wrong constraint will not allow the optimal parse to be found, because it will constrain the parser from searching the space which contains the optimal parse. A trade-off must be made between limiting computing resources (which would argue for making as many constraints as possible) and limiting the danger of supplying an incorrect constraint (which would argue for making as few constraints as possible). A method has been developed which addresses these issues in a principled way: posterior decoding of hidden Markov models. This method generates constraints that are stringent enough to reduce the search space to a manageable size, while being accurate enough that the optimal parse is contained within the constraints with high probability.

If the constraint generation procedure needed as much time and space as stochastic context-free grammars (SCFGs) then it would be useless. Unlike SCFGs which model pairwise interactions, HMMs model only linear structure. As shown in previous sections, HMMs are significantly less complex than SCFGs in both time and space. Thus, HMMs appear to be candidates for constraint generation. However one must remember when little primary sequence is conserved and the secondary structure information dominates, then all linear models including HMMs fail for constraint generation and cannot be reliably used.

The motivation for using HMMs comes from their ability to model primary sequence structure in a way identical to SCFGs. While HMMs are unable to model pairwise interactions, they are able model primary structure. Most molecules, even RNA molecules with a large number of secondary structure interactions, have some primary conservation of structure. Small subunit ribosomal RNA, for example, has regions of highly conserved primary structure across practically all divisions of life that are easily aligned using primary sequence only [O'Brien et al., 1998]. These highly conserved regions are, with little doubt, linked to structurally important blocks of SSU rRNA tertiary structure. HMMs are able to model these primary regions very well. Therefore, one can argue that if an HMM model of SSU rRNA maps a base of a sequence to an internal HMM model state with very high probability then that mapping can be believed with some confidence. This confidence can be quantified through HMM posterior decoding. Our constraint generation procedure uses an HMM approximation to the SCFG. A new string is posterior decoded using the HMM to identify high confidence base-to-HMM-state mappings. These mappings are then translated to constraints on SCFG nonterminals using a reference string.

The mapping from base x to HMM state H is made using posterior decoding of the HMM. The posterior decoding for a string x and HMM θ gives the probability that the state deriving the ith base is k,

$$P(\pi_i = k|x, \theta)$$

This value is computed efficiently by the HMM as a product of the forward and backward variables. The maximum posterior decoding is

$$\hat{\pi}_i = \text{argmax}_k P(\pi_i = k|x, \theta)$$

This maps each base, i, to its most likely posterior state, k. We then rely on the fact that a high posterior mapping implies high confidence that the mapping is correct.

The translation from HMM state to SCFG nonterminal is accomplished using a reference string. A Viterbi path is computed for the reference string under the SCFG. The reference string is a sequence with a well-known parse, such as *E.coli* SSU rRNA (Section 5). A maximum posterior decoding is computed for the reference string under an HMM approximation to the SCFG. If base x of the reference string is derived by nonterminal T in the SCFG Viterbi parse and by state H in the HMM then a mapping is made from HMM state H to SCFG nonterminal T. This technique maps HMM states used in the maximum posterior decoding to SCFG nonterminals used in the reference Viterbi parse. Alternatively, an HMM can be constructed by marginalizing a SCFG and the HMM-state-to-SCFG-nonterminal mapping could be computed directly from the construction.

Using a reference sequence to construct an HMM-state-to-SCFG-nonterminal mapping and using HMM posterior decoding to construct a base-to-HMM-state mapping gives the necessary components of a constraint generation system. There is a slight problem in that the maximum posterior decoding might give mappings that are not legal given the topology of the HMM. For example, state i might derive base 50 while state $i+1$ derives base 20 even though the HMM might disallow the state $i+1$ from deriving any base before state i. This might lead to contradictory constraints. Also constraint generation might want to enforce that the distance between any two nearest constrained bases be less than a window size, W. This might be done to strictly limit the size of the SCFG computation. These problems can be solved using a dynamic programming algorithm using the cost of matching base r to HMM state s, C_{rs}, which is given by the posterior decoding of the HMM and a cost of not matching base r to any match state, $D_r = max_s C_{rs}$. This information can be used to map each base to an HMM state or to no HMM state in such a way that the HMM states are monotone in order of the bases, the largest unconstrained gap is W, and the total cost of all matches and non matches is minimum. For an HMM with t match states and a string of length n this would lead to a dynamic programming table of size $(2tn)$. Because this dynamic programming is somewhat costly, a simple linear greedy algorithm is adopted. The algorithm greedily ensures monotonicity and minimum cost by using a sliding window size of W across the string and taking only the maximum posterior decoding state at each point that agrees with monotonicity.

With this algorithm, high-confidence mappings can be made from base to HMM state to SCFG nonterminal. If a mapping is made from base x to SCFG loop-state nonterminal T then the constraint is made: (T, x, x, x, ∞). This enforces the loop nonterminal to derive x and end the loop derivation after x. If a mapping is made from base x and base y to SCFG helix nonterminal U then the constraint is made: (U, x, x, y, y). This enforces the helix nonterminal to derive the basepair (x, y).

This method of constraint generation shares some similarities with the RNAlign program [Corpet and Michot, 1994]. This program is similar in its dynamic programming method taking into account both primary and secondary information. However the RNAlign system is not probabilistically based, is not based in formal languages, uses manually set heuristic scores, and has time and space complexities worse than SCFGs. However it does employ a method to construct a primary alignment, uses this alignment to look for runs of two or more identical aligned bases, and then uses these "strongly conserved" regions as anchor sites that constrain the computation of the full dynamic programming.

3.9 Practical Implementation

We tested the constraint generation technique using hidden Markov model posterior decoding by parsing 169 SSU rRNA sequences with an estimated SCFG and HMM using the methods described in this section. We compared the constrained dynamic programming table size to dynamic programming table size required by the CYK algorithm, which is $(N\frac{L(L-1)}{2})$, where N is the number of nonterminals and L is the length of string being parsed. The average size of the constrained tables is $0.000575(+/-0.000199)$ the size of the CYK tables. Reductions of this size allow previously impractical problems to be computed efficiently.

3.10 Conclusion and Future Work

Parsing constraints of the form (S, a, b, c, d) were presented for SCFG parsers and were shown to give significant performance enhancements. These enhancements allow previously impractical problems to be computed using SCFGs.

Recently a parser was introduced that allows some forms of RNA pseudoknot interactions [Rivas and Eddy, 1999]. This is significant because pseudoknot structures are not a context-free structure and are therefore not representable by SCFGs. However the running time of this algorithm is $O(n^6)$ and space is $O(n^4)$. This makes this algorithm applicable to only small problems. However the use of constraints in a slightly modified form to account for the presence of pseudoknots might be applied. This could lead to significant performance improvements and allow the algorithm to be applied to larger problems.

Chapter 4

The RNACAD Stochastic Context-Free Grammar Modeling System

The RNACAD system is a set of software tools that are used to model sequence structure using stochastic context-free grammars (SCFGs). This chapter gives a low-level overview of the RNACAD system to allow a concrete view of the application of SCFGs at an implementation level. The RNACAD system allows probabilistic modeling of single and pairwise interacting sequence positions with some restrictions on the topology of the the interactions. A user-friendly interface allows easy use of the system. A high-level hierarchical language representation of sequence structure is used which allows a complete and economical description. The modeling process can be constrained using biological knowledge.

The core task of the system is to find the most likely derivation of the string given a SCFG using a dynamic programming algorithm known as parsing. For large modeling problems such as small subunit ribosomal RNA that has an average length of about 2000 bases, the parsing algorithm requires computer resources that are not feasible for today's computers. A system of constraints in incorporated to prune the search space of the parsing algorithm and allow previously impractical problems to be analyzed.

4.1 Objectives

The RNACAD system consists of several methods which allow the modeling of sequence structure. The package allows the following tasks to be performed:

- **Specification of Secondary Model Structure:** The common secondary structure of a family of sequences can be specified using a high-level hierarchical modeling language. This is the start of the modeling process. It describes the general structure of the molecule. For example, t-RNA would be described as having a clover-leaf structure. Individual sequences might have deviations from this structure but the average consensus molecule is best described by this structure. This language allows specification of helices and loop regions and how they occur in a molecule. The specification can be generated automatically given a multiple alignment and a list of basepairs for a canonical sequence.

- **Sequence Classification:** A stochastic context-free grammar model can be used to determine the probability of a sequence. This probability can be used to find sequences in a database which are similar in primary and secondary structure to the sequences used to train the model.

- **Determination of a Sequence's Most Likely Structure:** The most likely secondary structure of a single sequence in a family of sequences with a common structure can be computed. The sequence will be parsed into helices and loop regions thereby specifying the secondary structure.
- **Alignment of Multiple Sequences:** A multiple alignment of sequences with a common secondary structure can be computed.
- **Learning of Model Parameters:** The parameters of a stochastic context-free grammar, a model of common secondary structure, can be optimized to fit a set of sequences.

4.2 Low-Level Overview

This section describes how the system operates at a low-level. It includes the use of constraints. For small grammars or small strings constraints are not necessary and do not have to be used.

This system begins with a **multiple alignment**(Section 4.3.9) of a set of **sequences** (Section 4.3.3) with a common secondary structure. The common secondary structure is given as a **list of basepairs** (Section 4.3.4) for a special reference sequence which is a sequence present in the multiple alignment that is a representative sequence. The system does not estimate the secondary structure, it must be given *a priori*.

From this multiple alignment a **high-level grammar**(Section 4.3.1) is generated. The high-level grammar gives the topology of the helix and loop regions and the average length of the different regions. The high-level grammar describes the one platonic ideal consensus string. All strings under this grammar are characterized as matches to, deletions from, or insertion to this platonic ideal. Non-basepaired positions are modeled either as loops or helix bulges in the model. This is determined by the information content of columns in the multiple alignment. High information causes the position to modeled as a loop. Loop positions model explicitly base identity. Low information positions cause the position to be modeled as a bulge in helix positions. Bulges only model length of the bulge not the identity of the bases. The high-level grammar is designed to be human readable. If the user wishes to enforce a certain loop be 8 positions even though the information content decrees that only 7 are needed, the user can edit the high-level grammar to enforce this. Users can also generate their own high-level grammars.

Additionally **constraints**(Section 4.3.6) on the reference sequence are generated. The grammar is generated directly from the multiple alignment. These constraints on the reference sequence enforce that the positions in the reference sequence are derived by the correct nonterminals in the grammar and consequently occupy the correct columns in a multiple alignment. This is important because without these constraints it would take infeasible computing resources to compute a parse of the reference sequence. With these constraints a parse is easily computed and this parse is used to bootstrap other constraints for other sequences.

The high-level grammar is then translated into a **low-level grammar** (Section 4.3.2). This expands high-level structures such as `LOOP`s and `HELIX`s into a set of low-level productions that characterize the match, insert, and delete structure of profile SCFG modeling. Translating a high-level grammar into a low-level grammar involves taking a high-level grammar describing how objects such as `LOOP`s and `HELIX`s are derived in relation to each other and translating it into a simple binary tree in which high-level objects are expanded into a number of productions on low-level nonterminals. This task involves converting the generic structure into a binary tree and expanding the high-level objects to low-level objects.

Translating the high-level structure to a binary tree is a simple operation commonly used in translating grammars to Chomsky normal form and involves introducing new nonterminals.

Expanding high-level objects into low-level objects is where the true probabilistic nature of the model comes into play. The expansion of a high-level object into a sequence of low-level objects defines the probability structure of the model.

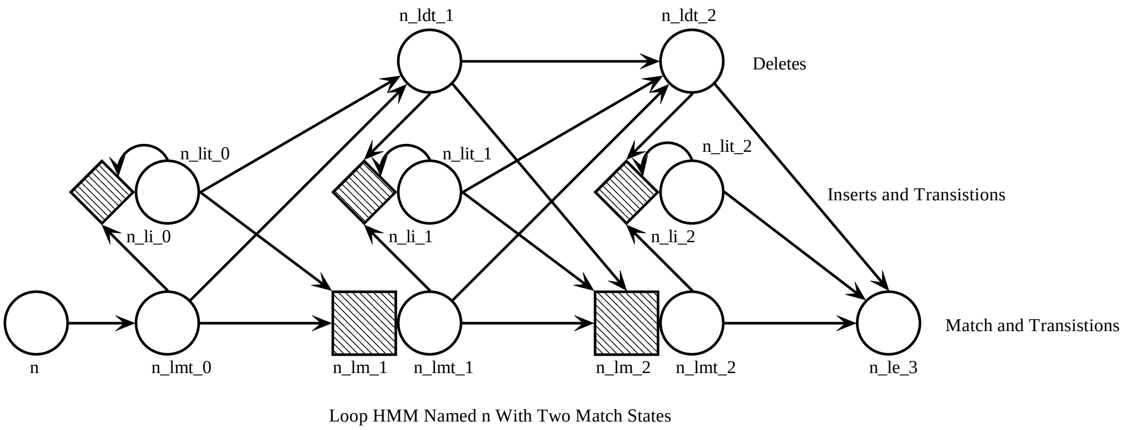


Figure 4.1: Low-level structure of a high-level LOOP 2 object. This object models two match positions in a loop. The two match positions are the two squares. These match positions can be deleted using the circles at the top of the diagram. Insertions can be made before and after every match state using the diamonds. Each one of these objects has its own nonterminal. The lines are transitions between nonterminals.

`LOOP`s are translated into a profile HMM in which there are n superstate positions that can be matched or deleted and there can be any number of inserts before or after each match state. See Figure 4.1.

`HELIX`s are similar to `LOOP`s except that match states output a pair of bases according to a joint multinomial distribution, inserts (or bulges) output a number of bases on the left and right according to a *helixinsert* probabilistic model and delete states delete a corresponding basepair in the model. See Figure 4.2.

The low-level grammar is regularized using **priors**(Section 4.3.10) yielding a correctly normalized stochastic context-free grammar. The reference sequence is then parsed using this normalized grammar and the constraints given for the reference sequence. This reference **parse**(Section 4.3.5) is important because it provides a mapping from every base in the reference sequence to a nonterminal in the grammar that generated that base in the most likely derivation. This is an SCFG-specific labeling of the reference sequence. This information allows high confidence matches to the reference string, using a method like HMM posterior decoding, to be translated into constraints for an SCFG.

Given this reference parse, constraints are then generated for all sequences in the multiple alignment. These constraints enforce that the correct nonterminals derive the correct columns in the multiple alignment. All sequences are then parsed given the constraints.

At this point each sequence has a parse. The structure of these parses is such that when a multiple alignment is generated using these parses, the original multiple alignment will be regenerated. There is a possibility that insert regions do not correspond exactly as they did before but all basepairs and high information loop positions will correspond exactly as given by the original multiple alignment. This set of parses can be used to estimate a grammar. This grammar best represents the multiple alignment information with common secondary structure in SCFG form.

In order to parse a new string an approximation to the SCFG is needed to generate constraints. The approximation can be an HMM trained on the multiple alignment. With the HMM, the SCFG parse of a new sequence first does an HMM **maximum posterior decoding** (Section 4.3.8) of the sequence and generates constraints from this. The parser then takes the string, the SCFG, and the constraints and finds the most likely derivation.

Learning of the grammar can be iterated by parsing sequences, estimating a new grammar given these new parses, and continuing the parse/estimate cycle until some stopping criterion is met.

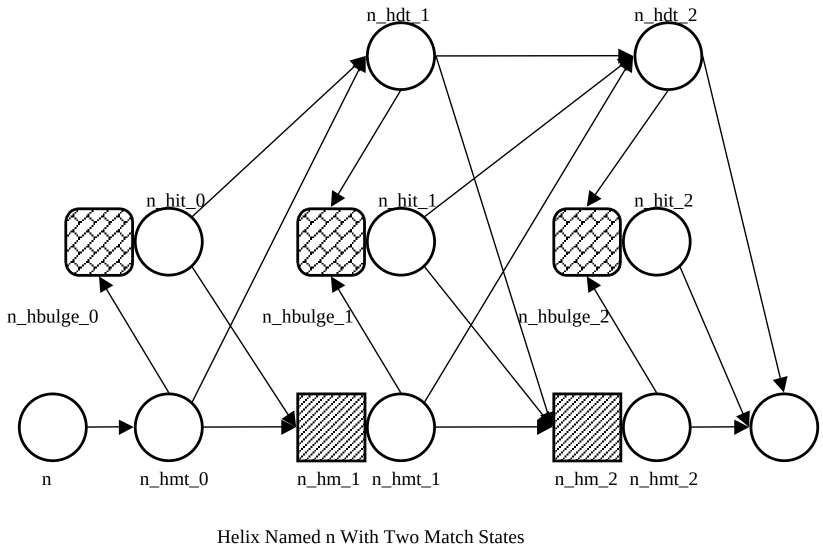


Figure 4.2: Low-level structure of a high-level HELIX 2 object. This object models two match position in a helix. The two match positions are the two squares. These match positions can be deleted using the circles at the top of the diagram. Insertions can be made before and after every match state using the rounded squares. Each one of these objects has its own nonterminal. The lines are transitions between nonterminals. The primary differences with LOOP modeling is that match and deletes are done in pairs and the structure of the insert is more complicated than a geometrically distributed loop insert.

4.3 RNACAD Data Formats

4.3.1 High-level Grammar

A *High-level Grammar* is a file that contains lines that have the following elements:

- `symbolic_name:` A name representing a subsequence of the molecule. A t-RNA might contain the name `anti-codon-loop`.
- `command:` A command is a modifier of a `symbolic_name`. The syntax is `symbolic_name:command`. The commands can be

  `(` Begin helix.

  `)` End helix.

  `>` Symbolic name derives a list of elements.

  `=` Symbolic name derives a low-level object.

- `LOOP <n>` A low-level object that describes a loop with an average of n nucleotide positions.
- `HELIX <n>` A low-level object that describes a helix with an average of n nucleotide pairs that does not allow bulges.
- `BULGEHELIX <n>` A low-level object that describes a helix with an average of n nucleotide pairs that allows bulges.
- `FIM` A low-level object that allows sequence inserts of length l with probability $(0.25)^l$.
- `NULLLOOP` A low-level object that serves as a place holder for a LOOP.
- `NULLHELIX` A low-level object that serves as a place holder for a HELIX.

For t-RNA, the secondary structure would be specified by describing the four stem-loop regions of the molecule and how long, on average, the different regions are. See Figure 4.3 for an example t-RNA high-level grammar.

4.3.2 Low-level Grammar

The low-level grammar is translated from the high-level grammar and represents the High-level Structure at the level of SCFG grammar productions. For example, a high-level object such as `LOOP 1` has the semantic interpretation of modeling a loop with one match position and is represented by several states that represent the possibilities of inserting a number of bases before the first state, matching or deleting the first state, and inserting a number of bases after the first state.

A low-level grammar is a sequence of comment and production line pairs. The comment line starts with the character `#` and has the following form: `# name:type:position`. The identifier `name` is the high-level object name associated with this production. The identifier `type` describes the type of production. For example the type `l_im_t` is a loop insert to match transition. See Figure 4.5 for a table of production types. The identifier `position` describes the place in the structure. For example the 5th match position in a `LOOP` would have a 5 as a position.

This naming convention is adopted in order to classify different productions based on their use. Because most grammars are generated from a high-level grammar, their structure is very well-defined, the same block match structures are repeated for the different positions of the structure. Different productions from

```
S:> init accept-stem:( ins1 D-dom ins2 anticodon-dom variable T-dom ins3 accept-stem:)
init:= LOOP 1
accept-stem:= HELIX 7
ins1:= LOOP 2
D-dom:> D-hel:( D-loop D-hel:)
D-hel:= HELIX 4
D-loop:= LOOP 8
ins2:= LOOP 1
anticodon-dom:> ac-hel:( ac-loop ac-hel:)
ac-hel:= HELIX 5
ac-loop:= LOOP 7
variable:= LOOP 7
T-dom:> T-hel:( T-loop T-hel:)
T-hel:= HELIX 5
T-loop:= LOOP 7
ins3:= LOOP 1
```

Figure 4.3: High-level grammar of t-RNA that describes t-RNA's cloverleaf structure. Parenthesises represent the left and right hand sides of helices. LOOP and HELIX represent a sequence of superstates that allow for matches, deletions and insertions for either single or paired bases.

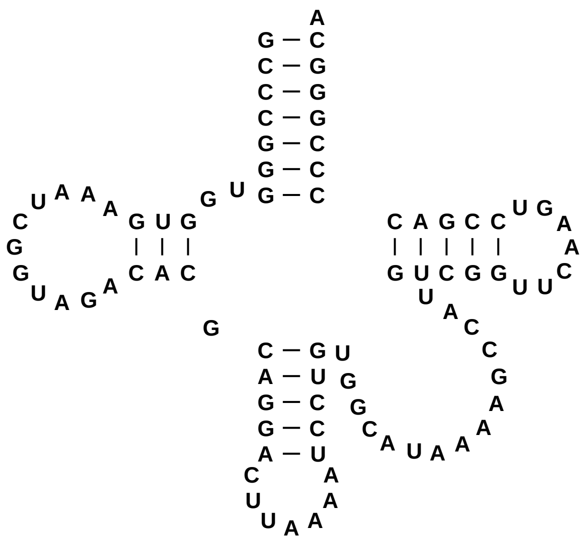


Figure 4.4: T-rna secondary structure. The structure, accept-stem, is the helix at the top. The structure D-hel is the helix at the left. The structure ac-hel is the helix at the bottom. The structure T-hel is the helix at the right. This structure is described by the high-level grammar.

these blocks are treated differently. A chain rule that moves from a match state to a match state will be interpreted differently from a chain rule that moves from a match state to a delete state even though the same production, a chain rule, is used. The comment allows the productions to be treated differently in the case of parameter estimation. The production comment allows different priors to be applied to the different types of productions indicating that the productions have different statistical behaviors.

A production has the form: `<prob> S <name> <production_type> <parameters>`. The identifier `<prob>` is a probability between 0 and 1 and represents the probability of using the production. The identifier `<name>` is the name of the state that has a naming convention similar to comments as described above. See Figure 4.5. The identifier `<production_type>` is a string describing the type of production. See below for a explanation of types. The identifier `<parameters>` are the parameters required by the production and are dependent on the `<production_type>`.

The `<production_type>`s are given below:

sequence

A binary sequence of nonterminals as in $S \to TU$. Parameters are the left nonterminal name and the right nonterminal name, `( T U )`.

In a top-down parser the **sequence** production adds complexity to the parser because it examines all possible ways of splitting the derivation onto the two nonterminals. For a string of length L this production makes $2L$ sub-queries representing L split-points. In order to reduce the complexity this production uses constraint information from the two nonterminals to make possibly less than $2L$ sub-queries. If the left nonterminal has constraints (a, b, c, d) and the right nonterminal has constraints (e, f, g, h) and the sequence production is trying to derive the subword $(begin, end)$ then the split-point, s, must follow the following relations in order to satisfy the constraints:

$$c \leq \quad s \quad \leq d \tag{4.1}$$

$$e \leq \quad s+1 \quad \leq f \tag{4.2}$$

$$begin - 1 \leq \quad s \quad \leq end \tag{4.3}$$

The set of legal split-points is precomputed for a given string so that the parser is as efficient as possible.

chain

A chain from one nonterminal to another as in $S \to T$. Parameter is the chained nonterminal name, `( T )`.

loop

An output production that derives one base on the left according to a multinomial distribution on single alphabet characters and then continues the derivation. Parameters are the chained nonterminal name and a list of four bases and associated probabilities, `( NT A 0.25 C 0.25 G 0.25 T 0.25)`.

helix

An output production that derives one pair of bases according to a multinomial distribution on pairs of alphabet characters and then continues derivation. Parameters are the chained nonterminal name a list of 16 basepairs and associated probabilities, `( NT AA 0.01 AT 0.01 ... TT 0.01 )`.

Comment	Production	Semantic Meaning	Nonterminal Name
n:branch	sequence	sequence branch point	n
n:l_b	chain	loop begin	n
n:l_i:p	loop	loop insert	n_li_p
n:l_m:p	loop	loop match	n_lm_p
n:l_e:p	lambda	loop end	n_le_p
n:l_mm_t:p	chain	loop match to match trans	n_lmt_p
n:l_md_t:p	chain	loop match to delete trans (AIL)	n_lmt_p
n:l_mi_t:p	chain	loop match to insert trans	n_lmt_p
n:l_dm_t:p	chain	loop delete to match trans	n_ldt_p
n:l_dd_t:p	chain	loop delete to delete trans (AIL)	n_ldt_p
n:l_di_t:p	chain	loop delete to insert trans	n_ldt_p
n:l_im_t:p	chain	loop insert to match trans	n_lit_p
n:l_id_t:p	chain	loop insert to delete trans (AIL)	n_lit_p
n:l_ii_t:p	chain	loop insert to insert trans	n_lit_p
h_b	chain	helix begin	n
n:h_m:p	helix	helix match	n_hm_p
n:h_bulge:p	helixinsert	helix bulge	n_hbulge_p
n:h_mm_t:p	chain	helix match to match trans	n_hmt_p
n:h_md_t:p	chain	helix match to delete trans (AIL)	n_hmt_p
n:h_mi_t:p	chain	helix match to insert trans	n_hmt_p
n:h_dm_t:p	chain	helix delete to match trans	n_hdt_p
n:h_dd_t:p	chain	helix delete to delete trans (AIL)	n_hdt_p
n:h_di_t:p	chain	helix delete to insert trans	n_hdt_p
n:h_im_t:p	chain	helix insert to match trans	n_hit_p
n:h_id_t:p	chain	helix insert to delete trans (AIL)	n_hit_p

Figure 4.5: RNACAD Naming Procedure. RNACAD grammars are very structured. This table gives the procedure for naming nonterminals and productions based on their production types and semantic meanings. The `Comment` column gives the comment in the parse associated with the production. The `Production` column gives the type of production. The `Semantic Meaning` column indicates how the production should be interpreted. The `Nonterminal Name` column gives the name of the nonterminal involved with the production. These conventions can be related to the high to low-level grammar translation. In the table, n represents a name, p represents a position, and AIL (absent in last) refers to the convention that loops and helices end in a match position and therefore do not have certain structures in the last position.

Bulge Length	Number of Bulges
1	45219
2	44973
3	42544
4	4733
5	12334
6	9812
7	4071
8	394
9	800
10	792
11	72
12	15
13	4
14	7
15	2
16	4
17	2
23	1
26	3
29	1
48	1

Table 4.1: Length Distribution on Bulge Lengths as computed from the Antwerp database of SSU rRNA. Most bulges have length three or less.

helixinsert

This production is used to model bulges in helices. The production derives a number of bases on the left, a number of bases on the right, and then continues the derivation. The production is modeled as a probabilistic mixture. One mixture component models bulges as a multinomial joint distribution on the length of the left bulge and the length of the right bulge for lengths of three or less. The other mixture component models bulges as a geometric distribution on the total length of the bulges: left bulge length plus right bulge length. This mixture model was chosen because most bulges are small and the joint distribution captures the information in these small bulges. This fact comes from examination of the Antwerp database of SSU rRNA structures which explicitly denotes secondary structure elements such as helices, loops, and bulges [de Peer et al., 1997]. See Table 4.1 for a table of bulge lengths. While there are a number of bulges having length five or more, the most frequent bulges have length three or less. Modeling bulges to length five requires more parameters than modeling to length three so bulge modeling was parameterized to length three. The geometric distribution, on the other hand, allows arbitrarily long bulges to be modeled with a single geometric parameter.

The **helixinsert** production is parameterized by 18 numbers, p_large, $p\_bulge[0..3][0..3]$ and $large\_avg$. In a RNACAD grammar, the parameters are given in the following order: avg_large, p_large, and then the indices of p_bulge with the associated probability. The parameters $p\_bulge[l][r]$ give the probability of a l length bulge on the left and a r length bulge on the right given $l \leq 3$ and $r \leq 3$. The parameter p_large is the probability of seeing a bulge that has length greater than three on either the left or right. Note that $\sum_{l=0}^{3}\sum_{r=0}^{3} p\_bulge[l][r] + p\_large = 1$. In terms of mixtures, the parameter p_large is the mixture coefficient for the geometric distribution. The parameter $large\_avg$ is the mean length of bulges that are not captured by the joint distribution (lengths greater than 3). The geometric parameter is estimated from this mean.

The cost of a bulge with l bases generated on the left and r bases generated on the right is the multinomial joint distribution if ($l \leq 3$ and $r \leq 3$)

$$p\_bulge[l][r] \cdot 1 \cdot (0.25)^{(l+r)}$$

or the geometric distribution if ($l > 3$ or $r > 3$)

$$p\_large \cdot \phi^{(l+r-1)}(1-\phi) \cdot (0.25)^{(l+r)}$$

The geometric parameter, ϕ, is related to the mean large bulge length, $large\_avg$, by

$$\frac{1}{1-\phi} = large\_avg$$

In a top-down parser the **helixinsert** production adds complexity to the parser, because it examines all possible ways of splitting the derivation by outputting l bases on the left and r bases on the right and then chaining to the next nonterminal. For a string of length L this production makes $O(L^2)$ sub-queries. In order to reduce the complexity this production uses constraint information from the chained nonterminal to make possibly less than $O(L^2)$ sub-queries. If the chained nonterminal has constraints (a, b, c, d) and the **helixinsert** production is trying to derive the subword $(begin \ldots end)$ then the number of bases derived on the left, l, and the number of bases derived on the right, r, must obey the following relations in order to satisfy the constraints:

$$\begin{aligned}
a - begin \leq & \quad l & \leq b - begin \quad & (4.4)\\
0 \leq & \quad l & \leq end - begin \quad & (4.5)\\
end - d \leq & \quad r & \leq end - c \quad & (4.6)\\
0 \leq & \quad r & \leq end - begin \quad & (4.7)\\
0 \leq & \quad (l+r) & \leq (end - begin + 1) \quad & (4.8)
\end{aligned}$$

The set of legal lengths on the left and on the right is precomputed for a given string so that the parser is as efficient as possible.

lambda

A production that derives the null string. There are no parameters.

FIM

A production that derives a string of length l with probability $(0.25)^l$. A **FIM** is a Free Insertion Module. There are no parameters.

4.3.3 Sequences

Sequences represent biological sequences. The format of sequences is FASTA:

```
>id_one
seq_one
>id_two
seq_two
...
```

Wildcard	Meaning
N	AGCT
V	AGC
H	ACT
D	AGT
B	GCT
S	GC
W	AT
K	GT
M	AC
R	AG
Y	CT

Figure 4.6: Sequence Wildcard Characters.

where `id_x` is the id of the sequence and `seq_x` is a string of bases. The bases are { `A,C,G,T` } with the wildcards {`NVHDBSWKMRY`}. The meaning of wildcards is given in Figure 4.6. RNACAD handles wildcards by returning the summed probability of all bases matching the wildcard. The base N, therefore, always matches a distribution with probability one.

4.3.4 Basepairs

Basepairs are given as a list. Each line is of the form, ``<seqID> <i>/<j>'', indicating the `<i>`th base is paired with the `<j>`th base for the sequence `<seqID>`.

4.3.5 Parses

A parse is a derivation of a terminal string that starts with a grammar's start state and through the application of grammar productions derives the terminal string.

A parse has the following structure:

```
<ID> <length> <log(p)> <saved_bits>
<sequence>
*NT: <name> (<i> , <j>) <production_log(p)> <production_saved_bits>
<comment>
<production>
<extra>
*NT: <name> (<i> , <j>) <production_log(p)> <production_saved_bits>
<comment>
<production>
<extra>
...
```

The identifier `<ID>` is the ID of the sequence. The identifier `<length>` is the length of the sequence. The identifier `<log(p)>` is the log probability of the sequence given the grammar. The identifier `<sequence>` is the sequence that was parsed. The identifier `<name>` is the name of the nonterminal that derives the subword, `( <i> , <j> )` with log probability, `<production_log(p)>`. The identifier `<production_saved_bits>` is savings in bits over the null model, $(0.25)^l$. The identifier `<comment>` is the comment of the production that was used. The identifier `<production>` describes the production that was used. The identifier `<extra>` is optional extra information. Thus, the first `*NT:` line will give the start symbol deriving the subword `(1 , L)`, and subsequent lines will give the entire parse.

The `<production>` outputs have the following forms based on the production type:

sequence: ``` ``sequence'' ```

chain: ``` ``chain'' ```

loop: ``` ``loop X'' ``` where X is a base.

helix: ``` ``helix /X/Y/'' ``` where X and Y are bases.

helixinsert: ``` ``helixinsert /XX/YYY/'' ``` where XX and YYY are a number of bases.

lambda: ``` ``lambda'' ```

FIM: ``` ``fim XXX'' ``` where XXX are a number of bases.

Outputs identified by `<extra>` are used to show delete structures or to explicitly state pairwise productions in linear order in the parses. This makes examination of the parse straightforward, because a linear scan of the parse shows how all terminals are ordered without having to reconstruct the derivation tree and perform an in-order traversal.

If a chain rule deletes a helix position then `<extra>` shows this delete explicitly by introducing the identifier `pseudohelixd` to indicate explicit deletion of both sides of a basepair:

```
comment%dopen
pseudohelixd -
...
comment%dclose
pseudohelixd -
```

If a chain rule deletes a loop position then `<extra>` shows this delete explicitly by introducing the identifier `loopd`:

```
commment%d
loopd -
```

For a helix production `<extra>` shows the 5' and 3' ends of the basepair explicitly by introducing the identifier `pseudohelix`. In the following example G is on the left side and C is on the right side:

```
comment%open
pseudohelix G
...
comment%close
pseudohelix C
```

For a helixinsert production `<extra>` shows the 5' and 3' ends explicitly, in this example XY on the left side and Z on the right side.

```
comment%bopen
loop X
comment%bopen
```

```
loop Y
...
comment%bclose
loop Z
```

For FIM productions <extra> explicitly shows each base, in this example XY.

```
commnent%f
loop X
loop y
```

4.3.6 Constraints

Constraints are used to specify that certain grammar nonterminals must derive certain subwords specified in some range. The format of a constraints is: ''<seqID> <NTname> <NT> <a> <b> <c> <d>'', one constraint per line. <seqID> is the sequence id for which the constraint is being given. <NTname> is the nonterminal name being constrained. <a b c d> are integers which specify the constraint with the following interpretation: if the nonterminal is used in the optimal parse of the sequence and it derives the subword (i, j) then it must be true that $(a \leq i \leq b)$ and $(c \leq j \leq d)$. If a constraint is set for a nonterminal so that <a b c d> are all equal to -2 then that nonterminal cannot be used by the parser in the derivation of the terminal string. This is used to explicitly disallow structures in the parse. If, for example, one wishes to disallow all inserts in a region of the model, then all insert nonterminals in this region can be made to be unusable.

4.3.7 Counts

This data describes summary statistics for the number of times different productions were used in a set of parses.

Statistics take different forms depending on whether the statistics are for a nonterminal, a helix, a helixinsert, or a loop.

For nonterminals the number of times each production was used is reported one line for each nonterminal of the form:

<comment1> <count1> <comment2> <count2> ...

where <comment> is the comment of the production used and <count> is the number of times that production was used.

For helix productions the number of times each basepair was used is reported:

helix_counts <NTchainedto> AA <C> AT <C> ...

where <NTchainedto> is the nonterminal that the helix production chained to, AA represents the AA basepair, and <C> represents counts.

For helixinserts the number of times each bulge length was used is reported:

helixins_counts <NTchainedto> <sum_large> <c_large> 0 1 <c> 0 2 <c> ...

where <NTchainedto> is the nonterminal that the helixinsert production chained to, <sum_large> is the sum of bulge lengths that have are longer than 3, <c_large> is the number of times that a bulge of greater than length 3 was used, 0 1 represents no left bulges and a length one bulge on the right, <c> represents counts.

For loop counts the number of times each base was used is reported:

loop_counts <NTchainedto> A <c> G <c> C <c> T <c>

where <NTchainedto> is the nonterminal that the loop production chained to and <c> represents counts.

4.3.8 Posterior Decoding

This data gives the maximum posterior decoding of a sequence given an HMM. This is a mapping from every base in a sequence to an HMM model state that has the maximum posterior probability of generating that base. The format is a file with lines of the form: `''<baseid> <basenum> <modelnum> <posterior>''` and "%" the comment character. The identifier `<baseid>` is the identity of the base. The identifier `<basenum>` is the position of the base in the sequence. The identifier `<modelnum>` is the number of the model state in the HMM. The identifier `<posterior>` is the value of the maximum posterior decoding.

4.3.9 Alignments

An alignment is a list of sequences with appropriate insert/delete characters inserted so that a structural positions that are shared in multiple sequences are lined up in columns of the alignment and the length of each sequence is equal.

An alignment has the following structure:

```
Structure_line
>id1 sequence1
>id2 sequence2
...
```

The structure_line is not needed for input and is generated by RNACAD to help identify paired columns in the alignment It uses the following symbols to show the structure:

- **( )** Helix match position.
- [ ] Helix insert position.
- `-` Loop match position.
- `.` Loop insert position.

In the sequence a base is printed in uppercase if it matches and lowercase if it is inserted. Dots are used to space out insert regions.

4.3.10 Priors

Priors define the parameters used in regularization. They are Bayesian pseudocounts based on single component Dirichlet priors. There are four types of prior. The identifier `double` is a simple scalar pseudocount that has an associated name. The identifier `loop_reg` is a 4-vector. The identifier `helix_reg` is a 16-vector. The identifier `bulge_reg` is a 16-vector with the 0 0 entry having probability zero. See Figure 4.7. The priors represent loop outputs, helix outputs, bulge outputs, and transitions between these objects.

Examining the priors leads to several observations on the *a priori* structure of RNA. The most likely loop position is A. Loops have only a few insertions and deletions. The most likely bulge is one base on the left and two bases on the right. The most likely helix position is a GC with the other Watson-Crick and wobble pairs following in likelihood. The least likely basepairs are CC and CT. Helices have only a few deletions and bulges. Basepairs are almost never deleted after a bulge is generated. It is somewhat likely that a deleted basepair is followed by a bulge. This can be justified because most basepair deletions occur by deletion of only one side of the basepair and a bulge must output the remaining half of that basepair.

```
loop_reg A 0.13609 G 0.0834278 C 0.0622226 T 0.0817855

double loop_mi_reg 0.00973904
double loop_mm_reg 0.516639
double loop_md_reg 0.0093873
double loop_ii_reg 0.0866312
double loop_im_reg 0.78417
double loop_id_reg 0.0001
double loop_di_reg 0.0309277
double loop_dm_reg 0.114987
double loop_dd_reg 0.0436671

double large_reg 0.04701
double sum_large_reg 0.4701
bulge_reg 0 0 0         0 1 0.0606848 0 2 0.0703159 0 3 0.0356998
          1 0 0.0739454 1 1 0.046944  1 2 0.149997  1 3 0.0224444
          2 0 0.0235127 2 1 0.0857707 2 2 0.0881527 2 3 0.0453631
          3 0 0.0603991 3 1 0.0340227 3 2 0.0846408 3 3 0.0112323

helix_reg AA 0.0189095 AG 0.0209065 AC 0.0121598 AT 0.158079
          GA 0.0237306 GG 0.0133492 GC 0.270011  GT 0.08335
          CA 0.0126686 CG 0.237971  CC 0.0043883 CT 0.00769489
          TA 0.181782  TG 0.0867067 TC 0.010991  TT 0.0244251

double helix_mi_reg 0.0104985
double helix_mm_reg 0.715731
double helix_md_reg 0.0146941
double helix_im_reg 45.0
double helix_id_reg 0.0001
double helix_di_reg 0.108253
double helix_dm_reg 0.0608185
double helix_dd_reg 0.247963
```

Figure 4.7: The priors used by RNACAD. These priors are based on a Dirichlet prior model and are estimated from a set of small subunit ribosomal RNA sequences. These priors define *a priori* characterizations of RNA structure. loop_reg are loop outputs. helix_reg are helix outputs. bulge_reg, large_reg, and sum_large_reg are bulge outputs based on length. loop_X_reg are loop transitions. helix_X_reg are helix transitions.

Chapter 5

Small Subunit Ribosomal RNA Modeling Using Stochastic Context-Free Grammars

The ribosome is one of the most complicated and interesting machines in the natural world. The ribosome translates the information contained in messenger RNA (mRNA) transcribed from DNA into a string of amino acids that constitute biologically active proteins. Without the ribosome, life as we know it would not exist. The complexity of this machine is enormous and elucidation of its structure and functioning has been the center of much research. In addition to this complexity is the problem of characterizing its evolutionary history. Because of its fundamental importance to life on Earth, this molecule has been studied extensively to find evolutionary relationships between different organisms and to help construct the phylogenetic tree of life. Constructing the tree of life has mainly concentrated on characterizing the RNA component of the ribosome. This work presents a new method of characterizing small subunit ribosomal RNA (SSU rRNA) by automatically producing multiple alignments of it that take into account both primary and secondary structure using stochastic context-free grammars (SCFGs). Stochastic context-free grammars are a fully probabilistic sequence model that are an extension of hidden Markov models. We compare the SCFG method to several other methods including a thermodynamic RNA folding program (MFOLD), a primary sequence aligner (ClustalW), and a hidden Markov model based primary sequence aligner (SAM). We show that the SCFG method outperforms the other methods and produces multiple alignments of quality close to hand-edited ones.

5.1 The Ribosome and Small Subunit Ribosomal RNA

The ribosome consists of several RNA and protein components [Watson et al., 1987]. The small subunit ribosomal RNA has length of approximately 2000 bases and forms a complicated secondary structure that has largely been determined using comparative sequence analysis [Gutell, 1984, Gutell, 1993, Gutell et al., 1992, Woese and Gutell, 1989, Woese et al., 1983]. The secondary structure consists of approximately 50 helical stalks. About two thirds of the bases in SSU rRNA are involved in basepair interactions. See Figure 5.1 for the secondary structure of SSU rRNA for E.coli [Maidak et al., 1997]. The high resolution three dimensional crystal structure of the ribosome is nearly complete [Clemons Jr. et al., 1999, Ban et al., 1999, Cate et al., 1999].

The ribosome is fundamentally important to life because all living things rely on it to produce protein

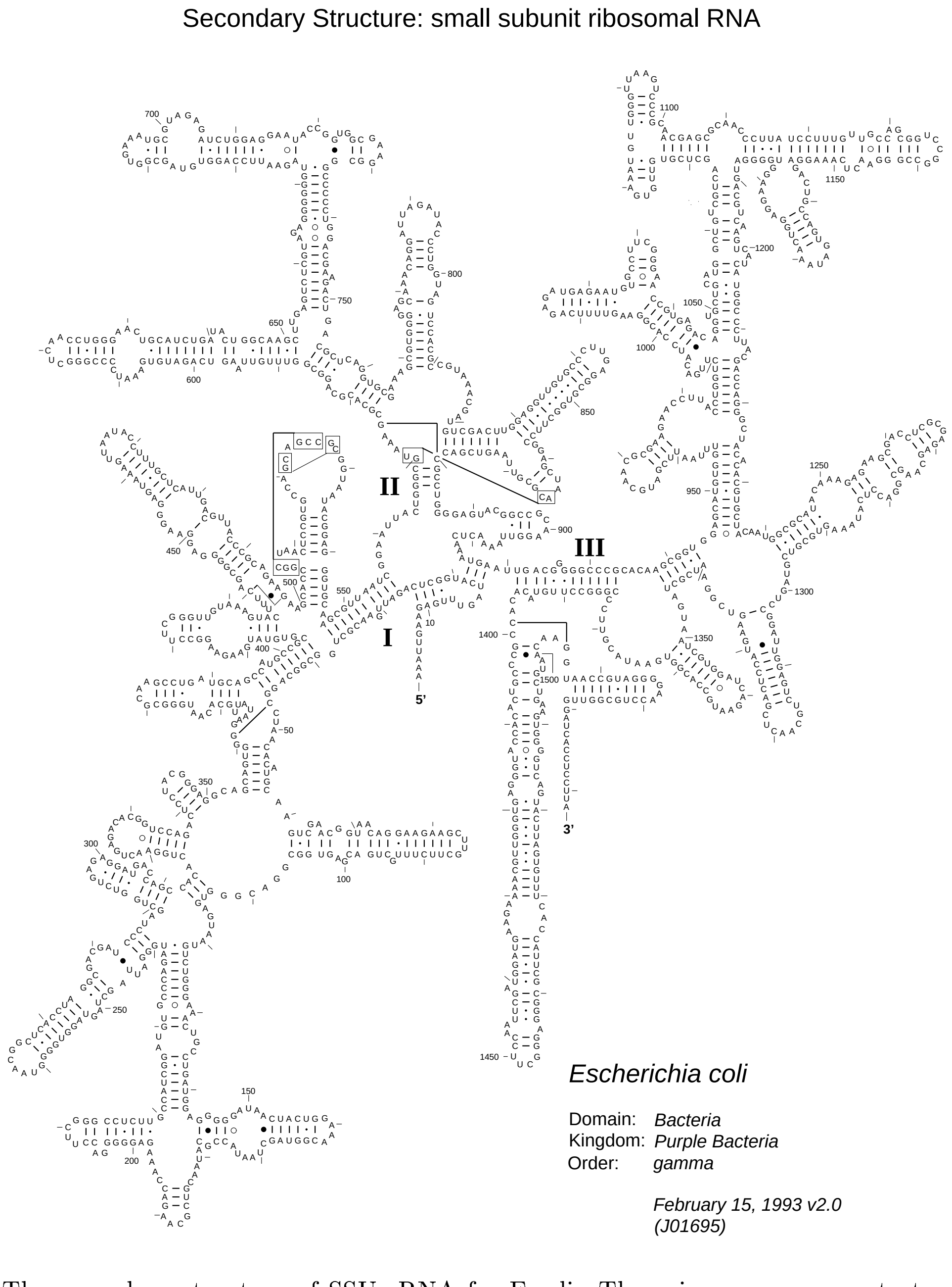


Figure 5.1: The secondary structure of SSU rRNA for E.coli. The primary sequence starts at the 5' end near the center of the image and ends at the 3' end to the right and down from the 5' end. Lines represent basepair interactions. Image from the Ribosomal Database Project.

from the mRNA transcripts of the genome. The ubiquity of the ribosome in living things has been exploited to construct phylogenetic trees containing widely different life forms from bacteria to humans by using information from the RNA components of the ribosome. There are several reasons why ribosomal RNA is informative for this purpose [Woese, 1987, Olsen and Woese, 1993]. Ribosomal RNA serves as a good molecular chronometer. Its function is highly conserved. It occurs in all organisms. Different positions in the sequence change at different rates allowing a wide dynamic range of time periods to be measured. Its size is large and consists of several domains that contain a high degree of information. It is easily and directly sequenced using reverse transcriptase reactions. There are several debates questioning the reliability of rRNA to represent phylogenetic relationships among organisms [Olsen and Woese, 1993], especially in the case of bacteria in which lateral gene transfer is common [Doolittle, 1999]. Nonetheless, analysis of rRNA has had a tremendous impact on this field.

The use of ribosomal RNA phylogenetic analyses has lead to fundamental insights in the the nature of life on Earth including the incorporation of a new domain into the tree of life, *Archaea* [Woese and Fox, 1977, Woese et al., 1990]. The ability to classify organisms based on genotypic rather than phenotypic features has opened the doors to research on bacterial phylogenies [Woese, 1987, Gray et al., 1999]. Ribosomal RNAs are routinely used to construct phylogenies of different organisms and help explain functional characteristics [Zardoya et al., 1995]. These phylogenies can be important in medicine, since several antibiotics interfere directly with ribosomal RNA [Fourmy et al., 1996].

Fundamental to all rRNA phylogenetic analysis is the construction of a multiple alignment of the sequences [Olsen and Woese, 1993, Woese, 1987, Feng and Doolittle, 1987, Morrison and Ellis, 1997]. Most phylogenetic analysis involves examining these multiple alignments to produce a tree relating the organisms. This work concentrates on constructing multiple alignments of ribosomal RNA. Readers interested in the phylogenetic analysis of these multiple alignments might refer to other works [Durbin et al., 1998, Gulko, 1995, Felsenstein, 1981].

5.2 Multiple Alignment of Ribosomal RNA Using Stochastic Context-Free Grammars

Previous attempts at aligning ribosomal RNA have relied on both automated methods and human hand fine-tuning. Indeed, recent papers have lamented that a fully automated, accurate alignment of rRNA sequences remains a difficult problem [O'Brien et al., 1998]. Some of the difficulties stem from the fact that ribosomal RNAs are large, a fact that also makes them good sources of information for phylogenetic inference. Another more fundamental difficulty arises because ribosomal RNAs form complex secondary structures. A correct alignment of rRNA must not only take into account primary sequence conservation but also information present in the secondary structure. Because most automated sequence alignment programs are unable to account for secondary structure, this poses a fundamental problem that has largely been solved by hand-tuning the alignment to make sure it agrees with the secondary structure. We propose stochastic context-free grammars (SCFGs) as a solution to both problems [Sakakibara et al., 1994a, Eddy and Durbin, 1994]. SCFGs solve the problem of aligning most secondary structure by forming a probabilistic model that takes into account both primary and secondary structure information. We also use a new probabilistically well founded technique that constrains the computation of SCFGs, allowing complex models with thousands of states to analyze biological sequences such as SSU rRNA with thousands of bases in a reasonable amount of time. Without these algorithmic techniques, simple implementations of the SCFG techniques would lead to impractical required computer resources.

Stochastic context-free grammars are an extension of hidden Markov models and have their basis in formal language theory and probability. SCFGs can be used to model secondary structure interactions of biological molecules. Like HMMs, SCFGs are probabilistic generative models. SCFGs use a grammar to generate a string by applying a series of string rewrite rules or productions. The sequence of productions used can be interpreted as representing different biological structures such as basepairs. See Figure 5.2 for an illustration of an SCFG grammar applied to RNA structure. SCFGs can be thought of as models that capture the information in a multiple alignment in which certain columns are considered to be basepaired and therefore have an informative joint distribution. SCFGs can be used to align new sequences of rRNA including prediction of all basepairs, bulges, and loop regions of a new sequence of SSU rRNA, discriminate between rRNA and non-rRNA, and create multiple alignments of rRNA [Sakakibara et al., 1994a, Eddy and Durbin, 1994].

Technically the SCFG is unable to correctly represent pseudoknots. Most of the basepairing structure of SSU rRNA is well nested and therefore representable in a SCFG. It is thought that the computational cost of representing pseudoknots outweighs the gain of modeling pseudoknot interactions in SSU rRNA for purposes of alignment and discrimination [Brown and Wilson, 1995, Rivas and Eddy, 1999, Chen et al., 1992].

Important to the application of SCFGs to large molecules such as SSU rRNA is the use of constraints. SCFG algorithm running times are known to be $O(ML^3)$ and space requirements are $O(ML^2)$ where M is the number of nonterminals and L is the length of the string. For SSU rRNA this translates to dynamic programming tables with approximately 16 billion entries. This is impractical for today's computers. A system of constraints is employed to make this problem tractable (Section 3.5).

5.3 Experimental Methods Overview

We gauge the ability of several algorithms to produce good alignments of small subunit ribosomal RNA (SSU rRNA). We examine stochastic context-free grammars(RNACAD), hidden Markov models(SAM), a primary sequence aligner(ClustalW), and an energy minimization technique(MFOLD). Each method is started with a small training set of phylogenetically diverse sequences and allowed to learn. Some methods require no training and therefore do not benefit from this learning phase. We than take an independent set of test sequences and align them using each method producing test alignments. We score each test alignment using a performance metric described below and report the results. Note that the energy minimization technique, MFOLD, does not produce an alignment as the other methods do. However, our performance metric allows comparison of MFOLD to the other methods.

5.4 Methods

5.4.1 Small Subunit Ribosomal RNA Data

There exist several sites maintaining alignments of small subunit ribosomal RNA (SSU rRNA) from thousands of species [Maidak et al., 1997, de Peer et al., 1997]. The data for this experiment was obtained from the Ribosomal Database Project (RDP) [Maidak et al., 1997]. The RDP has multiple alignments, phylogenetic trees, and secondary structures for ribosomal RNA. The multiple alignments and phylogenetic trees are available in a variety of human and machine readable forms. The secondary structures however are available only as postscript diagrams that show the secondary structure in graphical form and are not machine readable. The multiple alignments have no explicit identification of basepairing.

Our training set consists of 34 bacterial SSU rRNA sequences. See Table 5.1 for a list of the training sequences. The training set is a subset of the RDP set of Prokaryotic representative sequences. The RDP representative set consists of 80 sequences that form a representative sample across the tree of life. We removed from the representative set all sequences containing more than 1% dot characters. This is necessary because the dot character in the RDP multiple alignments indicates zero or more unknown bases and is hard to model because its definition is too loose. For the remaining sequences we changed all dots to the

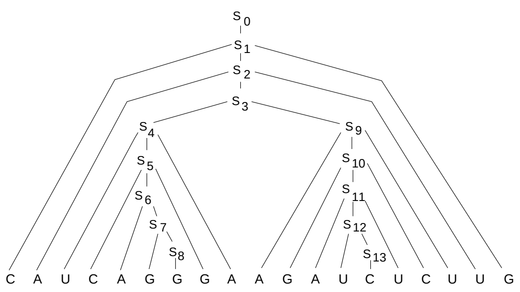
a. Productions

$$
P = \{ \quad \begin{array}{ll}
S_0 \rightarrow S_1, & S_7 \rightarrow \texttt{G}\ S_8, \\
S_1 \rightarrow \texttt{C}\ S_2\ \texttt{G}, & S_8 \rightarrow \texttt{G}, \\
S_2 \rightarrow \texttt{A}\ S_3\ \texttt{U}, & S_9 \rightarrow \texttt{A}\ S_{10}\ \texttt{U}, \\
S_3 \rightarrow S_4\ \ S_9, & S_{10} \rightarrow \texttt{G}\ S_{11}\ \texttt{C}, \\
S_4 \rightarrow \texttt{U}\ S_5\ \texttt{A}, & S_{11} \rightarrow \texttt{A}\ S_{12}\ \texttt{U}, \\
S_5 \rightarrow \texttt{C}\ S_6\ \texttt{G}, & S_{12} \rightarrow \texttt{U}\ S_{13}, \\
S_6 \rightarrow \texttt{A}\ S_7, & S_{13} \rightarrow \texttt{C}
\end{array} \quad \}
$$

b. Derivation

$$
\begin{aligned}
S_0 &\Rightarrow S_1 \Rightarrow \texttt{C}S_2\texttt{G} \Rightarrow \texttt{CA}S_3\texttt{UG} \\
&\Rightarrow \texttt{CA}S_4S_9\texttt{UG} \Rightarrow \texttt{CAU}S_5\texttt{A}S_9\texttt{UG} \\
&\Rightarrow \texttt{CAUC}S_6\texttt{GA}S_9\texttt{UG} \\
&\Rightarrow \texttt{CAUCA}S_7\texttt{GA}S_9\texttt{UG} \\
&\Rightarrow \texttt{CAUCAG}S_8\texttt{GA}S_9\texttt{UG} \\
&\Rightarrow \texttt{CAUCAGGGA}S_9\texttt{UG} \\
&\Rightarrow \texttt{CAUCAGGGAA}S_{10}\texttt{UUG} \\
&\Rightarrow \texttt{CAUCAGGGAAG}S_{11}\texttt{CUUG} \\
&\Rightarrow \texttt{CAUCAGGGAAGA}S_{12}\texttt{UCUUG} \\
&\Rightarrow \texttt{CAUCAGGGAAGAU}S_{13}\texttt{UCUUG} \\
&\Rightarrow \texttt{CAUCAGGGAAGAUCUCUUG}.
\end{aligned}
$$

c. Parse tree



d. Secondary Structure

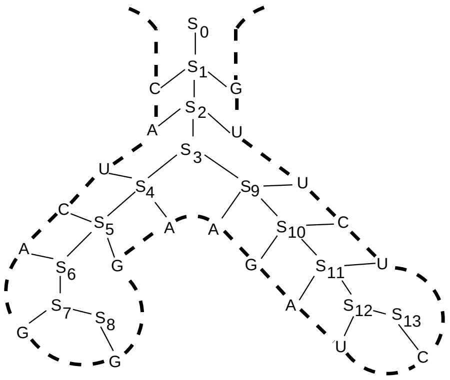


Figure 5.2: A simple context-free grammar which may be used to derive a set of RNA molecules including the specific example illustrated here, CAUCAGGGAAGAUCUCUUG. *a.* A set of productions P which generates RNA sequences with a certain restricted structure. S_0 (start symbol), $S_1, \ldots, S_{13}$ are nonterminals; A, U, G and C are terminals representing the four nucleotides. *b.* Application of the productions P could generate the given sequence by the derivation indicated. For example, if the production $S_1 \rightarrow CS_2G$ is selected, the string CS_2G replaces S_1 and the derivation step is written $S_1 \Rightarrow CS_2G$. *c.* The derivation in *b* may be arranged in a tree structure called a *parse* or *derivation tree*. *d.* The physical secondary structure of the RNA sequence is a reflection of the parse tree (or syntactic structure).

Tt.maritim	Tmc.roseum	D.radiodur	Bac.fragil	Sap.grandi	Chl.limico	Pir.staley	Clm.psitta
Nost.muscr	Syn.6301	Olst.lut_C	Zea_mays_C	Spi.haloph	Lpn.illini	Ric.prowaz	Nis.gonor1
Ps.aerugi3	E.coli	Dsv.desulf	Myx.xanthu	Hlb.pylor6	Cam.jejun5	Fus.nuclea	Stm.ambofa
Arb.globif	Bif.bifidu	Eub.barker	B.subtilis	L.casei	Eco.faecal	Acp.laidla	C.ramosum	M.capricol
C.pasteuri							

Table 5.1: The 34 bacterial sequences in the training set. The listing left to right and top to bottom is phylogenetically ordered. These sequences are identified in the RDP Bacterial representative set as spanning the phylogenetic tree.

indel character, '-', if the alignment column contains only indels and dots and to the wildcard character, 'N', otherwise. This left 34 sequences with no dots as our training set.

In addition to the training sequences, we include the secondary structure for the training sequence *E. coli* in our training data. The secondary structure is given as a list of pairs of bases that are thought to be involved in basepairing interactions. This list was obtained by examining the RDP secondary structure postscript diagram for *E.coli*. We remove any pseudoknotted interactions from the secondary structure specification because they are not representable by the SCFG. We infer basepairing interactions for sequences other than *E.coli* using the multiple alignment. If *E.coli* has a pair of basepairing bases located in two multiple alignment positions in the RDP alignment then all other sequences that have bases in those two columns are inferred to be basepairing as well. The stochastic context-free grammar algorithms are able to exploit the information contained in the pairwise interactions specified by the secondary structure. Therefore if there is additional information in the pairwise interactions that is not present in the primary sequence, the stochastic context-free grammar algorithms will be able to use this information and provide superior performance.

Our test set contains a total of 169 SSU rRNA sequences: 119 Bacterial and 50 Archaea sequences. This set is derived from the RDP Prokaryotic Multiple Alignment datafile and consists of all sequences in the database that do not contain the dot character. As mentioned previously, the dot character has a loose definition indicating zero or more unknown bases and is too noisy to be modeled accurately.

5.4.2 Tested Modeling Algorithms

Stochastic Context-Free Grammar, RNACAD

The RNACAD system was used to analyze the SSU rRNA data (Chapter 4). The system was presented with the training set and the list of basepairs. The system automatically generated an estimated grammar from this data that had approximately 9,000 states and 50,000 parameters. This includes 554 modeled loop positions and 478 modeled basepair positions. In order to use this grammar on the test set, an HMM approximation to the SCFG was needed. The SAM HMM modeling system was used[Hughey and Krogh, 1996]. The SAM system was given the multiple alignment and generated an estimated HMM using the `modelfromalign` program. The SCFG was then used to parse the test sequences with constraints supplied by the HMM. Problems of this size would not run without the HMM supplied constraints. The parses of the test set sequences were scanned to identify bases that were produced by helix productions and therefore were predicted to be basepairing.

Hidden Markov Model, SAM

The SAM system models primary sequence structure using hidden Markov models(HMMs)[Hughey and Krogh, 1996]. An HMM probabilistically encodes the information of a multiple alignment in its state structure but does not model any interactions between multiple alignment columns. HMMs model loop positions exactly as

SCFGs do. However, HMMs are unable to model pairwise interactions and therefore basepaired positions are modeled as two independent loop positions.

An HMM was trained on the RDP multiple alignment using `modelfromalign`. The number of of the HMM superstates was 1515. This HMM was then used to align the test set using `align2model`. Only default parameters were used. A list of basepairs was derived from this multiple alignment using the secondary structure of *E.coli* and inferring basepairing interactions for other sequences. SAM v2.2.1 was used.

Profile Alignment, ClustalW

The ClustalW method models primary structure and uses profile-based progressive multiple alignments to construct a full multiple alignment from a set of sequences [Thompson et al., 1994]. This is done by first constructing a distance matrix on all pairs of sequences using a pairwise dynamic programming alignment, clustering the distances to construct a guide tree, and using this tree to progressively align sequences starting with the most similar sequences and continuing with sequence-sequence, sequence-profile, and profile-profile alignments until all sequences are aligned. This is one of the most widely used sequence alignment programs.

ClustalW(v 1.7) was used to align the test set using the Slow/Accurate multiple alignment option. A list of basepairs was derived from this multiple alignment using the secondary structure of *E.coli* and inferring basepairing interactions for other sequences.

Energy Minimization, MFOLD

MFOLD relies on thermodynamic free energy parameters to optimally fold an RNA sequence taking into consideration basepairing interactions and stacking[Zuker et al., 1999, Mathews et al., 1999]. The parameters are thermodynamically derived and are not specific to the particular RNA that is being folded. This is one of the most widely used programs for predicting RNA secondary structure.

The test sequences were submitted to the MFOLD server and a list of basepairs was derived from resulting foldings. The web-based version of MFOLD 2.3 was used.

5.4.3 The Performance Metric

In order to gauge the ability of the algorithms to produce good alignments of small subunit ribosomal RNA (SSU rRNA) we use a stringent performance metric based on basepairing interactions. Performance on the test set for each method is reported by three numbers: the number of correctly identified basepairs (true positives), the number of missed basepairs (false negatives), and the number of false reported basepairs (false positives). All numbers are reported with respect to the basepairing interactions as reported or implied by our data source, the Ribosomal Database II Project (RDP).

This metric is an indication of good alignment because more than half of the bases in SSU rRNA are involved in basepairs. If an alignment program were to shift any of the bases involved in either the 3' or 5' side of a basepair, an error would result. Therefore a method can get no errors only if it identifies every base in every basepair exactly. The metric could be made more stringent by forcing all loop positions to align also but because some loop regions are thought to be "inserts", no alignment in these regions is expected. Thus penalizing for an incorrect alignment in these regions was thought to be too harsh.

In order to compute the performance metric, a list of correct basepairs must be generated from the RDP alignment. This is done by taking the secondary structure basepair list for *E.coli*. For each *E.coli* basepair, find the corresponding columns in the multiple alignment that contain that E.coli basepair. If the sequence, S, has bases i and j in those multiple alignment columns then a basepair (i, j) is reported for sequence S. Note that we assume that the RDP alignment is correct. If the RDP alignment contains any errors then an erroneous basepair might be reported and any method that corrects the error will be penalized by the performance metric because it disagrees with the RDP.

The stochastic context-free grammar system, RNACAD, and the energy minimization algorithm, MFOLD, both make basepair predictions directly. The hidden Markov model system, SAM, and the primary sequence aligner, ClustalW, produce multiple alignments that are processed as described above to produce basepair lists.

5.5 Results and Discussion

The test set contains 169 sequences and 76,167 basepairs as identified by the RDP multiple alignment and the secondary structure of *E.coli*.

5.5.1 Primary Alignment, ClustalW

False Positives	False Negatives	True Positives
5623	7283 (9.6%)	68884 (90.4%)

ClustalW is a primary sequence aligner and is able to exploit the well-conserved regions of SSU rRNA to produce a multiple alignment. However ClustalW does slightly worse than the HMM method which also is a primary sequence aligner. It seems as though the statistical information captured by the HMM is important in the alignment of SSU rRNA. ClustalW's method of using pairwise alignment, using these alignments to construct a guide tree, and then using this tree to construct a full alignment might have done well, especially because sequences are weighted to account for biased representation, but its performance is worse than an HMM.

5.5.2 Hidden Markov Models, SAM

False Positives	False Negatives	True Positives
5161	5632 (7.4%)	70535 (92.6%)

The HMM is unable to represent basepairing interactions but still does well. The HMM is able to model all loop positions and well conserved basepairing positions are modeled as two independent positions very well. This primary probabilistic modeling captures much of Bacterial SSU rRNA information as evidenced by its good performance.

5.5.3 Energy Minimization, MFOLD

False Positives	False Negatives	True Positives
7700	11284 (65.7%)	5883 (34.3%)

MFOLD does not do very well. In nature, SSU rRNA folds interacting with many other protein and RNA components. Perhaps simple thermodynamic parameters cannot capture all the necessary information for correct folding of complexes of RNA and other biological molecules.

Previous results show that the accuracy of MFOLD to be somewhat better than the results reported here with about a 50% true positive rate [Konings and Gutell, 1995]. This might be explained by the smaller test set of about 41 non-Eukaryote sequences used in the other work. This smaller set might contain more easily predicted sequences than the 169 sequences examined here.

5.5.4 Stochastic Context-Free Grammars, RNACAD

False Positives	False Negatives	True Positives
1926	2073 (2.7%)	74094 (97.3%)

Number of errors more than	but less than	Number of Nonterminals
-1	1	186
0	11	194
10	21	37
20	31	17
30	41	14
40	51	12
50	61	3
60	71	6
70	81	4
80	91	2
90	101	3
-1	101	478

Table 5.2: RNACAD basepair errors grouped by nonterminal. The table gives for a range of number of errors, the number of nonterminals having a number of errors in that range. Number of errors is the number of false positives plus the number of false negatives for that nonterminal. Most nonterminals have no or a small number of errors. The total number of helix producing nonterminals is 478.

The SCFG does very well, the best in these tests. The next section examines some of the errors in more detail.

Analysis of RNACAD Errors

When we examine the false negative and false positive errors made by RNACAD with respect to the RDP alignment we find that the errors group together. See Figure 5.3 for a graphical representation of these errors. Some errors occur in structure that is different for Archaea [Woese, 1987]. Because our training set was entirely Bacteria and our test set contained Archaea, errors have occured because the RDP alignment for Archaea was different from Bacteria. Because the RNACAD system is trying to model all sequences under *one* universal secondary structure, it tries to place the Archaea structure under the Bacteria structure in several regions and consequently these predictions are counted as errors when compared to the "correct" RDP alignment. Table 5.2 shows how errors are distributed among different nonterminals.

Examining the errors as grouped by sequence we find that most sequences have few errors. See Table 5.4 for a grouping of errors by sequence. See Table 5.3 for a graphical representation of how the errors are distributed phylogenetically given the phylogenetic grouping of sequences maintained by the RDP.

Interestingly the sequence Prth.zop_C with the worst prediction with 101 errors is close in the phylogenetic listing to Ochs.nea_C with only 1 error. Prth.zop_C is in SKELETONEMA SUBGROUP while Ochs.nea_C is in OCHROSPHAERA SUBGROUP both under BACTERIA, CYANOBACTERIA AND CHLOROPLASTS, CHLOROPLASTS AND CYANELLES, OTHER CHLOROPLASTS. The length of Prth.zop_C is 1515 and the length of Ochs.nea_C is 1483. While the two sequences have similar lengths, the sequence Ochs.nea_C is much more similar to E.coli than Prth.zop_C is. The sequence Prth.zop_C might have more basepair prediction errors because it is more distantly related to the set of sequences used to estimate the SCFG which includes E.coli. Even though the RDP places both Prth.zop_C and Ochs.nea_C under the subgroup OTHER CHLOROPLASTS, the sequences have mutated to be very different from each other. The subgroup OTHER CHLOROPLASTS, it seems, contains distantly related organisms.

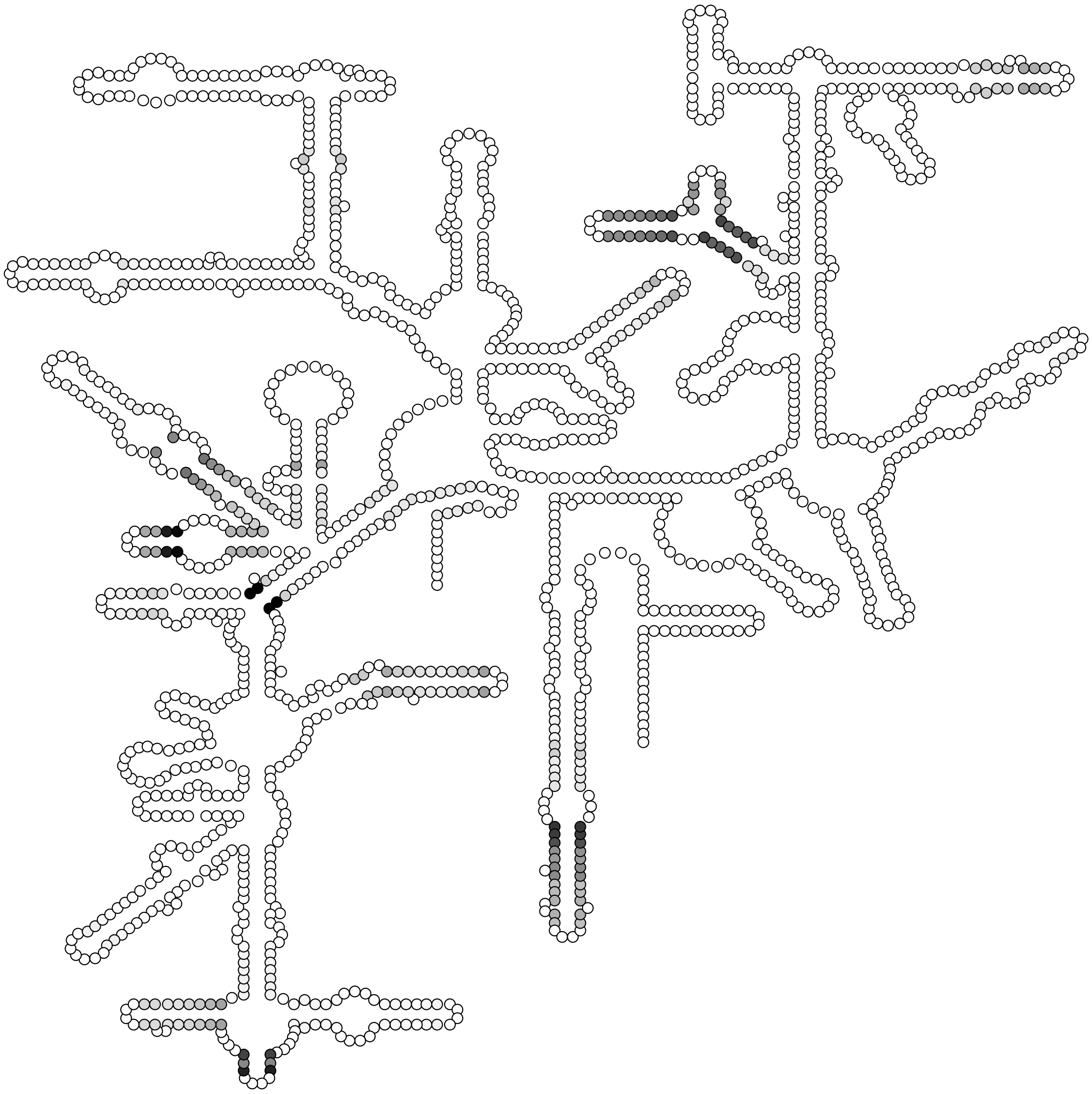


Figure 5.3: Number of basepair errors on the SSU rRNA test set made by the RNACAD system for all 169 test sequences. The graphic represents the E.coli secondary structure for SSU rRNA, one circle for each base. The darkness of each circle represents the number of errors that were made for that basepair. The number of errors is the number of false positives plus the number of false negatives. Black represents the maximum number of errors (100) and white represents no errors. It is clear the errors do not occur randomly but are grouped together.

= ARCHAEA 51.88
== EURYARCHAEOTA 48.02
=== METHANOCOCCALES 37.00
=== METHANOBACTERIALES 54.40
=== METHANOMICROBACTERIA_AND_RELATIVES 46.26
=== THERMOCOCCALES 68.00
=== METHANOPYRALES 42.00
== CRENARCHAEOTA 69.00
=== CRENARCHAEOTA_GROUP_I 70.88
=== CRENARCHAEOTA_GROUP_II 54.00
= BACTERIA 11.97
== THERMOTOGALES 16.00
=== GEOTOGA_GROUP 28.00
=== THERMOTOGA_GROUP 4.00
== GREEN_NON-SULFUR_BACTERIA_AND_RELATIVES 7.50
=== CHLOROFLEXUS_SUBDIVISION 15.00
=== DEINOCOCCUS-THERMUS_SUBDIVISION 0.00
== FLEXIBACTER-CYTOPHAGA-BACTEROIDES_PHYLUM 5.00
=== SUBDIVISION_I 5.00
== GREEN_SULFUR_BACTERIA 14.00
== PLANCTOMYCES_AND_RELATIVES 3.67
=== PLANCTOMYCES_SUBDIVISION 4.00
=== CHLAMYDIA_SUBDIVISION 3.50
== CYANOBACTERIA_AND_CHLOROPLASTS 20.03
=== CHLOROPLASTS_AND_CYANELLES 20.03
== SPIROCHETES_AND_RELATIVES 11.00
=== LEPTOSPIRA_SUBDIVISION 11.00
== PURPLE_BACTERIA 9.85
=== ALPHA_SUBDIVISION 23.67
=== BETA_SUBDIVISION 3.67
=== GAMMA_SUBDIVISION 3.44
=== DELTA_SUBDIVISION 13.50
=== EPSILON_SUBDIVISION 6.33
== FUSOBACTERIA_AND_RELATIVES 14.00
== GRAM_POSITIVE_PHYLUM 5.45
=== HIGH_G+C_SUBDIVISION 5.67
=== C.LITUSEBURENSE_GROUP 19.00
=== BACILLUS-LACTOBACILLUS-STREPTOCOCCUS_SUBDIVISION 1.87
=== MYCOPLASMA_AND_RELATIVES 9.50
=== CLOSTRIDIUM_AND_RELATIVES 5.25

Table 5.3: Number of RNACAD errors grouped on the RDP linearly arranged phylogenetic tree. The characters "=" represent level of the tree. The number after the group name is the average number of basepair errors for all sequences contained in that phylogenetic group. The length of the bars on the right represent the average number of errors for that group. The smallest non-zero average error is 1.87 and the largest is 70.88.

Number of errors more than	but less than	Number of Sequences
-1	11	82
10	21	13
20	31	19
30	41	8
40	51	21
50	61	10
60	71	10
70	81	2
80	91	4
90	101	0
100	102	1
-1	102	169

Table 5.4: RNACAD basepair errors grouped by sequence. The table gives for a range of number of errors, the number of sequences having a number of errors in that range. Number of errors is the number of false positives plus the number of false negatives for that nonterminal. The total number of sequences is 169.

5.5.5 RNACAD Model Analysis Using Support Vector Machines

The modeling of SSU rRNA using SCFGs allows probabilistic techniques to be applied to the model in the analysis of SSU rRNA. Information metrics such as mutual information on the basepair states are directly computed from the SCFG. Mutual information gives the the amount of additional information contained in a pair of positions over and above the information contained in the two positions independently [Cover, 1991]. See Figure 5.4 for a graphic representation of mutual information on basepairs.

When you model sequences with a generative probability model there is a way to embed sequences in a regular Euclidean space for purposes of classification. This ideas was first proposed by Jaakkola, Diekhans, and Haussler and was applied to the problem of protein super family classification using HMMs [Jaakkola et al., 1999]. Here we apply it for the first time to SCFGs.

Traditionally sequence classification is done by constructing a probabilistic model of sequences and then performing a Bayes decision using the log posterior odds score. This decision rule uses only the probability of the sequence under the probabilistic model. Because probabilistic generative models such as HMMs and SCFGs contain a discrete state structure that generates probabilities, a much richer feature set than simply the final probability is available for classification. The feature set could include the probability that each state was used in the derivation of the sequence being classified, for example. Thus, features pulled from the probabilistic generative model of a sequence can be used in addition to the final probability for classification purposes.

The Fisher information feature vector, $[\nabla_\theta(\log P(X|\theta))]$ where X is the sequence and θ represents a probabilistic generative model such as an HMM or SCFG, has been shown to be useful for classification [Jaakkola et al., 1999]. Another feature vector is used in this work, the bits-saved vector. This vector groups together parameters of a SCFG according to the superstate structure of the model (Section 3.3.3). If the SCFG has n superstates then the bits-saved vector has n components. The ith component of the bits-saved vector returns, for the Viterbi parse of the sequence, how many bits were saved by the SCFG over the null model for superstate i deriving its portion of the sequence. Let $S1$ represent a nonterminal corresponding to superstate one and $S2$ represent the next superstate. In the most likely parse of a sequence, x, let $S1$ derive the subword $x_i ... x_j$ with probability $P(S1 \Rightarrow^* x_i \dots x_j)$ and $S2$ derive the subword $x_{i'} \dots x_{j'}$ with probability $P(S2 \Rightarrow^* x_{i'} \dots x_{j'})$ where $i \le i' \le j' \le j$. The bits-saved vector component corresponding to

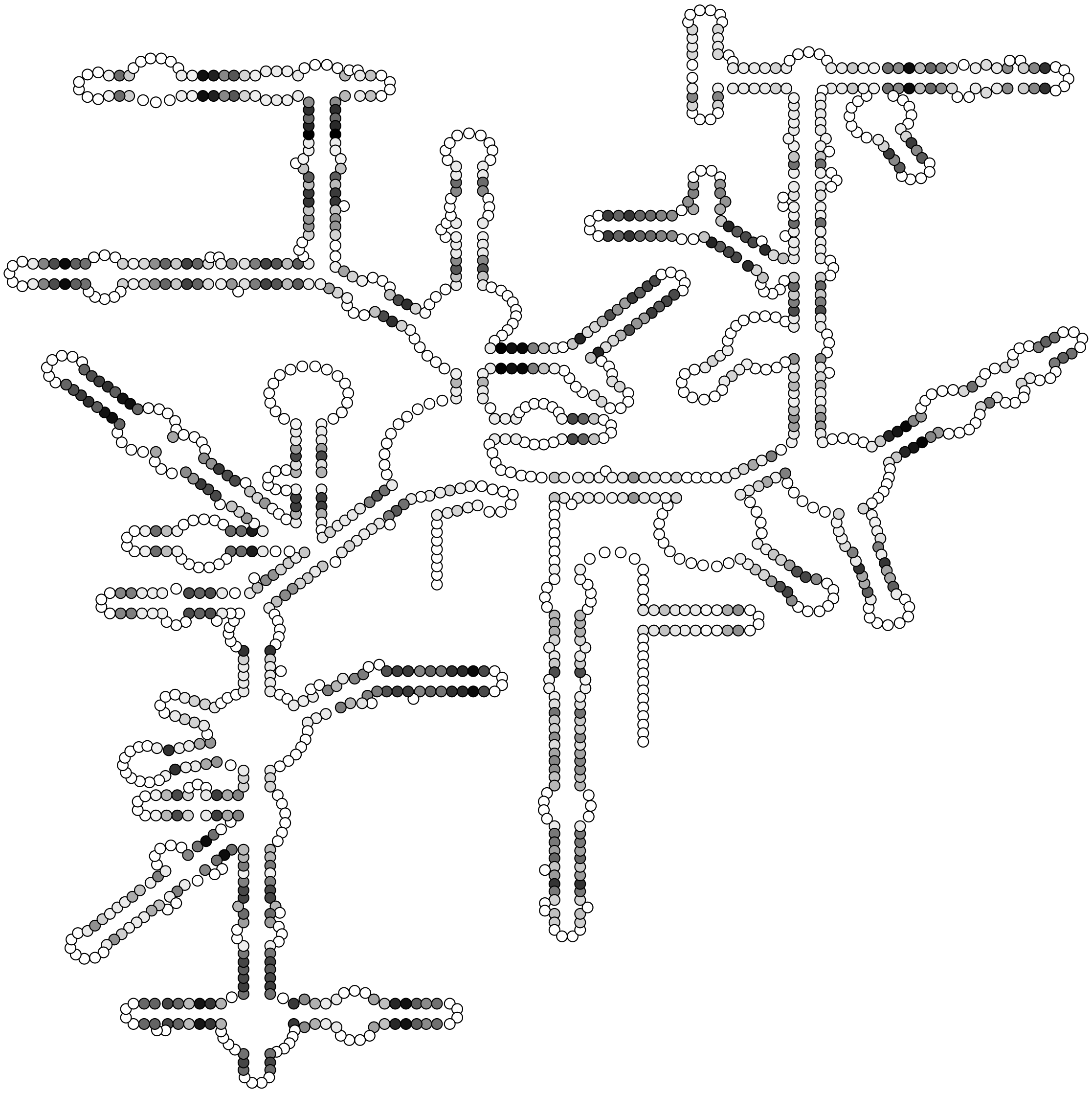


Figure 5.4: The mutual information metric computed on basepair distributions in the RNACAD model. The graphic represents the E.coli secondary structure for SSU rRNA, one circle for each base. The intensity of each circle represents the mutual information metric on the distribution responsible for that basepair. Black represents the maximal mutual information (1.77 bits) and white represents the minimal mutual information (0.27 bits). It is clear that some helices do not have high mutual information. This might be caused by basepair positions with very high conservation.

Subgroup	Number of sequences
HIGH_G+C_SUBDIVISION	3
C.LITUSEBURENSE_GROUP	1
BACILLUS-LACTOBACILLUS-STREPTOCOCCUS	15
MYCOPLASMA_AND_RELATIVES	10
CLOSTRIDIUM_AND_RELATIVES	4

Table 5.5: Phylogenetic subgroups of the Gram Positive Group as defined by the RDP. The subgroup name and the number of sequences in that subgroup are given.

superstate $S1$ has value

$$\log P(S1 \Rightarrow^* x_i \ldots x_j) - \log P(S2 \Rightarrow^* x_{i'} \ldots x_{j'}) + log(1/4)^{(j'-j)+(i-i')} \tag{5.1}$$

Where the null model for a string of length L has probability $(1/4)^L$. This quantity is the bits saved over the null model for $S1$ deriving its portion of the sequence in the most likely parse. The bits saved for superstate $S1$ is $\log P(S1 \Rightarrow^* x_i \ldots x_j) - log(1/4)^{j-i+1}$ and the bits saved for superstate $S2$ is $\log P(S2 \Rightarrow^* x_{i'} \ldots x_{j'}) - log(1/4)^{j'-i'+1}$. Therefore, because $S2$ derives a subword contained within the subword derived by $S1$, the savings in bits for $S1$ deriving only its portion of the sequence is the savings in bits for $S1$ minus the savings in bits for $S2$. Other bits-saved vector components are defined similarly for the different SCFG superstates. Every parse uses every superstate exactly once by construction of the SCFG and therefore the bits-saved vector is well-defined for any parse of any sequence using the SCFG.

To test the utility of the bits-saved feature vector in classification, a phylogenetic discrimination task was examined. As mentioned earlier, phylogenetic tree construction using SSU rRNA information has been the center of much work. The discrimination task based on phylogenetic analysis starts with an existing phylogenetic tree, takes one branch of that tree, and labels all sequences in that branch positively (+1) and all other sequences not in that branch negatively (-1). A phylogenetic classifier will take a new sequence and correctly label it positively or negatively according to whether it belongs in that branch in the phylogenetic tree. The accuracy of the classifier is measured in a cross-validated experiment in which the classifier is given a subset of the sequences and their labels for training and then the remaining sequences are used as a independent test set. Accuracy is measured by performance on the test set.

I examined the phylogenetic tree available from the Ribosomal Database Project [Maidak et al., 1997]. I chose the phylogenetic branch GRAM_POSITIVE_PHYLUM. All sequences in this branch are labeled positively (+1) and all other sequences are labeled negatively (-1). In order to perform a cross-validated experiment, the sequences must be divided into train and test sets. Sequences were divided according to phylogenetic subgroup. The GRAM_POSITIVE_PHYLUM contains five subgroups. See Table 5.5 for a list of these subgroups. The negative sequences contain both Archaea and Bacteria. There are eleven subgroups for the negative sequences. Negative subgroups were combined to construct two negative groups. See Table 5.6 for a list of these groups. Dividing sequences in this way gives five positive groups and two negative groups.

The experimental cross-validation consists of choosing one positive and one negative group. The sequences in these two chosen groups form the test set. The training set consists of the all the sequences not in the test set and their corresponding labels. The classifier is allowed to train on the training set and then makes predictions for each sequence in the test set. If the prediction is positive and the true label is positive then that sequence is counted as a true positive(tp). If both the prediction and the truth are negative then it is a true negative(tn). If the prediction is positive and the truth is negative then it is a false positive(fp). If the prediction is negative and the truth is positive then it is a false negative(fn). Accuracy is reported by these four numbers for each fold of testing on one positive and one negative group. Therefore, there are ten

Name	Subgroup	Number of sequences
zero	EURYARCHAEOTA	40
	CRENARCHAEOTA	9
	THERMOTOGALES	2
	GREEN_NON-SULFUR_BACTERIA	2
	FLEXIBACTER-CYTOPHAGA-BACTEROIDES	1
	GREEN_SULFUR_BACTERIA	1
	PLANCTOMYCES_AND_RELATIVES	3
one	CYANOBACTERIA_AND_CHLOROPLASTS	39
	SPIROCHETES_AND_RELATIVES	1
	PURPLE_BACTERIA	33
	FUSOBACTERIA_AND_RELATIVES	1

Table 5.6: Phylogenetic subgroups of the non-Gram Positive Group as defined by the RDP. The subgroup name and the number of sequence in that subgroup are given. These subgroups are combined to give the two test groups. Test group zero contains subgroups EURYARCHAEOTA through PLANCTOMYCES_AND_RELATIVES with 58 sequences and test group one contains subgroups CYANOBACTERIA_AND_CHLOROPLASTS through FUSOBACTERIA_AND_RELATIVES with 74 sequences.

(5 positive groups * 2 negative groups) experiments.

The standard method of performing this classification is to use a nearest-neighbor technique. A database of sequences and their phylogenetic labels is used. Given a new sequence, a BLAST search is done against the database. The best hit in the database is identified and the label of the best hit is predicted as the label for the new sequence. This is the method used by the RDP to place sequences in the phylogenetic tree using a slightly modified BLAST search [Maidak et al., 1997]. This method was implemented for testing purposes by constructing a BLAST database for the training set, performing a gapped BLAST search against this database for each test sequence using WU-BLAST (2.0a16), and predicting the label as the label of the best BLAST hit.

The method using the bits-saved vectors is based on a support vector machine (SVM) classification framework [Vapnik, 1998, Brown et al., 1999]. Bits-saved feature vectors are constructed for each sequence in the database by constructing a Viterbi parse for each sequence under the SSU rRNA SCFG (Section 5.4.2) and constructing bits-saved vectors from each parse. The SVM does all classification based on these bits-saved vectors and does not operate on the sequences themselves in the input space. The bits-saved vectors and their corresponding labels for the training set are presented to the SVM for training. The SVM estimates parameters from the training set using a radial basis kernel function similar to one used in previous work on microarray gene expression data analysis [Brown et al., 1999]. Testing is done by presenting a bits-saved vector for each sequence in the test set to the SVM, the SVM predicts the label to be positive or negative, and results are tallied.

The results for both BLAST and the SVM are given in Table 5.7. The SVM does significantly better than the BLAST method making on average 22.3 fewer errors over the ten tests. The SVM makes far fewer false positive errors. This means that when the BLAST method makes predictions for sequences that should have a negative label it finds that the best BLAST hit actually has a positive label. This happens especially often in negative group one. Negative group one consists of mostly Chloroplasts and Purple Bacteria. The best BLAST hits are to sequences in the positive group, Gram Positive Bacteria, and not to the sequences in negative group zero which are mostly Archaea and nine Bacteria. BLAST places the Purple Bacteria and Chloroplasts closer to Gram Positive Bacteria than to the bacteria in negative group zero and consequently makes false positive errors. The SVM also makes this mistake, as evidenced by its worse performance on

Positive Test	Negative Test	Method	fp	fn	tp	tn	BLAST-SVM Errors
HighGC	zero	blast	11	1	2	47	
HighGC	zero	svm	1	1	2	57	10
HighGC	one	blast	73	0	3	1	
HighGC	one	svm	51	0	3	23	22
C.Lit	zero	blast	32	0	1	26	
C.Lit	zero	svm	1	0	1	57	31
C.Lit	one	blast	73	0	1	1	
C.Lit	one	svm	54	0	1	20	19
Bac	zero	blast	28	0	15	30	
Bac	zero	svm	2	1	14	56	25
Bac	one	blast	71	0	15	3	
Bac	one	svm	58	0	15	16	13
Myco	zero	blast	31	1	9	27	
Myco	zero	svm	1	7	3	57	24
Myco	one	blast	73	0	10	1	
Myco	one	svm	38	6	4	36	29
Clos	zero	blast	32	0	4	26	
Clos	zero	svm	1	2	2	57	29
Clos	one	blast	73	0	4	1	
Clos	one	svm	52	0	4	22	21

Table 5.7: SSU rRNA phylogenetic classification results for the Gram Positive classification problem. False Positive (fp), False Negative (fn), True Positive (tp), and True Negative (fn) results for the BLAST and SVM method are reported. Positive Test is the positive group used for testing identified by a prefix of the full phylogenetic subgroup name as defined in Table 5.5. Negative Test is the negative group used for testing identified by group zero and one as defined in Table 5.6. The SVM does significantly better than the BLAST method when measuring the number of false positives plus the number of false negatives.

negative group one, however it makes significantly less errors than BLAST. The SVM is able to use the information in the bits-saved vector that gives it an edge over simple sequence similarity as measured by BLAST. The extra information in the bits-saved vector might be given by the basepairing structure that is captured by the SCFG but cannot be accounted for by the BLAST method.

Not only does the SVM with SCFG bits-saved vectors have better performance than BLAST, it also makes the use of complex generative models such as SCFGs feasible for large classification problems. The computational resources required by SCFGs are great especially for large molecules such as SSU rRNA. If the above experiments were performed using SCFGs alone, the procedure would require the estimation of a SCFG for each training set and a parse of each test set sequence given the estimated SCFG to determine whether the test sequence is a member. This is very costly. However, in the case of the SVM bits-saved classification framework, the bits-saved vectors are computed just once using SCFG parsing and do not change for different train and test splits of the data. The SCFG generative model is estimated once for SSU rRNA sequences and bits-saved feature vectors are constructed once from sequence parses. All cross-validated training in the SVM framework simply changes which bits-saved vectors that are used in training. Because the SVM training and testing is much less costly than SCFG training and testing, the SVM is able to perform this classification task more efficiently than using SCFGs alone.

5.6 Conclusions and Future Work

Probabilistic models of small subunit ribosomal RNA using stochastic context-free grammars produce superior multiple alignments in relation to several other methods in terms of basepair prediction and alignment. The quality of the SCFG alignment is close to the quality of the hand-edited alignment. The method runs in a reasonable amount time. This is significant because simple implementations of the method would lead to impossible required computer resources.

A web server that uses this method at http://www.cse.ucsc.edu/research/compbio/ssurrna.html. This web server allows you to enter a sequence and predict its secondary structure as a bacterial small subunit ribosomal RNA. You can then generate multiple alignments, report a list of basepairs, and interact with the secondary structure using a Java graphical user interface. See Figure 5.5 for a snapshot of this tool.

Because of the method's ability to produce high quality multiple alignments of ribosomal RNA, it has direct application in areas such as phylogenetic tree reconstruction [Olsen and Woese, 1993]. This could be important in areas such as organism diversity in natural environments [Pace, 1997] or identification of disease or infection [Riley et al., 1998]. The method of phylogenetic classification using SCFG bits-saved feature vectors could be optimized. Other feature vectors might be constructed and might lead to better performance.

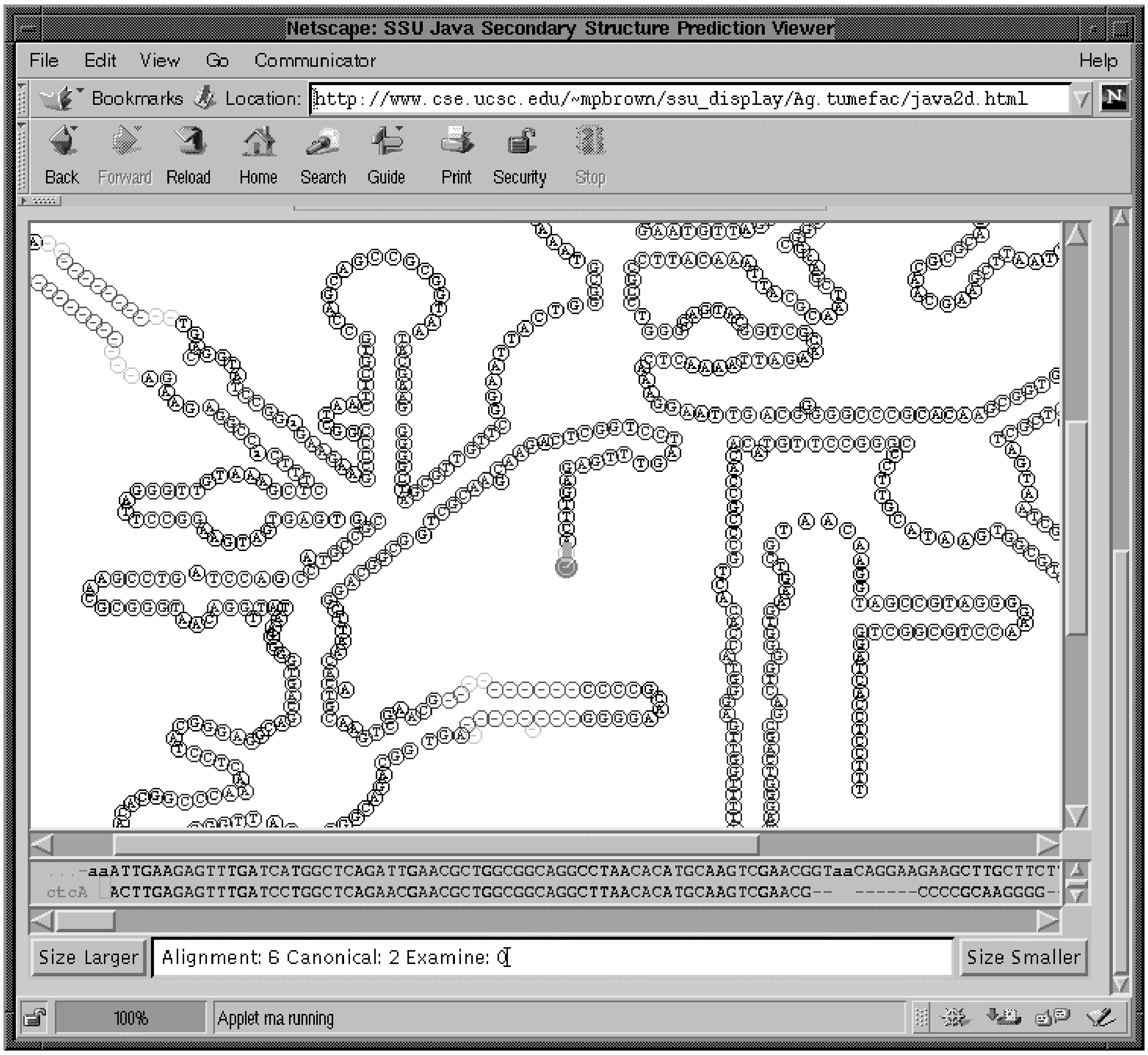


Figure 5.5: The web-based SSU rRNA RNACAD system located at http://www.cse.ucsc.edu/research/compbio/ssurrna.html. Shown is the java secondary structure viewer. The sequence Ag.tumefac is being examined. Ag.tumefac's secondary structure is shown in relation to E.coli's secondary structure. The alignment of Ag.tumefac and E.coli is also shown.

Chapter 6

Biotin Binding RNA: Pseudoknot Modeling Using Stochastic Context-Free Grammars

In light of the enormous generation of sequence data by projects such as the human genome project, the task of searching sequence databases has become increasingly important. The fields of computer science and biology have collaborated to design highly sensitive filters which can be used to sieve through this massive collection of data and return only those precious sequences for which the filters are designed.

The pseudoknot is an important structure of RNA that occurs in virtually all classes of RNA and has shown to be important in many biological functions [ten Dam et al., 1992]. Unfortunately, programs used to search large sequence databases for these pseudoknot structures in their general form seem to require running times that are prohibitive computationally. Examples are David Searls modeling of DNA repeats in any form [Searls, 1992], Mamitsuka and Abe modeling beta sheets using Stochastic Ranked Node Rewriting Grammars [Mamitsuka and Abe, 1994], and Eddy and Rivas work on thermodynamic folding including pseudoknots [Rivas and Eddy, 1999].

This paper presents a model based on intersections of stochastic context-free grammars which allows highly sensitive and relatively fast database searching for pseudoknotted RNA structures. It is shown that this model can discriminate RNA sequences containing a pseudoknotted structure for binding biotin from random RNA sequences which do not contain this structure and therefore can perform database searching.

The RNA pseudoknot is the representative problem domain for which our models provide a solution. A RNA pseudoknot is formed when bases outside a hairpin structure pair with bases within the hairpin loop to create a second stem and loop structure [Chen et al., 1992]. See Figure 6.1. Pseudoknots have been found in small subunit ribosomal RNA and seem to be very important in its functioning. Pseudoknots have been observed in many structures including telemerases, 7SL RNA, U2 snRNA, group I introns, and viral RNA. Pseudoknots seem to have importance in messenger RNAs with frame shifts in gene expression and suppressors [ten Dam et al., 1992]. Pseudoknotted RNA also frequently occurs in *in vitro* RNA selection experiments. Recent examples include a RNA pseudoknot inhibitor to HIV-1 reverse transcriptase [Green et al., 1995] and a pseudoknot binder of biotin [Wilson and Szostak, 1995]. The three dimensional x-ray crystal structure of the biotin binder is nearly solved [Nix et al., 1999, Wilson et al., 1998].

Stochastic context-free grammars (SCFGs) form the basis of our modeling. SCFGs are an extension of hidden Markov models (HMMs) [Rabiner and Juang, 1986, Rabiner, 1989] which have enjoyed much success in the modeling biological sequence structure [Krogh et al., 1994b, Krogh et al., 1994a, Baldi et al., 1994b,

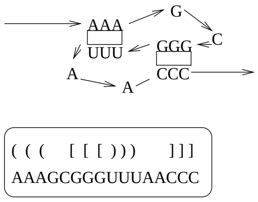


Figure 6.1: An RNA Pseudoknot. The basepairs formed by the A-U helix and the G-C helix are not well-nested. Bases within the hairpin loop of each helix interact with bases that are outside the loop. Abstractly this pairing is represented by "((([[[)))]]]" where "()" and "[]" represent two types of basepair interactions. A well-nested interaction is "((([[[]]])))".

Baldi and Chauvin, 1994, Baldi et al., 1994a, Tanaka et al., 1993, Raman and Overton, 1994, Fujiwara et al., 1994, Sonnhammer et al., 1997, Amtai, 1998, Park et al., 1998, Baldo and McClure, 1999, Bertilsson et al., 1998]. SCFGs have been used to model mainly RNA structure. Recent examples include modeling t-RNA [Eddy and Durbin, 1994, Sakakibara et al., 1994a, Grate et al., 1994, Lefebvre, 1995] and spliceosomal snRNA [Underwood, 1994]. Both SCFGs and HMMs models are grounded in a rigorous probabilistic framework which captures in a cohesive framework aspects of sequence similarity such as sequence mutation, deletion, and insertion.

6.1 Methods

6.1.1 Pseudoknot Modeling Problem and Solution

There are problems with modeling pseudoknots. The basic structure of a pseudoknot can be represented abstractly by the language, $\{w|w \text{ has the form } [^i(^j]^i)^j\}$ [1]. Here $[,]$ and $(,)$ denote abstract base-paired bases. This language is not context-free by the pumping lemma [Hopcroft and Ullman, 1979]. There is NO context-free grammar that can generate this language. However, consider the following language,

$$\{w|w \text{ has the form } [^i(^*]^i)^*\} \cap \{v|v \text{ has the form } [^*(^j]^*)^j\}$$

The intersection of these two languages is exactly the pseudoknot language.

Note that this argument uses context-free languages without a stochastic component. When modeling RNA, we do not have the alphabet, $[,],(,)$ but rather the symbols, A, U, G, C which may either basepair or not. To see how this argument relates to stochastic context-free grammars with the alphabet, A, U, G, C consider the language, $\{w|w \text{ has the form } A^iG^*U^iC^*\} \cap \{v|v \text{ has the form } A^*G^jU^*C^j\}$. This language is the same except the alphabet has been changed. This can be interpreted as the language describing pseudoknots in which one helix is composed entirely of $A-U$ basepairs and the other helix is composed entirely of $G-C$ basepairs. The stochastic context-free grammars which make up this intersection would have probability one for $A-U$ basepairs in one grammar and probability one for $G-C$ basepairs in the other. Obviously, in nature, there will rarely be instances in which a basepair will be $G-C$ with probability one. To make the model more realistic, relax the distribution on basepairs to allow for any basepair (each having its own respective probability). Now a sequence can be classified as a pseudoknot if it can be identified as having one helix which has high probability under one statistical model and simultaneously having the other helix which has high probability under the other statistical model.

This intersection argument is important for efficiency reasons. Consider designing a non-context-free pseudoknot grammar that could generate the pseudoknot language. Normally, in a context-free grammar, a

[1]The terminology $]^i$ represents the symbol,] repeated i times. $]^4 =]]]]$

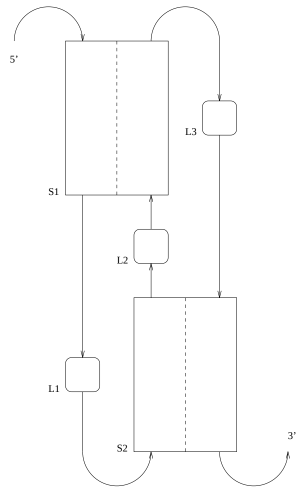


Figure 6.2: A pseudoknot abstraction. Rounded boxes represent loops and rectangles represent helixes.

nonterminal derives a subword, $i \dots j$. However, in pseudoknots, the derivations overlap because the helices are not properly nested and a nonterminal must derive the "subword", $(B \dots b)(e \dots E)$ where $B \dots b$ and $e \dots E$ are not contiguous in the string. In order to determine the running time of a CYK-like parser for such a language consider the running time of context-free grammar parser. The context-free grammar parser must fill in a table, $T[s, i, j]$, which is the probability of the nonterminal s deriving the subword $i \dots j$. Assume that there are T nonterminals and the length of the string to be parsed is N. There are order (TN^2) entries to be filled. The pseudoknot grammar parser, on the other hand, must fill in a table, $T[s, B, b, e, E]$, in which there are order (TN^4) entries. Assuming that $T \approx N$ then the running time is at least $O(N^5)$. This is prohibitive computationally. However, because of the intersection argument used above, the pseudoknot language can be approximated by using four $O(n^3)$ context-free parsing operations. The sense of this approximation is described below.

6.1.2 Approximating a Pseudoknot

Consider a pseudoknot with two helixes, $S1$ and $S2$, and three loops, $L1, L2, L3$, placed between the helixes. See Figure 6.2. Note that this paper deals with problems in which $L1, L2$, and $L3$ are simple loops. However, in general, $L1, L2$, and $L3$ can be arbitrarily complex stochastic context-free grammars containing multiple stem loop complexes. A full pseudoknot grammar can be decomposed into two different context-free grammars which will be called the 5' grammar and the 3' grammar. The 5' grammar breaks the helix closest to the 3' end, keeping the 5' helix intact. The 3' grammar breaks the helix closest to the 5' end, keeping the 3' helix intact. See Figure 6.3. Breaking a helix involves converting a basepairing production like $s \rightarrow AU$ into two productions like $g \rightarrow A$ and $h \rightarrow U$ in which the two halves of the basepair are derived independently. If the probabilities on basepairs for the nonterminal s is given in $P[i, j]$ where $i, j \in \{\text{A,U,G,C}\}$, then the probabilities for g and h should be marginal distributions. Probabilities for g will be given by $P[i] = \sum_j P[i, j]$ while the probabilities for h will be given by $P[j] = \sum_i P[i, j]$.

In this way, the pseudoknot grammar can be decomposed into two SCFGs and allow $O(n^3)$ parsing algorithms to be used to align the model instead of the $O(n^5)$ operation which would be required if the full pseudoknot grammar were used.

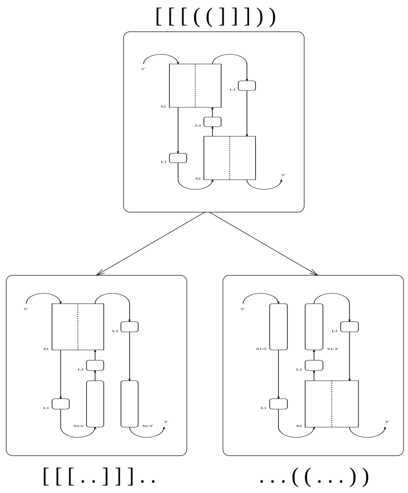


Figure 6.3: A pseudoknot broken into two grammars.

6.1.3 Finding Probabilities and Predicted Structures of Sequences

In order to perform a database search, a probability for each of the sequences must be computed given the model. The true probability of the sequence under the full pseudoknot grammar still appears to be difficult to compute exactly in $O(n^3)$ time. Therefore we compute a Viterbi-like approximation to this true probability. This approximate probability is computed in a seven stage process.

1. Compute the probability of the sequence under the 5' grammar storing that probability and the location of the 5' helix.

2. Compute the probability of the sequence under the 3' grammar storing that probability and the location of the 3' helix.

3. Compute the conditional probability of the sequence under the 5' grammar given that the 3' broken helix approximation is forced to be in the position specified by the 3' grammar parse. Store the location of the 5' helix under this constrained parse.

4. Compute the final 5' conditional probability by using the locations of the 3' and 5' helixes to compute the probability under the full pseudoknot grammar.

5. Compute the conditional probability of the sequence under the 3' grammar given that the 5' broken helix approximation is forced to be in the position specified by the 5' grammar parse. Store the location of the 3' helix under this constrained parse.

6. Compute the final 3' conditional probability by using the locations of the 3' and 5' helixes to compute the probability under the full pseudoknot grammar.

7. Return the largest of the final 3' and 5' conditional probabilities as the true probability of the sequence under the pseudoknot grammar.

Note that this procedure uses the "efficient" SCFG parses to place the constituent grammar features and then uses these placements to score the sequence under the full pseudoknot grammar. The predicted structure of the sequence can be easily constructed once parsing has finished. Simply take the parse of the most likely grammar and record where the helixes and loops are placed. This gives the complete predicted structure of the pseudoknot.

6.1.4 Database Searching Using the Pseudoknot Model

In order to perform a database search for pseudoknots we must, for each sequence in the database, classify the sequence as being generated by the pseudoknot model or as being generated by a competing NULL model [Duda and Hart, 1973, Rissanen, 1989].

This classification can be done using Bayes decision theory by considering the following function:

$$g(sequence) = \log \frac{P(sequence|pseudoknot)}{P(sequence|NULL)} + \log \frac{P(pseudoknot)}{P(NULL)}$$

where $P(sequence|pseudoknot)$ is the probability of the sequence under the pseudoknot model as approximated in the procedure described in the previous section, $P(sequence|NULL)$ is the probability of the sequence under the null model which is simply a uniform multinomial distribution over bases, $P(pseudoknot)$ is the prior probability of the pseudoknot model, and $P(NULL)$ is the prior probability of the null model.

If the function g is bigger than zero then the *pseudoknot* model is the "best" hypothesis otherwise the $NULL$ model is "better". For fixed priors, this reduces to computing the first term, $\log \frac{P(sequence|pseudoknot)}{P(sequence|NULL)}$ which is equal to $(-\log P(sequence|NULL) + \log P(sequence|pseudoknot))$.

This is interpreted as the savings in bits of the pseudoknot model over the $NULL$ model in encoding the sequence. This savings in bits in encoding a sequence will be called the *score* of the sequence. High scores imply the sequence was more likely generated by the *pseudoknot* model while low scores imply the sequence was more likely generated by the $NULL$ model.

If we knew, *a priori*, the probability of the pseudoknot model and of the NULL model, we could use $g(sequence)$ to perform a database search by calculating the first term by computing the probabilities using the models, finding the second term which is a cutoff value using prior information, and then adding them to see how their sum compares to zero. Unfortunately our prior knowledge does not allow us to find the second term accurately. Instead, we use an empirical method which estimates the second term. This estimation is done by computing the scores of many random sequences. We estimate the cutoff value used in the function g as the savings in bits such that all the random sequences have scores less than that number.

Therefore to perform a database search on a long string (or even an entire genome), the string is broken into overlapping windows. For our software, a window size of 128 runs the fastest. As described above, a score is computed for each of these windows and a cutoff value is applied so that only those windows which have a high enough score are accepted. After this cutoff is made, the predicted structure for all accepted sequences is generated and analyzed.

6.1.5 Parameter Estimation of the Pseudoknot Model

Estimating parameters of a pseudoknot grammar using intersections of SCFGs involves combining the estimates from the two constituent SCFGs into estimates for the pseudoknot grammar and then generating two SCFGs from these combined counts. These estimates are based on frequencies of occurrence of various events, such as the number of times a certain production was used.

Computing these estimates for SCFGs can be performed with an algorithm called the inside/outside algorithm [Lari and Young, 1990]. Both the inside and outside variables can be computed in $O(n^3)$ time. These variables can be used to estimate the probability of a production being used. A more computationally efficient SCFG parameter estimation technique has been developed [Sakakibara et al., 1994b]. This algorithm

```
   ((((( [[[[[[[)))))                              ]]]]]]]
GGCACCGACCATAGGCTCGGGTTGCCAGAGGTTCCACACTTTCATCGAAAAGCCTATGCTA
```

Figure 6.4: A pseudoknot for binding biotin.

is called the Tree-Grammar EM training algorithm. This algorithm uses labeled trees as input rather than observation sequences. These labeled trees represent "folded" sequences which are a set of trees in which the frontier, or leaves, of the tree are known but the internal nodes are not. The algorithm is able to estimate parameters of a grammar given the trees in $O(n^2)$ time.

Once these estimates are computed for the constituent SCFGs, the question is how to combine the counts. One heuristic that is used is as follows: For loop regions, simply combine the counts for the respective loop regions from the two SCFGs. The loop regions should coincide between the two SCFGs so they should share parameters and therefore the estimates are added. For broken helixes, counts from the broken helix grammar and marginal counts from the full helix are combined. The broken helix is represented as a broken helix in one grammar and as one half of a full helix in the other grammar. Therefore the full counts from the broken helix and the marginal counts from the full helix are combined. For full helixes, the counts for the full helix are used while the counts for broken helix are not. For full helixes, we found that the broken helix approximation did not produce as good estimates as the full helix representation and therefore estimates from the broken helix approximation are not used. Once estimate counts have been combined, two SCFGs are produced, each of which approximates one of the helixes in the pseudoknot with independently derived helix approximations. This heuristic has been implemented as an extension to the Tree-Grammar EM training algorithm.

6.2 Results

6.2.1 Biotin Binder Pseudoknot

Our methods were first tested on modeling RNA sequences which contained pseudoknots which bind biotin [Wilson and Szostak, 1995, Nix et al., 1999, Wilson et al., 1998]. These sequences are important because of their potential support of the 'RNA World' model [Benner et al., 1989]. The RNA World model proposes that much of modern metabolism evolved prior to the evolution of encoded protein synthesis, and that early ribozyme-catalyzed metabolic transformations form the basis of our present protein-catalyzed metabolism. This proposal requires that ribozymes should be able to catalyze a broad range of chemical transformations. The search for "new" ribozymes leads us to the biotin binder RNA sequences. These sequences were generated from *in vitro* selection experiments. A pool of approximately 5×10^{14} different random sequence RNAs was generated by *in vitro* transcription of a DNA template containing a central 72-nucleotide random sequence region, flanked at both ends by 20-nucleotide constant regions. On average, any given 28 nucleotide sequence has a 50% probability of being represented in a pool of this complexity. This pool was applied to an agarose column derivatized with 2-6 mM biotin and then washed with 15 column volumes of binding buffer. Specifically-bound RNAs were then eluted by washing the column with binding buffer containing 5 mM biotin. Reverse transcription of the specifically eluted RNA, followed by PCR amplification and *in vitro* transcription yielded an enriched pool of RNA for subsequent re-selection.

After several rounds of selection, the RNA was sequenced. The biotin binder structure is short and contains one pseudoknot. See Figure 6.4 for a representative sequence.

A model for the biotin binder was constructed by analyzing the sequences derived from the *in vitro* selection experiments and several mutational experiments. This data was used to construct the overall structure

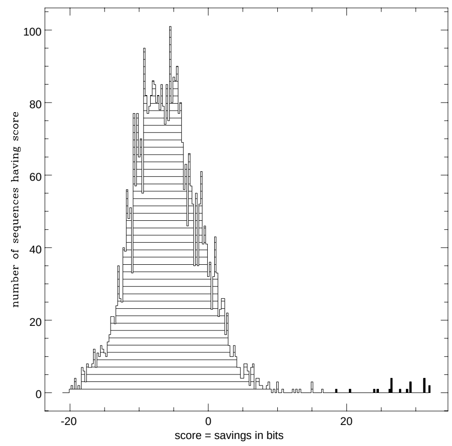


Figure 6.5: A histogram showing number of sequences having a certain sequence score. Dark areas are biotin binders. Light areas are random sequences.

of the model. The biotin binder sequences were then used as a training set and parameters of the model were estimated using an extension of the Tree-Grammar EM training algorithm [Sakakibara et al., 1994a]. This model was then hand tuned using biological knowledge to capture the structure of the biotin binder sequence without over-fitting the biases which were present in the training sequences.

In order to judge significance of sequence scores, the model was used to compute the scores of 4096 random sequences which had lengths uniformly distributed over $60 \ldots 79$ and had a uniform distribution over bases. The average score of these random sequences was -6.03836 with a variance of 4.9239. The highest score of a random sequence was 16.571716.

The model's discriminative power was first tested by computing the scores of the biotin binding sequences which were generated by the *in vitro* selection experiments. The highest score of a biotin binder was 31.919823. The lowest score was 18.493336. See Figure 6.5 for a histogram of scores for both the random and biotin binding sequences. This figure shows that choosing a cutoff between 16.58 and 18.49 (almost 2 bits of separation between random and biotin binding sequences) will discriminate between biotin binders and random sequences. This cutoff would correspond to the *a priori* belief that there is approximately one biotin binder for every 2^{17} sequences in a database search.

Our database search involved searching through all GenBank database sequences which had the keyword "biotin". These sequences were broken into overlapping windows of length 76 yielding 1737 windows. Scores were computed for each of the windows and a cutoff of 16.75 was chosen. Only one window had a score above this cutoff. The score of this window is 17.023933. This window corresponded to the sequence for the gene for biotin carboxyl carrier protein in Mycobacterium leprae (EMBL: MLBCCPG. Accession: X63470).

The pseudoknotted structure started at position 449 in this sequence (the position of the 5' base in the first basepair) and has the following predicted structure:

```
     [[[[     {{{{]]]]       }}}}
cacaatcgatccgcgacctcggcgacaaggtcaccgc
```

This sequence's predicted structure is similar to other biotin binding RNA. To test whether this sequence could bind biotin, an experiment similar to the *in vitro* selection process described above was performed. A pool made up of this sequence was applied to a column derivatized with biotin and then washed with binding buffer. Then the column was washed with biding buffer containing biotin to elute the specifically-bound RNA. Unfortunately there was not any significant binding of the RNA and attempts to optimize the experiment by changing temperature and Mg concentration did not yield any more promising results.

6.2.2 Importance of Intersected SCFGs

The pseudoknot structure cannot be modeled by one SCFG alone. The processes of constraining the parses in the intersection yields discriminative power. In order to show this, the 5' and 3' grammars were not intersected but were used alone to perform discrimination. Again, as with the intersected model, a cutoff value must be chosen. This was done by scoring random sequences described in the previous section and recording the highest scoring sequences. For the 5' grammar the highest scoring random sequence was 14.012839. For the 3' grammar the highest scoring random sequence was 15.061969. These two numbers serve as the cutoff values for the two respective grammars.

The biotin binding sequences derived from *in vitro* selection were then analyzed. Whereas the intersected model was able to completely discriminate these sequences from random by choosing a cutoff anywhere between 16.58 and 18.49 (as much as a 2 bit difference), the 5' and 3' models alone could not completely discriminate these sequences. The 5' model had ten sequences below the cutoff. The highest scoring of these was 13.630875 (0.4 bit below cutoff). The 3' model did much better having only one sequence with a score below the cutoff. That score was 14.672070 (0.3 bit below cutoff). This shows that the 3' model contains much of the information as captured by the intersected model, but not enough to discriminate true biotin binding sequences. The intersection process must be used to get maximum discrimination.

6.3 Conclusion

A new method for modeling RNA pseudoknots using intersections of stochastic context-free grammars has been presented. This method is novel in its use of intersected SCFGs for approximating the location of the pseudoknot. The model has successfully discriminated biotin binding RNA from random RNA sequences and has found in a database search a sequence which closely resembles biotin binders derived from *in vitro* selection experiments. The pseudoknot model gives discriminative power which is greater than simple stochastic context-free grammars when modeling pseudoknots.

Appendix A

RNACAD Programs

These are the programs used in the RNACAD system. The programs are listed alphabetically grouped by function: constraintTools, modelTools, parser, perfromanceTools, SAMTools, seqTools.

Each entry gives a synopsis of how to run the program, a description of what the program does, parameters the program takes, exceptions that the program throws, and resources the program needs.

Parameters are given in the form:

```
(from) = [I|O] type name
   description
```

(from) specifies how the parameter is given command line option (CLOption), position on command line (argv0), or CGI variable (CGI). `[I|O]` is I if the parameter is an input and O if it is output. name is the name of the object so that it can be referred to in the description. The name is @ if the name is implicit as when an object is printed to standard out in which case standard out is the object and a name pointing to a file containing the object is not applicable.

Resources are given in the form:

```
[C|I|O] type name
   description
```

`[C|I|O]` is C if the resource is a program call such as an awk script, I if the resource is input, O if the resource is output. If a program writes a result file under some naming convention this will be identified by an O resource with a given name.

This documentation is automatically generated from the special documentation embedded in the source files themselves and is modeled after the `javadoc` system from Sun Microsystems.

A.1 constraintTools

A.1.1 corresp.pl — Two sequence correspondence.

Synopsis

corresp.pl |*alignment*| |*runLength*| > |*correspond*|

Description

Generates the correspondence between the two sequences given |*alignment*| and the minimum run of exact matches, |*runLength*|. A corespondence is a list of lines having the form: (refBase examBase examBaseID). refBase is the base number in the reference sequence. examBase is the base number in the examined sequence. examBaseID is the identity of the examBase. (refBase -1) implies there no correspondence to the reference sequence (deleted). If |*runLength*| is zero then if reference sequence has base and examined sequence has indel then (refBase -1) is generated. Checks for illegal characters but this information is usually lost because errors are only printed to stdout.

Parameters

(argv0) = I fastaMA |*alignment*|
The file containing the alignment of two sequences.

(argv1) = I int |*runLength*|
The run of exact matches that must occur before the region is made to correspond.

(stdout) = O tdcorrespond @
The correspondence between the two sequences.

Exceptions

IllegalChar
Illegal sequence character.

Resources

C perl @
Perl file.

A.1.2 loosenConstraints.awk — Make constraints looser.

Synopsis

gawk -f loosenConstraints.awk |*constraints*| > |*newconstraints*|

Description

Given a constraint file subtract 10 from left sides and add 10 to right sides.

Parameters

(argv0) = I tdconstraint |*constraints*|
The original constraint file.

(stdout) = O tdconstraint @
The loosend constraints.

Resources

C gawk @
Gawk file.

A.1.3 maconst.pl — Generate constraints from multiple alignment.

Synopsis

maconst |*alignment*| |*refParse*| |*runLength*| |*destSeqs*| |*destConst*|

Description

Generates constraints and sequences given the |*alignment*| and the parse of the reference sequence, |*refParse*|. A constraint is generated if it occurs in a |*runLength*| or more exact matches in the alignment. If the reference parse has nonterminal T deriving base i and the alignment puts base j of another sequence in the same column of the reference sequence's base i then a constraint for that nonterminal T deriving sequence's base j is made. In basepairs, both sides of a basepair must be in correspondence. No one sided constraints are made. If a constraint is made then the corresponding delete nonterminal is made to be impossible. Assumes reference sequence is first in the alignment and doesn't output constraints for it.

Parameters

(argv0) = I fastaMA |*alignment*|
The multiple alignment of the sequences with the reference sequence first.

(argv1) = I tdparse |*refParse*|
The parse of the reference sequence.

(argv2) = I int |*runLength*|
The minimum length of exact match region for constraints to be made.

(argv3) = O tdsequences |*destSeqs*|
The file where the raw sequences from the alignment should be written.

(argv4) = O tdconstraints |*destConst*|
The file where the constraints should be written.

Environment

corresp_pl
Internal variable giving location of corresp.pl.

resmap_pl
Internal variable giveing location of resmap.pl.

Resources

C perl @
Perl file.

O fastaMA two_align
Temporary two sequence alignment file.

O tdcorrespond correspond
Temporary correspond file.

O tdconstraints pre.constraints
Temporary constraint file.

C corresp.pl @
Call to corresp.pl.

C resmap.pl @
Call to resmap.pl.

C rm, cat, sed @
Shell call.

A.1.4 makepdconst.pl — Generate SCFG constraints for sequences from HMM.

Synopsis

makepdconst.pl |*hmm*| |*refSeq*| |*examSeqs*| |*window*| |*refParse*| |*destConst*|

Description

Generates SCFG constraints for sequences given HMM model. Gets posterior decode of sequence from HMM and maps correspondence between reference and examine bases with high posterior. Generates constraints from this correspondence and loosens them.

Parameters

(argv0) = I SAMHMM |*hmm*|
The HMM producing the posterior decode. It is usually an approximation to the SCFG.

(argv1) = I fastaSeq |*refSeq*|
The file containing the reference sequence.

(argv2) = I fastaSeqs |*examSeqs*|
The file containing the sequences to be examined.

(argv3) = I integer |*window*|
The maximum length window allowed for which there are no constraints. (256...)

(argv4) = I tdparse |*refParse*|
The parse of the reference sequence.

(argv5) = O tdconstraints |*destConst*|
The destination file for the constraints.

Environment

stoccfg
: Environment variable giving path to the RNACAD system under which the program directory resides.

Resources

C perl @
: Perl file.

C resmap.pl, grabdp, post_corresp.pl, loosenConstraints.awk @
: RNACAD calls.

C rm
: System call.

O miscellaneous examine.seq ,correspond, pre.constraints, preloose.constraints
: Temporary files.

A.1.5 post_corresp.pl — Generate two sequence correspondence given HMM PD.

Synopsis

post_corresp.pl |*refpd*| |*exampd*| |*window*|

Description

Generate correspondence between the two sequences given SAM HMM posterior decodes for both sequences. |*window*| gives maximum allowed region length with no constraints. Uses a greedy approach. Uses all correspondences with posterior decode greater than 0.8. For regions with no existing correspondences uses a sliding window of length |*window*| and takes the maximum posterior decode correspondence in each unconstrained window.

Parameters

(argv0) = I hmmPD |*refpd*|
: The posterior decode for the reference sequence.

(argv1) = I hmmPD |*exampd*|
: The posterior decode for the examined sequence.

(argv2) = I integer |*window*|
: The maximum allowed unconstrained window.

Resources

C perl @
: Perl file.

A.1.6 resmap.pl — Generate constraints from correspondence.

Synopsis

resmap.pl |*refParse*| |*corresp*| |*outConstraints*|

Description

Generates constraints given the parse of the referenece sequence, |*refParse*| and the correspondence between the reference sequence and the examined sequence, |*corresp*|. For loops: correspondence, (ref -1), implies nonterminal impossible. Legal correspondence implies nonterminal delete impossible. For helix: correspondence, (ref -1), on either side of the helix implies nonterminal impossible. Leagal correspondence implies nonterminal delete impossible. No one-sided constraints for helix are made.

Parameters

(argv0) = I tdparse |*refParse*|
The parse of the reference sequence.

(argv1) = I tdcorrespond |*corresp*|
The correspondence between the reference sequence and the examined sequence.

(argv2) = O tdconstraints |*outConstraints*|
The file for the resulting constraints.

Resources

C perl @
Perl file.

A.2 modelTools

A.2.1 align — Generate multiple alignment.

Synopsis

align |*parses*| > |*MA*|

Description

Generates a FASTA multiple alignment of the sequences contained the the |*parses*| file. Capital letters represent matches, lower-case represent inserts, '-' represents delete, and '.' represents insert spacing. First line is the structure line. '-' means loop match/delete, '.' means loop insert, '[' means 5' side of match/delete helix, ']' means 3' side of match/delete helix, '(' means 5' side of helix insert, ')' means 3' side of helix insert.

Parameters

(argv0) = I tdparse |*parses*|
The parses of the sequences.

(stdout) = O fastaMA @
The FASTA multiple alignment.

A.2.2 bootstrapParseFromMA — Generate parses that reflect the alignment.

Synopsis

bootstrapParseFromMA ssu.ma E.coli.bp E.coli.seq

Description

Given a multiple alignment, ssu.ma, headed by the reference sequence and basepairing for the reference sequence, E.coli.bp, generate a set of parses that reflect the alignment and basepairing into the file ssu.parse.0.

Parameters

(argv0) = I fastaMA ssu.ma
The multiple alignment headed by the reference sequence ending in extension ".ma".

(argv1) = I tdbasepairs E.coli.bp
The list of basepairs for the reference sequence.

(argv2) = I sequence E.coli.seq
The sequence of the reference sequence.

Resources

C sh @
Shell script.

C rm @
Shell call.

C meta,htolow,collect,tdparse,maconst.pl,tdparse @
RNACAD calls.

I tdprior ssurrna.prior
The prior file for ssurrna.

O tdmeta ssu.meta
The meta grammar.

O tdgrammar ssu.tdgrammar.null
The estimated grammar for no sequences.

O tdconstraints E.coli.constraints
The constraints on the reference sequence.

O tdparse E.coli.parse
The parse of the reference sequence.

O fastaSeqs ssu.seq
The sequences in the multiple alignment.

O tdconstraints ssu.constraints.0
The constraints for all sequences in the multiple alignment.

O tdparse ssu.parse.0
The parses of all the sequences in the multiple alignment.

removed O tdgrammar ssu.tdgrammar
The translation from the meta grammar.

A.2.3 htolow — translate high meta grammar to low.

Synopsis

htolow |*metaGrammar*| > |*lowGrammar*|

Description

Generates the low level tdgrammar from the high level meta grammar. See description of high level grammar.

Parameters

(argv0) = I tdmetaGrammar |*metaGrammar*|
The metagrammar.

(stdout) = O tdgrammar @
The tdgrammar translation.

A.2.4 meta — Construct grammar from alignment and basepairs.

Synopsis

meta |*alignment*| |*basepairs*| |*outMeta*| |*refName*|

Description

Generates a high level meta grammar from an alignmment and a list of basepairs for a reference sequence in the alignment. Also outputs constraints on the reference sequence in which the correct nonterminals output the correct terminals of the reference. These constraints are for nonterminals in the low level grammar which is the "htolow" translation of the meta grammar. The program determines which columns in the alignment contain enough information for modeling and then outputs the meta grammar to |*outMeta*| and the reference constraints to the file "|*refName*|.constraints". A column has enough information if the ratio of bases to indels is greater than 0.5 and the information for those bases is greater than 0.322. Assumes the reference sequence is first sequence in the multiple alignment and that the sequence is composed of ACGT, all other characters are treated as indels.

Parameters

(argv0) = I fastaMA |*alignment*|
The multiple alignment of the form described in the description.

(argv1) = I tdbasepairList |*basepairs*|
A list of basepairs for the reference sequence.

(argv2) = O tdmetaGrammar |*outMeta*|
The destination file for the meta grammar.

(argv3) = I string |*refName*|
The name of the reference sequence.

A.2.5 parseSeqsGivenHMMSCFG — parse sequences given SCFG and HMM.

Synopsis

parseSeqsGivenHMMSCFG |*seqs*| |*hmm*| |*scfg*|.tdgrammar.|*num*| |*refSeq*| |*refParse*|

Description

Parse sequences given HMM and SCFG. Needs reference sequence and parse. The name of the SCFG is assumed to be |*scfg*|.tdgrammar.|*num*| where |*scfg*| is a name and |*num*| is an integer representing a version number. Resulting parses are stored in |*scfg*|.parse.|*num+1*|.

Parameters

(argv0) = I fastaSeq |*seqs*|
The sequences to be parsed.

(argv1) = I HMM |*hmm*|
The HMM;

(argv2) = I tdgrammar |*ssu*|**.tdgrammar.**|*num*|
The tdgrammar with structured name.

(argv3) = I fastaSeq |*refSeq*|
The reference sequence.

(argv4) = I tdparse |*refParse*|
The parse of the reference sequence.

Resources

C perl @
Perl file.

C makepdconst.pl,tdparse @
RNACAD call.

O tdconstraints |*scfg*|**.constraints.**|*num+1*|
The HMM posterior decoded constraints for all the sequences.

O tdparse |*scfg*|**.parse.**|*num+1*|
The resulting parses.

A.2.6 parseToAlignHMMTdgrammar — generate alignment, HMM, and SCFG from parses

Synopsis

parseToAlignHMMTdgrammar |*name*|.parse.|*num*| |*SCFG*|

Description

Generates alignment, HMM, and SCFG from parses. Assumes parses have structured name. Generates alignment in |*name*|.align.|*num*|, SCFG in |*name*|.tdgrammar.|*num*|, and HMM in |*name*|.hmm.|*num*|.mod.

Parameters

(argv0) = I tdparse |*name*|**.parse.**|*num*|
The parses with a structured name.

(argv1) = I tdgrammar |*SCFG*|
The grammar used to generate the parses.

Resources

C perl @
Perl file.

C tail,rm @
System calls.

C modelfromalign
SAM call.

C align,collect @
RNACAD calls;

O fastaMA |*name*|**.align.**|*num*|
The alignment.

O tdgrammar |*name*|**.tdgrammar.**|*num*|
The SCFG.

I tdprior ssurrna.prior
The priors for ssurrna.

O samHMM |*name*|**.hmm.**|*num*|**.mod**
The HMM.

A.3 parser

A.3.1 collect — Estimate grammar from set of parses.

Synopsis

collect |*scfg*| |*parses*| |*prior*| [|*seqWeights*|] > |*outSCFG*|

Description

Generate estimated grammar given parses, prior, and optional seqWeights. Uses parses and collects (counts) number of times productions are used, adds prior, and normalizes. An optional form: collect |*grammar*| |*parses*| > |*outputCounts*| is used when estimating priors.

Parameters

(argv0) = I tdgrammar |*scfg*|
The grammar to reestimate.

(argv1) = I tdparses |*parses*|
The parses used as positive examples.

(argv2) = I tdprior |*prior*|
The prior.

(argv3) = optional I tdseqWeights |*seqWeights*|
Optional sequence weighting. Adds weight instead of 1.0.

(stdout) = O tdgrammar |*outSCFG*|
The estimated grammar.

A.3.2 tdparse — parse sequence(s) given grammar.

Synopsis

tdparse |*scfg*| |*seqs*| [|*constraints*|] > |*outParses*|

Description

Parse sequences given grammar and optional constraints and print to stdout. An optional testing form is tdparse |*tdgrammar*| > |*tdgrammar*| which reads in the grammar and outputs it back out.

Parameters

(argv0) = I tdgrammar |*scfg*|
The grammar.

(argv1) = I fastaSeqs |*seqs*|
The sequences.

(argv2) = optional I tdconstraints |*constraints*|
Optional constraints.

A.4 performanceTools

A.4.1 avgscore.pl — Generates score statistics from parses.

Synopsis

avgscore |*parses*|

Description

Returns the max, min, and average over number of sequences and number of bases the savings in bits for the parses.

Parameters

(argv0) = I tdparse |*parses*|
The parses.

(argv1) = O string @
Report of findings.

A.4.2 bponly.awk — converts full basepair constraints to only basepairs.

Synopsis

cat |*fullbasepairs*| — gawk -f bponly.awk > |*bponly*|

Description

Generates basepairs only: (>seq i/j) where i and j are the 5' and 3' sides from full basepair constraints (seq nt 5' 5' 3' 3' helix res).

Parameters

(stdin) = I fullbp @
The full basepairs.

(stdout) = O bponly @
basepairs only.

A.4.3 compareBPOnly.pl — Compare BP truth and prediction.

Synopsis

compareBPOnly.pl |*truth*| |*prediction*|

Description

Given basepairs only for truth and prediction compute fn, fp, tp.

Parameters

(argv0) = I bponly |*truth*|
The truth for basepairs.

(argv1) = I bponly |*prediction*|
The prediction for basepairs.

(stdout) = O string @
The report for fn, fp, tp.

Resources

C perl @
Perl file.

C sort,comm,wc,rm @
System calls;

A.4.4 consttobp.pl — Generate basepairs from constraint file.

Synopsis

consttobp.pl |*constraints*| > |*bp*|

Description

Generates basepair info from given constraints. Used for checking bp accuracy.

Parameters

(argv0) = I tdconstraints |*constraints*|
The constraints.

(stdout) = O bpConstraints @
The basepair information.

Resources

C perl @
Perl file.

A.4.5 parsetobp.pl — Generate basepairs from parses.

Synopsis

parsetobp.pl |*parses*| > |*bp*|

Description

Generates basepair info from given parses. Used for checking bp accuracy.

Parameters

(argv0) = I tdparse |*parses*|
The parses.

(stdout) = O bpConstraints @
The basepair constraints information.

Resources

C perl @
Perl file.

A.5 samTools

A.5.1 grabdp — Produces max posterior decode for sequence given HMM

Synopsis

grabdp |*name*| -i |*hmm*| -db |*seq*|

Description

Generates max posterior decode for sequence given HMM. Stored in file |*name*|.pdoc. A max PD file consists of lines (baseid basenum modelnum posterior) with "%" being the comment character. Baseid is the identity of the base (AGCT), basenum is the number of the base in the sequence (indexed from zero), modelnum is the maximal posterior decoded model state number that generated the base, posterior is the value of the max posterior. The source modified SAM forward2.c to update the maxPost and max index (check freq->node[ii]->match[c] and update it and ii), and grabdp.c to output the correct format.

Parameters

(CLOption -i) = I HMMModel |*hmm*|
The HMM

(CLOption -db) = I fastaSeqs |*seq*|
The sequence.

Resources

O samPD |*name*|**.pdoc**
The max posterior decode file.

A.6 seqTools

A.6.1 genbankToFasta.pl — genbank to fasta

Synopsis

cat |*gb*| — genbankToFasta.pl > |*fasta*|

Description

Converts sequence format genbank to fasta. Fasta ID is LOCUS first field. Bases between ORIGIN and // are outputted, 0-9 and spaces removed, uppercased, and U to T,   to -.

Parameters

(stdin) = I genbank @

(stdout) = O fastaSeq @

A.6.2 pullFastaSeqs.pl — pull sequences that match IDs.

Synopsis

cat |*seqs*| — pullFastaSeqs.pl |*IDs*| > |*outSeqs*|

Description

Output sequences from |*seqs*| that have and ID in the ID list, |*IDs*|. ID list is line list of ID without the ">" character.

Parameters

(stdin) = I fastaSeqs @

(argv0) = I IDlist |*IDs*|

(stdout) = O fastaSeqs @

Bibliography

[Altschul et al., 1989] Altschul, S. F., Carroll, R. J., and Lipman, D. J. (1989). Weights for data related by a tree. *JMB*, 207:647–653.

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Meyers, E. W., and Lippman, D. J. (1990). Basic local alignment search tool. *JMB*, 215:403–410.

[Amtai, 1998] Amtai, M. (1998). Hidden modlels in biopolymers. *Science*, 282(5393):1436.

[Baase, 1988] Baase, S. (1988). *Computer Algorithms*. Addison Wesley.

[Baldi et al., 1994a] Baldi, P., Brunak, S., Chauvin, Y., Engelbrecht, J., and Krogh, A. (1994a). Hidden Markov models of human genes. In J.Cowan, Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems*, volume 6, pages 761–768. Morgan Kaufmann.

[Baldi and Chauvin, 1994] Baldi, P. and Chauvin, Y. (1994). Smooth on-line learning algorithms for hidden Markov models. *Neural Computation*, 6(2):305–316.

[Baldi et al., 1994b] Baldi, P., Chauvin, Y., Hunkapillar, T., and McClure, M. (1994b). Hidden Markov models of biological primary sequence information. *PNAS*, 91:1059–1063.

[Baldo and McClure, 1999] Baldo, A. and McClure, M. (1999). Evolution and horizontal transfer of dutpase-encoding genes in viruses and their hosts. *J. Virology*, 73(9):7710–7721.

[Ban et al., 1999] Ban, N., Nissen, P., Hansen, J., Capel, M., Moore, P., and Steitz, T. (1999). Placement of protein and rna structures into a 5 angstrom resolution map of the 50s ribosomal subunit. *Nature*, 400:841–847.

[Bender, 1996] Bender, E. (1996). *Mathematical methods in artificial intelligence*. IEEE Computer Society.

[Benner et al., 1989] Benner, S., Ellington, A., and Tauer, A. (1989). Modern metabolism as a palimpsest of the rna world. *PNAS*, 86:7054–8.

[Bertilsson et al., 1998] Bertilsson, G., Heidrich, J., Svensson, K., Asman, M., Jendeberg, L., Sydow-Backman, M., Ohlsson, R., Postlind, H., Blomquist, P., and Berkenstam, A. (1998). Identification of a human nuclear receptor defines a new signaling pathway for cyp3a induction. *PNAS*, 95(21):12208–12213.

[Brock et al., 1994] Brock, T., Madigan, M., Martinko, J., and Parker, J. (1994). *Biology of Microorganisms*. Prentice Hall.

[Brown et al., 1993] Brown, M. P., Hughey, R., Krogh, A., Mian, I. S., Sjölander, K., and Haussler, D. (1993). Using Dirichlet mixture priors to derive hidden Markov models for protein families. In Hunter, L., Searls, D., and Shavlik, J., editors, *ISMB-93*, pages 47–55, Menlo Park, CA. AAAI/MIT Press.

[Brown et al., 1999] Brown, M. P. S., Grundy, W. N., Lin, D., Cristianini, N., Sugnet, C., Furey, T. S., M.Ares, J., and Haussler, D. (1999). Knowledge-based analysis of microarray gene expression data using support vector machines. Technical Report UCSC-CRL-99-09, University of California Santa Cruz. Revised version submitted to PNAS.

[Brown and Wilson, 1995] Brown, M. P. S. and Wilson, C. (1995). Rna pseudoknot modeling using intersections of stochastic context free grammars with applications to database search. In Hunter, L. and Klein, T., editors, *Pacific Symposium on Biocomputing*, pages 109–125.

[Cary and Stormo, 1995] Cary, R. and Stormo, G. (1995). Graph-theoretic approach to rna modeling using comparative data. In *Intelligent Systems for Molecular Biology*, pages 75–90.

[Cate et al., 1999] Cate, J., Yusupov, M. M., Yusupova, G. Z., Earnest, T. N., and Noller, H. F. (1999). X-ray crystal structures of 70s ribosome functional complexes. *Science*, 285(5436):2095–2104.

[Chen et al., 1992] Chen, J.-H., Le, S.-Y., and Maizel, J. V. (1992). A procedure for rna pseudoknot prediction. *CABIOS*, 8:243–248.

[Clemons Jr. et al., 1999] Clemons Jr., W., May, J., Wimberyly, B., McCutcheon, J., Capel, M., and Ramakrishnan, V. (1999). Structure of a bacterial 30s ribosomal subunit at 5.5 Angstom resolution. *Nature*, 400:833–840.

[Coppersmith and Winograd, 1990] Coppersmith, D. and Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9:251–280.

[Corazza et al., 1991] Corazza, A., Mori, R. D., Gretter, R., and Satta, G. (1991). Computation of probabilities for an island-driven parser. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13:936–949.

[Corazza et al., 1994] Corazza, A., Mori, R. D., Gretter, R., and Satta, G. (1994). Optimal probabilistic evaluation functions for search controlled by stochastic context-free grammars. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:1018–1027.

[Corpet and Michot, 1994] Corpet, F. and Michot, B. (1994). Rnalign program: alignment of rna sequences using both primary and secondary structures. *CABIOS*, 10(4):389–399.

[Cover, 1991] Cover, T. M. (1991). *Elements of information theory.* Wiley.

[de Peer et al., 1997] de Peer, Y. V., Jansen, J., Rijk, P. D., and Wachter, R. D. (1997). The antwerp ribosomal rna database. *NAR*, 25:111–116.

[Dempster et al., 1977] Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the *EM* algorithm. *J. Royal Statistical Society B*, 39:1–38.

[Doolittle, 1999] Doolittle, W. (1999). Phylogenetic classification and the universal tree. *Science*, 284:2124–2128.

[Duda and Hart, 1973] Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis.* Wiley, New York.

[Durbin et al., 1998] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. (1998). *Biological sequence analysis : probabalistic models of proteins and nucleic acids.* Cambridge University Press.

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

[Felsenstein, 1981] Felsenstein, J. (1981). Evolutionary trees from dna sequences. *Journal of Molecular Evolution*, 17:368–376.

[Feng and Doolittle, 1987] Feng, D. F. and Doolittle, R. F. (1987). Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360.

[Fourmy et al., 1996] Fourmy, D., Recht, M. I., Blanchard, S. C., and Puglisi, J. D. (1996). Structure of the a site of escherichia coli 16s ribosomal rna complexed with an aminoglycoside antibiotic. *Science*, 274(5291):1367–1371.

[Fu, 1982] Fu, K. (1982). *Syntactic Pattern Recognition and Applications.* Prentice-hall.

[Fujiwara et al., 1994] Fujiwara, Y., Asogawa, M., and Konagaya, A. (1994). Stochastic motif extraction using hidden markov model. In *Intelligent Systems for Molecular Biology*, pages 121–129.

[Grate, 1999] Grate, L. (1999). *Computer Analysis of RNA.* PhD thesis, University of California Santa Cruz.

[Grate et al., 1994] Grate, L., Herbster, M., Hughey, R., Mian, I., Noller, H., and Haussler, D. (1994). RNA modeling using Gibbs sampling and stochastic context free grammars. In *Intelligent Systems for Molecular Biology*, pages 136–144, Menlo Park, CA. AAAI/MIT Press.

[Gray et al., 1999] Gray, M., Burger, G., and Lang, B. (1999). Mitochondrial evolution. *Science*, 283:1476–1481.

[Green et al., 1995] Green, L., Waugh, S., Binkley, J., Hostomska, Z., Hostomsky, Z., and Tuerk, C. (1995). Comprehensive chemical modification interference and nucleotide substitution analysis of an rna pseudoknot inhibitor to hiv-1 reverse transcriptase. *JMB*, 247:60–68.

[Gribskov et al., 1990] Gribskov, M., Lüthy, R., and Eisenberg, D. (1990). Profile analysis. *Methods in Enzymology*, 183:146–159.

[Gulko, 1995] Gulko, B. (1995). Using phylogenetic markov trees to detect conserved structure in rna multiple alinments. Master's thesis, UC Santa Cruz.

[Gutell, 1993] Gutell (1993). Comparitive analysis of rna. In Gestel and Atkins, editors, *The RNA World.* Addison-Wesley.

[Gutell, 1984] Gutell, R. (1984). *Comparative structural analysis of 16s ribosomal rna.* PhD thesis, UC Santa Cruz.

[Gutell et al., 1992] Gutell, R. R., Power, A., Hertz, G. Z., Putz, E. J., and Stormo, G. D. (1992). Identifying constraints on the higher-order structure of RNA: continued development and application of comparative sequence analysis methods. *NAR*, 20:5785–5795.

[Harrison, 1978] Harrison, M. A. (1978). *Introduction to Formal Language Theory.* Addison Wesley.

[Hopcroft and Ullman, 1979] Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[Hughey and Krogh, 1996] Hughey, R. and Krogh, A. (1996). Hidden Markov models for sequence analysis: Extension and analysis of the basic method. *CABIOS*, 12(2):95–107. Information on obtaining SAM is available at `http://www.cse.ucsc.edu/research/compbio/sam.html`.

[Jaakkola et al., 1999] Jaakkola, T., Diekhans, M., and Haussler, D. (1999). Using the Fisher kernel method to detect remote protein homologies. In *Intelligent Systems For Molecular Biology*.

[Jaeger et al., 1990] Jaeger, J. A., Turner, D. H., and Zuker, M. (1990). Predicting optimal and suboptimal secondary structure for RNA. *Methods in Enzymology*, 183:281–306.

[Jefferys and Berger, 1992] Jefferys, W. and Berger, J. (1992). Ockham's razor and bayesian analysis. *American Scientist*, 80:64–72.

[Juan and Wilson, 1999] Juan, V. and Wilson, C. (1999). Rna secondary structure prediction based on free energy and phylogenetic analysis. *JMB*, 289:935–947.

[Karchin and Hughey, 1998] Karchin, R. and Hughey, R. (1998). Weighting hidden Markov models for maximum discrimination. *Bioinformatics*, 14(9):772–782.

[Konings and Gutell, 1995] Konings, D. A. M. and Gutell, R. R. (1995). A comparison of thermodynamic foldings with comparatively derived structures of 16s and 16s-like rrnas. *RNA*, 1:559–574.

[Krogh et al., 1994a] Krogh, A., Brown, M., Mian, I. S., Sjölander, K., and Haussler, D. (1994a). Hidden Markov models in computational biology: Applications to protein modeling. *JMB*, 235:1501–1531.

[Krogh et al., 1994b] Krogh, A., Mian, I. S., and Haussler, D. (1994b). A Hidden Markov Model that finds genes in *E. coli* DNA. *NAR*, 22:4768–4778.

[Lari and Young, 1990] Lari, K. and Young, S. J. (1990). The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56.

[Lefebvre, 1995] Lefebvre, F. (1995). An optimized parsing algorithm well suited to rna folding. In *Intelligent Systems for Molecular Biology*.

[Lowe and Eddy, 1997] Lowe, T. and Eddy, S. (1997). trnascan-se: a program for improved detection of transfer rna genes in genomic sequence. *NAR*, 25:955–964.

[Lowe and Eddy, 1999] Lowe, T. and Eddy, S. (1999). A computational screen for methylation guide snornas in yeast. *Science*, 283:1168–1171.

[Maidak et al., 1997] Maidak, B., Olsen, G., Larsen, N., Overbeek, R., McCaughey, M., and Woese, C. (1997). The ribosomal data project (rdp). *NAR*, 25:443–453.

[Mamitsuka and Abe, 1994] Mamitsuka, H. and Abe, N. (1994). Predicting location and structure of beta-sheet regions using stochastic tree grammars. In *Intelligent Systems for Molecular Biology*, pages 276–284.

[Mathews et al., 1999] Mathews, D., Sabina, J., Zuker, M., and Turner, D. (1999). Expanded sequence dependence of thermodynamic parameters provides robust prediction of rna secondary structure. *JMB*, 288:911–940.

[Morrison and Ellis, 1997] Morrison, D. and Ellis, J. (1997). Effects of nucleotide sequence alignment of phylogeny estimation. *Mol. Biol. Evol.*, 14(4):428–441.

[Neal and Hinton, 1993] Neal, R. and Hinton, G. (1993). A new view of the em algorithm that justifies incremental and other variants. Preprint. Dept. of Computer Science, Univ. of Toronto. available from ftp://archive.cis.ohio-state.edu/pub/neuroprose/neal.em.ps.Z.

[Needleman and Wunsch, 1970] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similiarities in the amino acid sequence of two proteins. *JMB*, 48:443–453.

[Nix et al., 1999] Nix, J. C., Newhoff, A. R., and Wilson, C. (1999). Preliminary crystallographic characterization of an in vitro evolved biotin-binding rna pseudoknot. *ACTA CRYSTALLOGRAPHICA SECTION D-BIOLOGICAL CRYSTALLOGRAPHY*, 55:323–325.

[O'Brien et al., 1998] O'Brien, E., Notredame, C., and Higgins, D. (1998). Optimization of ribosomal rna profile alignments. *Bioinformatics*, 14(4):332–341.

[Olsen and Woese, 1993] Olsen, G. and Woese, C. (1993). Ribosmal rna: a key to phylogeny. *FASEB*, 7:113–123.

[Pace, 1997] Pace, N. (1997). A molecular view of microbial diversity and the biosphere. *Science*, 276:734–740.

[Park et al., 1998] Park, J., Karplus, K., Barrett, C., Hughey, R., Haussler, D., Hubbard, T., and Chothia, C. (1998). Sequence comparisons using multiple sequences detect 3 times as many remote homologs as pairwise methods. *JMB*, 284:1201–1210.

[Pereira and Schabes, 1992] Pereira, F. and Schabes, Y. (1992). Inside-outside reestimation from partially bracketed corpora. *30th Annual Meeting of the Association for Computational Linguistics*, pages 128–135.

[Rabiner, 1989] Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257–286.

[Rabiner and Juang, 1986] Rabiner, L. R. and Juang, B. H. (1986). An introduction to hidden Markov models. *IEEE ASSP Magazine*, 3(1):4–16.

[Raman and Overton, 1994] Raman, R. and Overton, C. (1994). Application of hidden Markov modeling to the characterization of transcription factor binding sites. In *Proc. 27th Hawaii Int. Conf. on System Sciences*, pages 275–283. IEEE Computer Society Press.

[Riley et al., 1998] Riley, D., Berger, R., Miner, D., and Krieger, J. (1998). Diverse and related 16s rrna-encoding dna sequences in prostate tissues of men with chronic prostatitis. *JOURNAL OF CLINICAL MICROBIOLOGY*, 36(6):1646–1652.

[Rissanen, 1989] Rissanen, J. (1989). *Stochastic Complexity in Statistical Inquiry.* World Scientific.

[Rivas and Eddy, 1999] Rivas, E. and Eddy, S. R. (1999). A dynamic programming algorithm for rna structure prediction including pseudoknots. *JMB*, 285(5):2053–2068.

[Sakakibara et al., 1994a] Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjölander, K., Underwood, R. C., and Haussler, D. (1994a). Stochastic context-free grammars for tRNA modeling. *NAR*, 22:5112–5120.

[Sakakibara et al., 1994b] Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjölander, K., Underwood, R. C., and Haussler, D. (1994b). Recent methods for RNA modeling using stochastic context-free grammars. In *Proc. Asilomar Conf. on Combinatorial Pattern Matching*, New York, NY. Springer-Verlag.

[Sańchez et al., 1996] Sańchez, J., Benedi;, J., and Cassacuberta, F. (1996). Comparison between the inside outside algorithm and the viterbi algorithm for stochastic context free grammars. In Perner, Wang, and Rosenfeld, editors, *Advance in Structural and Syntactic Pattern Recognition*. Springer.

[Searls, 1992] Searls, D. B. (1992). The linguistics of DNA. *American Scientist*, 80:579–591.

[Sjölander et al., 1996] Sjölander, K., Karplus, K., Brown, M., Hughey, R., Krogh, A., Mian, I., and Haussler, D. (1996). Dirichlet mixtures. *CABIOS*, 12(4):327–345.

[Sonnhammer et al., 1997] Sonnhammer, E., Eddy, S., and Durbin, R. (1997). Pfam: A comprehensive database of protein domain familes based on seed alignments. *Proteins Structure Function and Genetics*, 28:405–420.

[Stolcke, 1995] Stolcke, A. (1995). An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Lingustics*, 21.

[Strassen, 1969] Strassen, V. (1969). Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356.

[Tanaka et al., 1993] Tanaka, H., Ishikawa, M., Asai, K., and Konagaya, A. (1993). Hidden Markov models and iterative aligners. In *ISMB-93*, pages 395–401, Menlo Park. AAAI Press.

[ten Dam et al., 1992] ten Dam, E., Pleij, K., and Draper, D. (1992). Structural and functional aspects of rna pseudoknots. *Biochemistry*, 31:11665–11676.

[Thompson et al., 1994] Thompson, J. D., Higgins, D. G., and Gibson, T. J. (1994). CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties, and weight matrix choice. *NAR*, 22(22):4673–4680.

[Tomita, 1991] Tomita, M. (1991). *Generalized LR Parsing*. Kluwer.

[Underwood, 1994] Underwood, R. C. (1994). Stochastic context-free grammars for modeling three spliceosomal small nuclear ribonucleic acids. Master's thesis, University of California, Santa Cruz.

[Valiant, 1975] Valiant, L. (1975). General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10:308–315.

[Vapnik, 1998] Vapnik, V. (1998). *Statistical Learning Theory*. Wiley.

[Waterman, 1989] Waterman, M. S. (1989). Dynamic programming alignment of two sequences. In Waterman, M. S., editor, *Mathematical Methods for DNA Sequences*, chapter 3. CRC Press.

[Watson and Crick, 1953] Watson, J. and Crick, H. (1953). Molecular structure of nucleic acids: a structure for deoxyribose nucleic acid. *Nature*, 171:737–738.

[Watson et al., 1987] Watson, J., Hopkins, N., Roberts, J., Steitz, J., and Weiner, A. (1987). *Molecular Biology of the Gene*. Benjamin/Cummings.

[Wilson et al., 1998] Wilson, C., Nix, J., and Szostak, J. (1998). Functional requirements for specific ligand recognition by a biotin-binding rna pseudoknot. *Biochemistry*, 37(41):14410–14419.

[Wilson and Szostak, 1995] Wilson, C. and Szostak, J. (1995). In vitro evolution of a self-alkylating ribozyme. *Nature*, 374:777–782.

[Winkler, 1972] Winkler, R. (1972). *An Introduction to Bayesian Inference and Decision*. Holt.

[Woese, 1987] Woese, C. (1987). Bacterial evolution. *Microbiological Reviews*, 51(2):221–271.

[Woese and Fox, 1977] Woese, C. and Fox, G. (1977). Phylogenetic structure of the prokaryotic domain. *PNAS*, 74:5088–5090.

[Woese and Gutell, 1989] Woese, C. and Gutell, R. (1989). Evidence for several higher order structural elements in ribosomal rna. *PNAS*, 86:3119–3122.

[Woese et al., 1990] Woese, C., Kandler, O., and Wheelis, M. (1990). Towards a natural system of organisms. *PNAS*, 87:4576–4579.

[Woese et al., 1983] Woese, C. R., Gutell, R. R., Gupta, R., and Noller, H. F. (1983). Detailed analysis of the higher-order structure of 16S-like ribosomal ribonucleic acids. *Microbiology Reviews*, 47(4):621–669.

[Zardoya et al., 1995] Zardoya, R., Costas, E., Lopex-Rodas, V., Garrido-Pertierra, A., and Bautista, J. (1995). Revised dinoflagellate phylogeny inferred from molecular analysis of large subunit ribosomal rna gene sequences. *Journal of Molecular Evolution*, 41:637–645.

[Zuker, 1989] Zuker, M. (1989). On finding all suboptimal foldings of an rna molecule. *Science*, 244.

[Zuker et al., 1999] Zuker, M., Mathews, D., and Turner, D. (1999). Algorithms and thermodynamics for rna secondary structure prediction. In Barciszewski, J. and Clark, B., editors, *RNA Biochemistry and Biotechnology*. Kluwer.

[Zuker and Sankoff, 1984] Zuker, M. and Sankoff, D. (1984). RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621.

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