RAPPORT DE RECHERCHE

L R I

GenRGens VERSION 2.0 USER MANUAL

PONTY Y / DENISE A

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

04/2006

Rapport de Recherche N° 1447

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

GenRGenS v2.0 User Manual

Yann Ponty Alain Denise[1]

LRI, UMR CNRS 8623
Université Paris-Sud XI

Yann.Ponty@lri.fr Alain.Denise@lri.fr

Research Report n°1447 - April 2006

Abstract:
GenRGenS is a software dedicated to the random generation of genomic sequences and structures. It allows to handle several useful models in sequence analysis, as Markov chains, Hidden Markov models, weighted context-free grammars, regular expressions and Prosite expressions. In particular, GenRGenS is the only program that handles weighted context-free grammars, allowing the user to model and generate structured objects (as RNA secondary structures) of any given desired size. GenRGenS also allows the user to combine several of these different models.
The present user manual describes the different classes of models for random sequences supported by GenRGenS, and illustrates the implementation of such models with several examples inspired by real biological models. It also describes the different ways to invoke the software under various operating systems.

Availability:
Source and executable files of GenRGenS (in Java), as well as the complete user's manual, are freely available at http://www.lri.fr/bio/GenRGenS.

Contact: dev.GenRGenS@lri.fr

[1]This work was partially supported by the French IMPG Bioinformatics program, the CNRS Specific Action "Modélisation et algorithmique des structures secondaires d'ARN" and the French education ministry founded action ACI IMPBio.

Contents

Chapter 1

Introduction

Random sequences can be used to extract relevant information from biological sequences. The random sequences represent the "background noise" from which it is possible to differentiate the real biological information. Random sequences are widely used to detect over-represented and under-represented motifs, or to determine whether the scores of pairwise alignments are relevant. Analytic approaches exist for solving these kinds of problems (see *e.g.* [?].) although for the most complex cases, an experimental approach (*i.e.* the computer generation of random sequences) is still necessary.

Some programs are already currently available for generating random sequences. For example, the GCG package contains a few generation tools, such as *HmmerEmit* that generates sequences according to HMM profiles, and *Corrupt* that adds random mutations to a given sequence [?]. Seq-Gen randomly simulates the evolution of nucleotide sequences along a phylogeny [?]. The Expasy server has *RandSeq*, which generates random amino acid sequences according to a Bernoulli process [?]. Shufflet is a program that generates random shuffled sequences [?]. However, until now, there has been no software package that can integrate several statistical and syntaxical models of random sequences and combine them. This is the purpose of GenRGenS.

The random sequence models currently handled by GenRGenS are the following:

- `MARKOV` : Markovian random generation. Puts probabilistic constraints on the occurences of k-mers among the generated sequences. A markovian model can be automatically built from real biological data by a tool bundled with GenRGenS
- `GRAMMAR` : Random generation based on context-free grammars. This syntaxic formalism allows to take into account both sequential and structural constraints. Most long-range interactions and correlations can be captured by this formalism.
- `MASTER` : Random generation of hierarchical sequences. Combines different levels of abstraction. Sequences of symbols are generated using a `master` description file and then some of these symbols are rewrited into sequences generated with respect to `auxiliary` description files and distributions over sequences lengths.
- `RATEXP` : Prosite patterns and rational expressions. Generates sequences uniformly at random from a rational expression or a prosite pattern. Long ago, searches in language theory stated that these formalisms' expressivity are included in that of the context-free grammars'[1]. However, more efficient generation algorithms are available for this subclass. Moreover, support for Prosite patterns allows a *copy/paste* approach for random generation that some may find convenient.

[1]i.e. If a property is captured by a Prosite-based model, it is always possible to build a context-free grammar-based model capturing the same property.

Chapter 2

Generalities and file formats

2.1 How to run GenRGenS

This section is dedicated to the installation and usage of GenRGenS v2.0.
GenRGenS performs random generation from a given sequences model. Models are passed to the main generation engine through *description* files. These are text files meeting certain syntax criteria. Until further improvements GenRGenS supports four classes of models. Each of these models' description file has a specific syntax, although they all share certain properties.

2.1.1 Downloading and installing GenRGenS

Foreword: GenRGenS' current implementation uses Java, so it will be **necessary** to download and install a version of Java's runtime environment (or virtual machine) superior to 1.1.2, freely downloadable from `http://java.sun.com`.

GenRGenS latest versions (binaries+sources) can be found at the following URL:

`http://www.lri.fr/bio/GenRGenS`

First download the bundle file found in the *download* section of the site. If needed, uncompress the archive into an appropriate directory, using free or shareware tools like `7zip` (.ZIP files), Java bundled tool `jar` (.JAR files) or GNU `tar` (.TAR files) into a root directory of your choice. We'll assume for the next sections that the chosen directory is *`GenRGenSDir`*.

2.1.2 Command-line version

After decompression of the archive, move to *`GenRGenSDir`* and open a shell to invoke the Java virtual machine through the following command:

```
java -cp .  GenRGenS.GenRGenS [options] [-nb k] -size n DescriptionFile
```

where:

- k is the number of sequences to be generated by GenRGenS. Defaults to 1.
- n is an indicative length for the generated sequences. Depending on the class of models, it can either be the exact length, an upper bound or just ignored depending on the type of generation. **Required**.
- *`DescriptionFile`* is the path to a description file describing the random sequence model. **Required**.

Specific options may be available for some classes of models and will be detailed further.

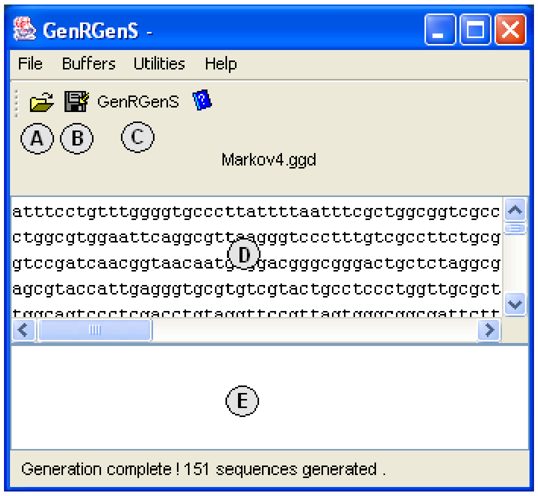
2.1.3 Graphical User Interface

There are two ways to run GenRGenS interactive version, one is achieved through command line and the other uses file associations supported by certain platforms:

- **From Shell:** Move to *GenRGenSDir* and, at the prompt, enter the following command:

```
      java -cp .  GenRGenS.GenRGenS
or    java -jar GenRGenSvX.X.JAR
```

- **From a graphical environment:** Some operating systems maintain associations between files and eligible executables, based on file name suffixes or file header analysis. Under these platforms, a simple double-click on the JAR archive will execute the main application.



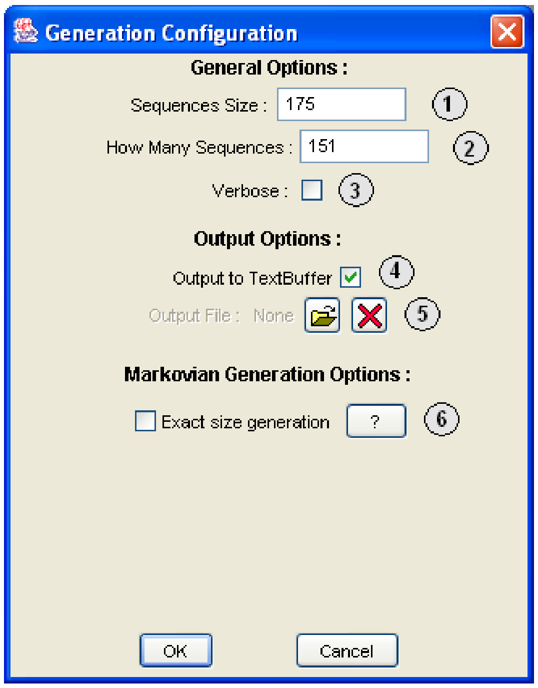
```
atttcctgtttggggtgcccttattttaatttcgctggcggtcgcc
ctggcgtggaattcaggcgttaagggtccctttgtcgccttctgcg
gtccgatcaacggtaacaatg[illegible]gacgggcgggactgctctaggcg
agcgtaccattgagggtgcgtgtcgtactgcctccctggttgcgct
tggcagtccctcgacctgtaggttccgttagtgggcggcgattctt
```

Generation complete ! 151 sequences generated .

Figure 2.1: Screenshot of the main GUI

Once GenRGenS UI is run, a typical random generation scenario would be:

- Open a description file, using `'File/Open Description File...'` or `'File/Reopen'` menu items or clicking on button **A**
- Define some required/optional parameters, such as the sequence length or the number of sequences, inside the `Generation Configuration` window spawned from `'Utilities/GenRGenS'` menu item or by clicking on button **C**.
- Validate generation. Random sequences are generated and displayed on text buffer **D**. Potential errors or warnings are shown on text buffer **E**.
- Save generated sequences to disk, using `'Buffers/Save Output Buffer Content...'` or by cliquing on button **B**.


Generation Configuration

General Options :

Sequences Size : 175 ①

How Many Sequences : 151 ②

Verbose : ☐ ③

Output Options :

Output to TextBuffer ✓ ④

Output File : None ⑤

Markovian Generation Options :

☐ Exact size generation ? ⑥

Figure 2.2: Defining the generation parameters

The `Generation Configuration` window defines of a few additional parameters:

- Box **1**: The size of the emitted sequences or an upper-bound depending on the type of generation.
- Box **2**: The number of generated sequences.
- Checkbox **3**: Whether or not to display informations about the generation during the process.
- Checkbox **4**: Toggles displaying of generated sequences on and off. Useful when large numbers of sequences are required.
- Messagebox **5**: Selects the output file.
- Panel **6**: Display a number of generator-specific option. Here, for a markovian generator, it is possible to provide the *size* parameter (Box **1**) as an upper bound for the sequence size. The sequence is then generated letter by letter, until a dead-end is encountered, i.e. there is no sequel for this sequence in this model, or the size provided is reached.

2.2 Description files

Description files describe random models to the generation engine of GenRGenS. They are composed of clauses, each defining a parameter in the random model. The syntax of description files will be detailed below, along with the most common clauses.

2.2.1 Main structure

GenRGenS description files are sequences of clauses. All clauses are based on the pattern `Param_Name = Param_Value`, where `Param_Name` is the name of the parameter being defined and `Param_Value` its value. Parameters available are specific to a given random model, although some parameters are shared by most if not all description files types. Clauses must be ordered in a generation-specific way, otherwise they will be rejected by GenRGenS' main engine. A clause can be optional: if omitted, its associated parameter will default to a value detailed in the random model's description part of this document.

2.2.2 Common clauses

2.2.2.1 The `TYPE` clause

`TYPE = {MARKOV,GRAMMAR,RATEXP,MASTER}`

The `TYPE` clause is the first clause of any description file. It defines the type of random model to be used for generation. Currently supported values for `FileType` are listed below:

TYPE	Random model description
MARKOV	Markovian random generation.
GRAMMAR	Random generation based on context-free grammars.
MASTER	Random generation of hierarchical sequences.
RATEXP	Prosite patterns and rational expressions.

2.2.2.2 The `ALIAS` clause

`ALIASES =` s_1=id_1 s_2=id_2 ...

s_i is a symbol used for random generation
id_2 is a new representation for this symbol

Another common clause is the `ALIAS` clause. It causes GenRGenS to substitute the right hand sides of the equalities to the left hand sides after the generation is performed. This clause simplifies the writing of large random models while keeping the output explicit, as one can write the whole model using letters and substituting more explicit symbols to letters afterward. See chapter 3 for advanced use of this clause.

2.2.3 Simple example

```
1  TYPE = MARKOV

2  ORDER = 0

3  FREQUENCIES =
        a   33   c   20
        g   15   t   32

4  ALIASES =
        a = A   c = C
        g = G   t = U
```

Figure 2.3: A simple Markovian description file

For instance, here is the toy example of a description file describing a simple Markovian model:

Clause **1** defines the class of random model to be used for random generation. Here, a Markovian model is choosen, thus raising a need for the definition of various parameters whose roles a explained below.

Clause **2** defines the order of the Markovian model. A 0 value stands for a Bernoulli model, i.e. the probability of emission of a letter doesn't depend on letters priorly emitted. Further details and definitions can be found on chapter 3.

Between clause **2** and clause **3**, an optional clause `PHASE = int` is omitted. Default value 1 will then be used for the number of phases, thus defining an homogenous Markovian model.

Clause **3** defines the emission probabilities for the various symbols. Instead of asking the user to provide the probabilities for the different k-mers, we preferred to compute the probabilities from given numbers of occurrences. This approach is well fit for the use of a Markovian profile built from a real sequence. Here, we are using the DNA bases `A`,`C`,`G` and `T`. Here, Adenosine(A) will be emitted with a $\frac{33}{33+20+15+32} = 0.33$ probability.

Clause **4** performs-post generation rewriting of the sequence. Here, it allows generation of RNA sequences from a DNA-dedicated Markovian model [1].

[1] Which may seem a little lazy as the model is simple, but the size of a Markovian model grows exponentially with respect to its order, and rewriting manually a Markovian model of order 6 may bore the most wilful scientist.

Chapter 3

The MARKOV package : Markovian models

Markovian models are the simplest, easiest to use statistical models available for genomic sequences. Statistical properties associated with a Markovian model make it become a valuable tool to the one who wants to take into account the occurrences of k-mers in a sequence. Their most commonly used version, the so-called *classical* Markovian models, can be automatically built from a set of real genomic sequences. Hidden Markovian models are now supported by GenRGenS at the cost of some preprocessing, as shown in section 3.1.3. Apart from genomics, such models appear in various scientific fields including-but-not-limited-to speech recognition, population processes, queuing theory and search engines[1].

3.1 Some theoretical aspects

3.1.1 Main definition

Formally, a classical Markovian model applied to a set of random variables $V_1, \ldots, V_n$ causes the probabilities associated with the potential values for V_n to depend on the values of $V_i, \ldots, V_{n-1}$. We will focus on homogenous Markovian models, where the probabilities for the different values for V_n are conditioned by the values already chosen for a small subset $[V_{i-k-1}, V_{i-1}]$ of variables *from the past*, also called *context* of V_i. The parameter k is called the *order* of the Markovian model. Moreover, the probabilities of the values for V_i in an homogenous Markovian model cannot in any way be conditioned by the index i of the variable.

Applied to genomic sequences, the random variable V_i stands for the i^{th} base in the sequence. The Markovian model constrains the occurrence probability for a base α in a given context composed of the k previously assigned letters, therefore weakly[2]! constraining the proportions of each $k+1$-mers.

3.1.2 What about heterogeneity ?

GenRGenS handles a small subset of the heterogenous Markovian class of models.
Formally, an heterogenous model allows the probabilities associated with a variable V_i to depend on any of the variables prior to V_i. Our subset of the heterogenous Markovian models will use an integer parameter called *phases* to compute the probabilities for V_i. The phase of a variable V_i is

[1]Google computes its PageRank, which is a score for the relevance of a page, by modelling the behavior of a randomly clicking net-surfer using a Markovian model ...

[2]When total control over the occurrences of $k+1$-length words(and shorter...) is required, one should consider using *shuffling* algorithms

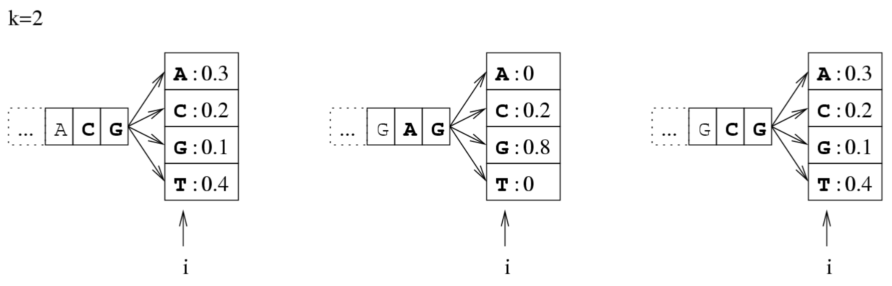


Figure 3.1: The probability of the event : "the i-th base is an a" depends on its k predecessors.

simply given by the formula $i \mod phases$. In our subset of the heterogenous Markovian models, the probabilities for V_i depend on both context and phase of the variable. Such an addition is useful to model phenomenons in which variables are gathered in packs of *phases* consecutive variables, and their values not only depend of their contexts, but also of their relative position inside the pack. Such are sequences of protein-coding DNA, where the position of a base inside of a base triplet is well-known to be of interest.

It is also possible to simulate such phase alternation by introducing dummy letters encoding the phase in which they will be produced. For instance, an order 0 model having 3 phases (for coding DNA...) would result in a sequence model over the alphabet $\{a_0,c_0,g_0,t_0,a_1,c_1,g_1,t_1,a_2,c_2,g_2,t_2\}$ having non-null transition probabilities only for $\cdots x_0 \rightarrow y_1$, $\cdots x_1 \rightarrow y_2$ and $\cdots x_2 \rightarrow y_0$. Therefore, the expressivity of our heterogenous models do not exceed that of the homogenous ones, but it is much more convenient to write these models using an heterogenous syntax.

3.1.3 Hidden Markovian Models (HMMs)

Hidden Markovian models address the hierarchical decomposability of most sequences.

An hidden Markovian model is a combination of a top-level Markovian model and a set of bottom-level Markovian models, called hidden states. The generation process associated with an HMM initiates the sequences using a random hidden state. At each step of the generation, the algorithm may switch to another hidden using probabilities from the top-level model, and then emits a symbol using probabilities related to the current urn.

Once again, this class of models' expressivity seems to exceed that of the classical Markovian models. However, in our context, it is possible to emulate an hidden model with a classical one just by duplicating the alphabet so that the emitted character also contains the state which it belongs to.

3.2 Implementing a Markovian model

This section describes the syntax and semantics of Markovian description files, as shown in figure 3.2.

3.2.1 Main structure

Clauses nested inside square brackets are optional. The given order for the clauses is **mandatory**.

3.2.2 Markovian generation specific clauses

Markovian description files allows definition of the Markovian model parameters.

```
TYPE = MARKOV
ORDER = ...
[PHASE = ...]
[SYMBOLS = ...]
[START = ...]
[FREQUENCIES | HMMFREQUENCIES] = ...
[ALIASES = ...]
```

Figure 3.2: Main structure of a Markovian description file

3.2.2.1 The `ORDER` clause

$$\texttt{ORDER = } k$$

$$k \in \mathbb{N}$$

Required
Sets the *order* of the underlying Markovian model to a positive integer value k. The order of a Markovian model is the number of previously emitted symbols taken into account for the emission probabilities of the next symbol. See section 3.1.1 for more details about this parameter.

3.2.2.2 The `PHASE` clause

$$\texttt{PHASE = } \mathit{phases}$$

$$\mathit{phases} \in \mathbb{N}^*$$

Optional, defaults to `PHASE = 0`
Sets the *number of phases* parameter of GenRGenS' heterogenous Markovian models subclass to a positive non-null integer value *phases*. See section 3.1.2 for more details about this parameter.

3.2.2.3 The `SYMBOLS` clause

$$\texttt{SYMBOLS = \{WORDS,LETTERS\}}$$

Optional, defaults to `SYMBOLS = WORDS`
Chooses the type of symbols to be used for random generation.
When `WORDS` is selected, each pair of symbols must be separated by at least a blank character (space, tabulation or newline). A Markovian description file written using `WORDS` will then be easier to read, as explicit names for symbols can be used, but may take a little longer to write, as illustrated by the toy example of the `FREQUENCIES` clause's content for a simple Markovian description file modelling the ORF/Intergenic alternation at a base-triplet level :

```
Start ORF 100   ORF Stop 35   Stop Intergenic 93   Intergenic Intergenic 85
                 ORF ORF 65       Stop ORF 7          Intergenic Start 15
```

Figure 3.3: A simple Markovian model for the Intergenic/ORF alternation

A `LETTER` value for the `SYMBOLS` parameter will force GenRGenS to see a word as a sequence of symbols. For instance, the definition of a 15/35/23/27 occurrences proportions for the symbols `A`/`C`/`G`/`T` in a context `ACC` will be expressed by the following statement inside of the `FREQUENCIES` clause :

```
ACCA 15 ACCC 35 ACCG 23 ACCT 27
```

3.2.2.4 The `START` clause

$$\texttt{START = } s_1\ n_1\ s_2\ n_2\ \ldots \qquad n_i \in \mathbb{N}$$

Optional, defaults to the distribution of `k`-mers, `k` being the value *order*(see below).
Defines the frequencies associated with various eligible prefixes for the generated sequence.
Each s_i is either a sequence of symbols separated by white spaces or a word, depending on the value of the `SYMBOLS` parameter. Every s_i must be composed of the same amount m of symbols, $m \geq order$.
If omitted, the beginning of the sequence is chosen according to the distribution of `k`-mers implied by the content of the `FREQUENCIES` clause and computed as follows :

$$\forall \omega \in V^k,\ p_\omega = \sum_{c \in V} p_{\omega.c}$$

where V is the set of symbols used inside of the sequence, V^k is the set of words of size k, and $p_{c,\omega}$ is the probability of emission of c in a context ω as defined by the `FREQUENCIES` clause. Note that no specific order is required for definition of a `START` clause.

3.2.2.5 The `FREQUENCIES` clause

This clause is used to define the probabilities of emission of the Markovian model.

$$\texttt{FREQUENCIES = } s_1\ n_1\ s_2\ n_2\ \ldots \qquad n_i \in \mathbb{N}$$

Required
Defines the probabilities of emission for the different symbols.
Each s_i is either a sequence of symbols separated by white spaces or a word, depending on the value of the `SYMBOLS` parameter. Each s_i is composed of $k+1$ symbols, k being the order. The first k symbols define the context and the last letter a candidate to emission. The relationship between the frequency definition $s_i\ n_i$ and the probability p_{c_i,ω_i} of emitting c_i in a context ω_i, $s_i = \omega_i.c_i$ is given by the following formula :

$$p_{c_i,\omega_i} = \frac{n_i}{\displaystyle\sum_{s_j=\omega_i.c} n_j}$$

The (simple) idea behind this formula is that the probability of emission of a base in a certain context equals to the context/base, concatenation's frequency divided by the sum of the frequencies sharing the same context . Choice has been made to write frequencies instead of direct probabilities because most of the Markovian models encountered in genomics are built from real data by counting the occurrences of all $k+1$-mers, thus allowing the direct injection of the counting process' result into the description file. As for the `START` clause, it is not necessary to provide values for the `FREQUENCIES` in a specific order.

3.2.2.6 The `HMMFREQUENCIES` clause

$$\begin{array}{c}\texttt{HMMFREQUENCIES = } s_1\ n_1\ s_2\ n_2\ \ldots\ ;\\ \alpha_1\ :\ s_1^1\ n_1^1\ s_2^1\ n_2^1\ \ldots\ ;\\ \alpha_2\ :\ s_1^2\ n_1^2\ s_2^2\ n_2^2\ \ldots\ ;\\ \ldots\ ;\\ s_i \in \{\alpha_j\}^*,\ n_i \in \mathbb{N},\ n_i^j \in \mathbb{N}\end{array}$$

Required
Defines the hidden Markovian model's probabilities of emission.
First, a master model s_1 n_1 s_2 n_2 ... is defined for the alternation of the hidden states. s_1, s_2, ... are sequences of hidden states α_i, separated or not by whitespaces depending on the previously defined value for the clause `SYMBOLS`. n_i are sequences of integers, representing the frequencies of the corresponding sequence. Their lengths equal the phase parameter of the model, as provided by the `PHASE` clause.
Then, a model definition is required for each hidden states α_i through the following syntax:

$$\alpha_i \ : \ \ s_1^i \ n_1^i \ s_2^i \ n_2^i \ \ldots \ ;$$

Each s_i^j is composed of symbols β_k, which are part of the emission vocabulary.
Once such a model is defined, a sequence is issued starting from a random hidden state h_0. At each step k of the generation, the process is allowed to move from the current hidden state h_k to another hidden state h_{k+1} (potentially $h_k = h_{k+1}$) and then emits a symbol according to the probabilities of the hidden state h_{k+1}. The process is iterated until the expected number of symbols are generated.
Remark 1: Each model definition must be ended by a semicolon **;** to avoid ambiguity.
Remark 2: The vocabularies $A = \{\alpha_i\}$ and $B = \{\beta_i\}$ respectively used for the master and the hidden states model definition must be disjoint.
Remark 3: The numbers of phases for heterogenous master and hidden states model **must be equal**.
Remark 4: Different orders for the master and states models are supported.

3.3 Examples

3.3.1 A Bernoulli model

3.3.1.1 Source

We illustrate the design of a Markovian model with a description file generated automatically from the 22-th human chromosome, using the tool `BuildMarkov` bundled with `GenRGenS` and described in section 3.4.2, through the following command:

```
java -cp .  GenRGenS.markov.BuildMarkov -o 0 -p 1 q22.fasta -d q22.ggd
```

This model is a Bernoulli one, as no memory is involved here, i.e. the probability of emission of a base doesn't depend on events from the past.

```
1  TYPE = MARKOV
2  ORDER = 0
3  PHASE = 1
4  SYMBOLS = LETTERS
5  FREQUENCIES =
     T 8800702   G 8083806   C 8090307   A 8846873
```

Figure 3.4: A Bernoulli model for the entire 22-th human chromosome(cleaned up from unknown `N` bases.)

3.3.1.2 Semantics

In **1**, we choose a Markovian random generation.

In **2**, we choose an amnesic model, i.e. there is no relation between two bases probabilities.

In **3**, we choose an homogenous model, the probabilities of emission won't depend on the index of the symbol among the sequence.

In **4**, we use letters as symbols for simplicity sake.

Clause **5** provides definition for the frequencies. Here, the `A` and `T` are slightly more likely to occur than `C` and `G`. Precisely, at each step an `A` symbol will be emitted with probability $\frac{8846873}{8800702+8083806+8090307+8846873} \approx 0,2616$.

3.3.2 A model for a mRNA having order 1

3.3.2.1 Source

This description file has been built automatically from a portion of the 17q11 part of the 17-th human chromosome by the tool `BuildMarkov` invoked through the following command:

```
java -cp .  GenRGenS.markov.BuildMarkov -o 1 -p 1 17-q11.fasta -d 17-q11.ggd
```

```
1  TYPE = MARKOV
2  ORDER = 1
3  PHASE = 1
4  SYMBOLS = WORDS
5  FREQUENCIES =
     a a 64    a c 98    a g 127   a t 52
     c a 126   c c 174   c g 58    c t 138
     g a 112   g c 127   g g 135   g t 66
     t a 39    t c 96    t g 121   t t 48
```

Figure 3.5: Order 1 Markov model for the sequence of the mRNA encoding the S-protein involved in serum spreading.

3.3.2.2 Semantics

In **1**, we choose a Markovian random generation.

In **2**, we choose an order 1 model, i.e. the probability of emission of a symbol only depends on the symbol immediately preceding in the sequence.

In **3**, we choose an homogenous model, the probabilities of emission won't depend on the index of the symbol among the sequence.

In **4**, we illustrate the use of `WORDS`. The symbols must be separated in the `FREQUENCIES` by at least one whitespace or carriage return. Space characters will be inserted between symbols in the output.

Clause **5** provides definition for the frequencies, subsequently for the transition/emission probabilities.

3.3.3 An heterogenous model

3.3.3.1 Source

The following description file describes a Markovian model for the first chromosome's 1q31.1 area of the human genome, built by `BuildMarkov` through :

```
java -cp .  GenRGenS.markov.BuildMarkov -o 2 -p 3 1q31.1.fasta -d 1q31.1.ggd
```

and edited afterward to include the `START` clause.

```
1  TYPE = MARKOV
2  ORDER = 2
3  PHASE = 3
4  SYMBOLS = LETTERS
5  START =
     atg 99
     gtg 1
6  FREQUENCIES =
    agg  19  18  22   cgg   3  10   6   act  42  27  28   agc  30  24  22
    aga  47  30  40   cct  22  33  35   cgc  11   9  11   tag  23  28  31
    cga   3   6   5   att  68  80  53   ggg  14  12  15   tac  17  26  36
    ctt  46  38  45   taa  58  49  48   gct  26  31  24   ggc  15   8  21
    acg  10   2   5   gga  25  22  17   acc  22  18  22   ccg   8  12  11
    aca  38  28  45   gtt  41  38  26   tgt  42  35  40   atg  32  42  44
    ccc  17  21  17   cca  20  39  29   atc  31  24  23   gcg   5   7   5
    ctg  24  27  41   ata  43  56  46   gcc  25  24   8   ctc  26  19  23
    gca  25  26  20   cta  29  32  24   gtg   9  30  27   tgg  32  29  28
    gtc  20  16  13   tct  26  42  38   tgc  32  16  27   gta  34  27  26
    tga  54  26  47   ttt  89  99  95   tcg   5   6   5   tcc  41  29  20
    tca  38  34  36   ttg  41  43  48   aat  71  48  50   ttc  34  40  51
    tta  55  62  61   cat  35  27  37   gat  37  33  24   aag  31  38  51
    aac  19  32  36   cag  27  40  36   aaa  74  96  94   cac  25  19  26
    caa  43  35  28   gag  19  22  21   agt  43  28  43   gac  14  23  14
    gaa  39  51  25   cgt   9   3   5   tat  59  58  62   ggt  17  26  16
```

Figure 3.6: A Markovian description file for the 1q31.1 area of the first human chromosome

3.3.3.2 Semantics

In **1**, we choose a Markovian random generation.

In **2**, we will consider the frequencies of base triplets, e.g. the probability of emission of a base is conditioned by the 2 last bases.

In **3**, we differentiate the behaviour of the Markovian process for each phase, in order to capture some properties of the coding DNA.

In **4**, we use letters as symbols for simplicity sake.

Clause **5** initiates each sequence with an `atg` base triplet with probability 0.99 or chooses `gtg` with probability 0.01.

Clause **6** provides definition for the frequencies. To explain the semantics of this clause, we will focus on some particular frequencies definitions :

`agg 19 18` **`22`** `aga 47 30` **`40`** `agt 43 28` **`43`** `agc 30 24` **`22`**

Writing `agg 19` means that base `g` has occurred 19 times in an `ag` context on phase 0. In other words, the motif `agg` has occurred 19 times on phase 1. We will illustrate the link with the emission probability of `g` in a context `ag` in the next section.

3.3.3.3 Random generation scenario for this example

- First a random start `atg` is chosen with probability 0.99.
- The context, here composed of the two most recently emitted bases, is now `tg`, and the phase[3] of the next base is 2. The emission probabilities for the bases of a `g` is $p^2_{g,ag} = \frac{22}{22+40+43+22} = \frac{22}{127}$. Similarly, probabilities of emissions for the other bases are $p^2_{a,ag} = \frac{40}{127}$, $p^2_{t,ag} = \frac{43}{127}$ and $p^2_{c,ag} = \frac{22}{127}$.
- After a call to a random number generator, a `g` base is chosen and emitted.
- The new context is then `gg`, and the new phase is 0. We then consider new probabilities for the bases `a,c,g` and `t`:

  `gga` **`25`** `22 17 ggc` **`15`** `8 21 ggg` **`14`** `12 15 ggt` **`17`** `26 16`

  The probabilities of emission for the different bases are then $p^0_{a,gg} = \frac{25}{71}$, $p^0_{c,gg} = \frac{15}{71}$, $p^0_{g,gg} = \frac{14}{71}$ and $p^0_{t,gg} = \frac{17}{71} \dots$

3.3.4 Basic Hidden Markov Model

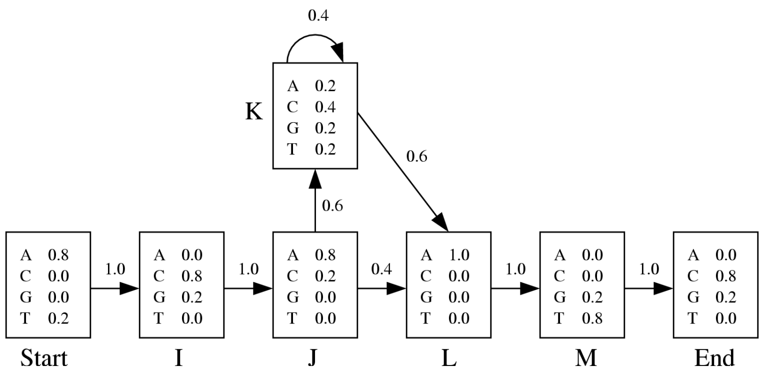


Figure 3.7: Basic HMM excerpted from the sequence analyst's bible[**?**]

Consider the basic output of a HMM profile building algorithm drawn in Figure 3.7.

3.3.4.1 Source

It can be translated into the following GenRGenS input file:

[3]We recall that the phase equals to $n \mod Phases$, where n is the number of bases previously generated.

```
1  TYPE = MARKOV
2  ORDER = 1
3  SYMBOLS = WORDS
4  HMMFREQUENCIES =
    St I 100  I J 100  J K 60  J L 40
    K K 60  K L 40  L M 100  M End 100 ;
      St:   A   80   C  00   G  00   T  20 ;
      I:    A   00   C  80   G  20   T  00 ;
      J:    A   80   C  20   G  00   T  00 ;
5     K:    A   20   C  40   G  20   T  20 ;
      L:    A  100   C  00   G  00   T  00 ;
      M:    A   00   C  00   G  20   T  80 ;
      End:  A   00   C  80   G  20   T  00 ;
```

Figure 3.8: A Markovian description file for the 1q31.1 area of the first human chromosome

3.3.4.2 Semantics

In **1**, we choose a Markovian random generation.

In **2**, an order of 1 is defined for the master model.

In **3**, we choose to manipulate words as symbols.

Part **4** is a definition for the frequencies of the master Markovian model. For instance, the probability of choosing `L` as the next hidden state while in state `J` is $p_{JL} = \frac{40}{100}$.
Remark: The word `START` is a keyword and cannot be used as an identifier.

Part **5** contains the different emission frequencies, converted from probabilities to integers.

3.4 Command-line options and additional tools

3.4.1 Markov-specific option: Dead-Ends tolerance

Inside some markovian models, it is theoreticaly possible that some states may be *dead-ends*, the frequencies of all candidate symbols summing to 0 in this context. It is then impossible to emit an extra symbol. That is why `GenRGenS` always checks weither each reachable state can be exited. The default behaviour of `GenRGenS` is to refuse to generate sequences in such a case, as the generated sequences are supposed to be of the size provided by the user, and it is senseless to keep on generating letters when a *dead-end* is encountered. However, this phenomenon may be intentional, in order for instance to end the generation with a specific word (perhaps a STOP codon...) if total control over the size can be neglected.

Markovian specific command line option usage:

`java -cp . GenRGenS.GenRGenS -size` *n* `-nb` *k* **-m [T|F]** *MarkovianGGDFile*

- **-m [T|F]**: Toggles rejection of models with dead-ends *on* (`T`) and *off* (`F`). Notice that disabling the rejection may result in shorter sequences than that specified by mean of the *size* parameter. **Defaults to -m T.**

3.4.2 BuildMarkov: Automatic construction of `MARKOV` description files

3.4.2.1 Tool description

This small software is dedicated to the automatic construction of `MARKOV` description files from a sequence. It scans the sequence(s) provided through the command line and evaluates the probabilities of transitions from and to each markov states.

3.4.2.2 Usage

After decompression of the `GenRGenS` archive, move to ***`GenRGenSDir`*** and open a shell to invoke the Java virtual machine through the following command:

```
java -cp .  GenRGenS.markov.BuildMarkov [options] InputFiles
```

where:

- *InputFiles* is a set of paths aiming at sequences that will be used for Markov model construction. **Required(At Least one)**.

The following parameters/options can also be specified :

- `-d` *OutputFile*: Outputs the description file to the file `OutputFile`.
- `-p` n: Defines the number of phases in the markov model. n must be > 0. $n = 1$ means that the model will be homogenous. **Defaults to n=1.**
- `-o` k: Defines the order of the markov model. k must be ≥ 0. $k = 0$ means that the model will be a bernoulli model. **Defaults to n=0.**
- `-v` 1: Verbose mode, show the progress of the construction(useful for large files).

3.4.2.3 Input files

The input files can be *flat files*, that are text files containing a *raw* sequence, or `FASTA` formatted files, that are an alternation of *title* lines and sequences, and usually look like the figure below.

Title line →	> seq1 This is the description of my first sequence.
Seq Def →	AGTACGTAGTAGCTGCTGCTACGTGCGCTAGCTAGTACGTCA
	CGACGTAGATGCTAGCTGACTCGATGC
	> seq2 This is a description of my second sequence.
	CGATCGATCGTACGTCGACTGATCGTAGCTACGTCGTACGTAG
	CATCGTCAGTTACTGCATGCTCG

Figure 3.9: A typical FASTA formatted file, containing several sequences.

Remark1: Inside a sequence definition, both for `FASTA` and flat files, `BuildMarkov` ignores the spaces, tabulations and carriage returns so that the sequence can be indented in a user-friendly way.
Remark2: When several sequences are found inside a `FASTA` file, they are considered as different sequences and processed to build the Markov model. If the `FASTA` file contains a sequence split into *contigs*, thus needing to be concatenated, then a preprocessing of the file is required to remove the title lines before it can be passed as an argument to *BuildMarkov*.

Chapter 4

The GRAMMAR package: Context-Free Grammars(CFG)

4.1 Minimal amount of language theory

For many years, the context free grammars have been used in the theoretical languages field They seem to be the best compromise between computability and expressivity. Applied to genomics, they can model long range interactions, such as base pairings inside an RNA, as shown by D.B. Searls[**?**]. We provide in this package generation algorithms for two classes of models based on Context Free Grammars: the uniform models and the weighted models. In the former, each sequence of a given size has the same probability of being drawn. In the latter, each sequence's probability is proportional to its composition. Like stochastic grammars, a weight set can be guessed to achieve specific frequencies. Unlike stochastic grammars, one can control precisely the size of the output sequences.

The term context-free for our grammars equals to a restriction over the form of the rules. This restriction is based on language theoretical properties. A sample of context-free grammar, and the way a sequence is produced from the axiom can be seen in figures 4.1 and 4.2.

1	S	$\to$	$aSbS$
2	S	$\to$	ϵ

Figure 4.1: A context free grammar for the well-balanced words

$$\begin{aligned}
\underline{S} &\underset{1}{\to} \mathbf{a\underline{S}bS} \underset{1}{\to} a\mathbf{a\underline{S}bS}bS \underset{2}{\to} aab\underline{S}bS \underset{1}{\to} aab\mathbf{a\underline{S}bS}bS \underset{2}{\to} aabab\underline{S}bS \\
&\underset{2}{\to} aababb\underline{S} \underset{1}{\to} aababb\mathbf{a\underline{S}bS} \underset{1}{\to} aababba\mathbf{a\underline{S}bS}bS \\
&\underset{2}{\to} aababbaab\underline{S}bS \underset{2}{\to} aababbaabb\underline{S} \underset{2}{\to} aababbaabb
\end{aligned}$$

Figure 4.2: Derivation of the word *aababbaabb* from the axiom S. **Bold** are the freshly produced letters. *Underlined* are the letters that will be rewritten at next step.

4.1.1 Grammars and discrete models

In the computer science and combinatorics fields, words issued from grammars are used as codes for certain objects, like computer programs, combinatorial objects, . . . For instance, the grammar whose rules are given in figure 4.1 naturally encodes well balanced parenthesis words, if a stands for a opening parenthesis and b stands for a closing one . A one to one correspondence has also been shown between the words generated by this grammar and some classes of trees. As trees are also used as approximations of biological structures, it is possible to consider the use of context-free grammars as discrete models for genomic structures.

4.1.1.1 Toward models for structures: Modelling long range interactions

We now discuss the utility of context-free grammars as discrete models for genomic structures. Most of the commonly-used models do only take into account a smallest subset of their neighborhood. For instance, in a Markov model, the probability of emission of a letter depends only on a fixed number of previously emitted letters. Although these models can prove sufficient when structural data can be neglected, they fail to capture the long-range interaction in structured macromolecules such that RNAs or Proteins. Back to our CFGs, figure 4.2 illustrates the fact that letters produced at the same step can at the end of the generation be separated by further derivations. As we claim that dependencies inside a CFG-based model rely on some terminal symbols proximity inside of the rules, CFGs can capture both sequential and structural properties, as illustrated by figure 4.3.

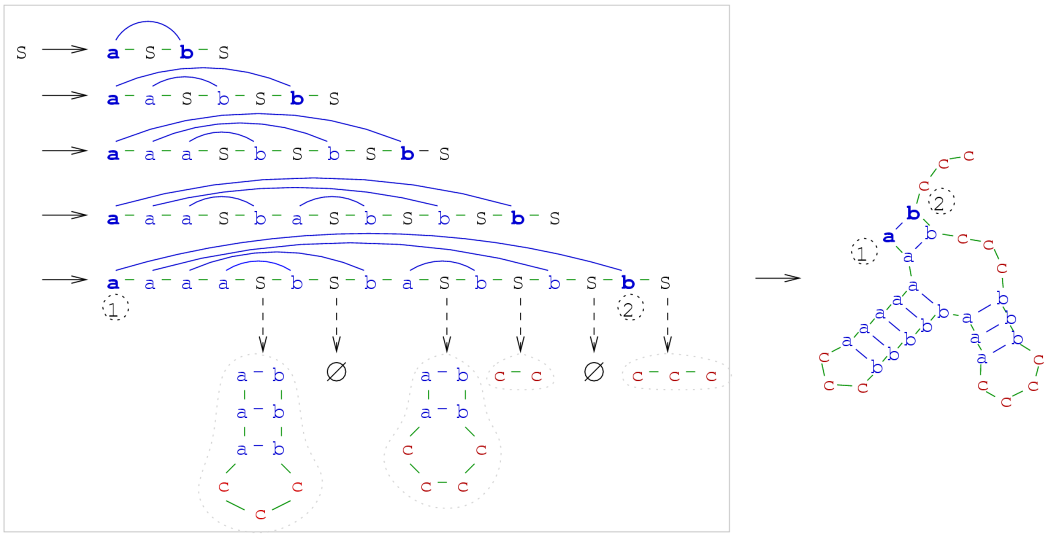


Figure 4.3: Long range interaction between bases 1 and 2 can be captured by CFGs, as they are generated at the same stage, and then separated by further non-terminal expansions. The CFG used here is an enrichment of the one shown in Figure 4.2

4.2 Uniform random generation of words of context-free languages

The uniform random generation algorithm, adapted from a more general one[**?**], is briefly described here. At first, we motivate the need for a precomputation stage, by pointing out the fact that a stochastic approach is not sufficient to achieve uniform generation. Moreover, controlling the size of the output using stochastic grammar is already a challenge in itself.

4.2.1 Stochastic approach does not guarantee uniformity

To illustrate this point, we introduce the historical grammar for RNA structures, whose rules are described in figure 4.4.

1	S	$\rightarrow$	$aTbS$
2	S	$\rightarrow$	cS
3	S	$\rightarrow$	ϵ
4	T	$\rightarrow$	$aTbS$
5	T	$\rightarrow$	cS

Figure 4.4: A CFG grammar for the RNA Secondary Structures

Now, suppose one wants to generate a word of size 6 uniformly at random from this grammar. A natural idea would be to rewrite from the axiom S until a word of size 6 is obtained. Therefore, you have to check wether each derivation yields a word of the desired length, as, for instance, the $S \rightarrow \epsilon$ rule produces only the empty word and needs not to be investigated.

Once you have built a set of derivations suitable for a given length, you have to carefully associate probabilities with each derivation. Indeed, choosing any of the eligible k rules with probability $1/k$ may end in a biased generation, as the numbers of sequences accessible from any rules are not equal. This fact is illustrated by the example from figure 4.4, where the two eligible rules for sequences of size 6 from the axiom S are $S \rightarrow aTbS(1)$ and $S \rightarrow cS(2)$.

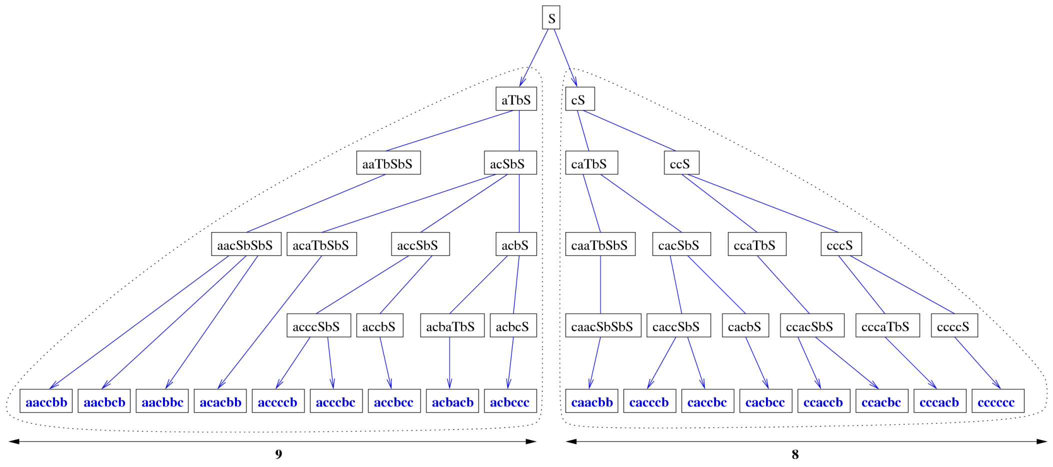


Figure 4.5: Deriving words of size 6 from S

As illustrated by figure4.5, choosing rules (1) or (2) with equal probabilities is equivalent to grant the set of words produced by rules (1) and (2) with total probabilities 1/2 and 1/2. As these sets do not have equal cardinalities, uniform random generation cannot be achieved.

4.2.2 A precomputation stage is needed

That is why the grammar is first analyzed during a preprocessing stage, that precomputes the probabilities of choosing rules at any stage of the generation. These probabilities are proportional to the number of sequences accessible after the choice of each applicable rule for a given non-terminal, normalized by the total number of sequence for this non-terminal.

Once these computations are made, it is possible to perform uniform random generation by choosing a rewriting using the precomputed probabilities, as shown in figure 4.6.

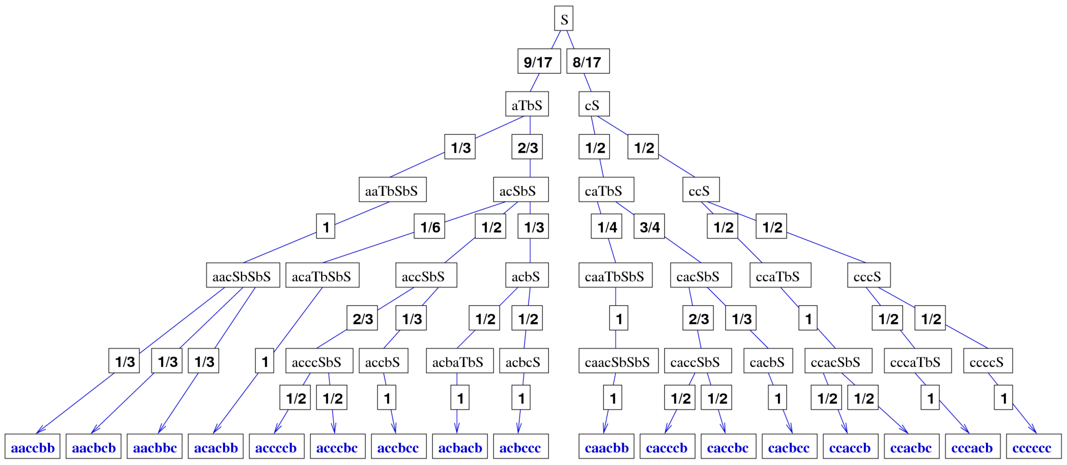


Figure 4.6: The complete tree of probabilities for the words of size 6.

4.3 Weighted languages for controlled non-uniform random generation

Although uniform models can prove useful in the analysis of algorithms, or to shed the light on some overrepresented phenomena, they are not always sufficient to model biological sequences or structures. For instance, it can be shown using combinatorial tools that the classical model for RNA secondary structures from figure 4.4 produces on the average *one base-pair-long helices* (see [?] or [?]), which is not realistic. Furthermore, substituting the rule 1 with a rule $S \to a^kTb^kS$ in order to constrain the minimal number of base pairs inside an helix only raises the average length of an helix to exactly k, loosing all variability during the process.

Thus, we proposes in [?] a way to control the values of the parameters of interest by putting weights $\pi_{i_1}, \pi_{i_1} \dots \pi_{i_\alpha}$ on the terminal letters $i_1, i_2 \dots i_\alpha$. We then define the weight of a sequence to be the product of the individual weights of its letters. For instance, in a weighted CFG model having weights $\pi_a = 2$, $\pi_b = 1$ and $\pi_c = 3$, a sequence *aacbcacbcbc* has a weight of $2^3.1^3.3^5 = 1944$.

By making a few adjustments during the precomputation stage, it is possible to constrain the probability of emission of a sequence to be equal to its weight, normalized by the sum of weights of all sequences of the same size. Under certain hypothesis, it can be proved that for any expected frequencies for the terminal letters, there exists weights such that the desired frequencies are observed. By assigning heavy weights to the nucleotides inside helices, and low ones to helices starts, it is now possible to control the average size of helices, terminal loops, bulges and multiloops.

To illustrate the effect of such weightings, we consider the previous example from figure 4.5, and we point the fact that the average number of unpaired bases equals to $\frac{58}{17} \approx 3.412\dots$ in the uniform model. By assigning a weight $\pi_c = 10$ to each *unpaired base* letter, we obtain the probabilities of emission for sequences of size 6 summarized in table 4.7.

Word ω	Probability $\mathbb{P}(\omega)$
`aaccbb`	$100/total$
`aacbcb`	$100/total$
`aacbbc`	$100/total$
`acacbb`	$100/total$
`acbacb`	$100/total$
`caacbb`	$100/total$
`accccb`	$10000/total$
`acccbc`	$10000/total$
`accbcc`	$10000/total$
`acbccc`	$10000/total$
`cacccb`	$10000/total$
`caccbc`	$10000/total$
`cacbcc`	$10000/total$
`ccaccb`	$10000/total$
`ccacbc`	$10000/total$
`cccacb`	$10000/total$
`cccccc`	1000000/total
$Total$	1100600

Figure 4.7: Probabilities for all words of size 6 under $\pi_c = 10$

These probabilities yield a new expectancy for the number of *unpaired bases* in a random structure of $\frac{506}{103} \approx 4.91\ldots$. So, by assigning a weight barely higher than $10(< 12)$, it is possible to constrain the average number of the unpaired base symbol `c` to be 5 out of 6. This property holds only for sequences of size 6, but another weight can be computed for any sequence size. Moreover, we claim that, under certain *common sense* hypothesis, the frequency of a given symbol quickly reaches an *asymptotic behaviour*, so that the weight do not need to be adapted to the sequence size above a certain point.

4.4 Implementing a CFG-based model

This section describes the syntax and semantics of context-free grammar based description files.

4.4.1 Main structure

```
TYPE = GRAMMAR
[SYMBOLS = ...]
[START = ...]
RULES = ...
[WEIGHTS = ...]
[ALIASES = ...]
```

Figure 4.8: Main structure of a context-free grammar description file

Clauses nested inside square brackets are optional. The given order for the clauses is **mandatory**.

4.4.2 Grammar generation specific clauses

Context-free grammar based description files define some attributes and properties of the grammar.

4.4.2.1 The SYMBOLS clause

SYMBOLS = {WORDS,LETTERS}

Optional, defaults to `SYMBOLS = WORDS`
Chooses the type of symbols to be used for random generation.
When `WORDS` is selected, each pair of symbols must be separated by at least a blank character (space, tabulation or newline). This affects mostly the `RULES` clause, where the right-hand-sides of the rules will be made of words, separated by blank characters. This feature can greatly improve the readability of the description file as shown in the example from figure 4.9.

```
RULES=
  tRNA -> Acceptor_Stem Acceptor_Site ;
  Acceptor_Stem -> A Acceptor_Stem U ;
  Acceptor_Stem -> MultiLoop ;
  MultiLoop -> Fill D_Stem Anticodon_Stem Variable_Loop T_Psi_C_Stem ;
  ...
```

Figure 4.9: The skeleton of a grammar for the tRNA

A `LETTERS` value for the `SYMBOLS` parameter will force GenRGenS to see a word as sequence of symbols. For instance, the right hand side of the rule `S->aSbS` will decompose into `S -> a S b S` if this option is invoked. Moreover, blank characters will be inserted between symbols in the output.

4.4.2.2 The START clause

START = nt

Optional, defaults to `START =` n_0, with n_0 being the left hand side of the first rule defined by the `RULES` clause.
Defines the axiom nt of the grammar, that is the non-terminal letter initiating the generation. This letter must of course be the left-hand-side of some rule inside the `RULES` clause below.

4.4.2.3 The RULES clause

RULES = nt_1 -> s_1; nt_2 -> s_2; ...

Required
Defines the rules of the grammar.
Rules are composed of letters, also called symbols. Each symbol nt_i that appears as the left-hand side of a rule becomes a non-terminal symbol, that is a symbol that needs further rewriting. s_i are sequences of symbols that can weither appear or not as the left-hand side of rules, separated by spaces or tabs if `SYMBOLS=LETTERS` is selected.
If the `START` clause is omitted, the axiom for the grammar will be nt_1.
The characters `::=` can be used instead of `->` to separate the left-hand side symbols from the right-hand side ones.

4.4.2.4 The WEIGHTS clause

WEIGHTS = $l_1 \quad w_1 \quad l_2 \quad w_2 \quad \ldots$
$w_i \in \mathbb{R}$

Optional, weights default to 1.
Defines the weights of the terminal letters l_i.
As discussed in section 4.3, assigning a weight w_i to a terminal letter l_i is a way to gain control over the average frequency of l_i. For instance, in a grammar over the alphabet $\{a, b\}$, adding a clause `WEIGHTS = a 2` will grant sequences `aaab` and `bbbb` with weights 8 and 1, i.e. the generation probability of `aaab` will be 8 times higher than that of `bbbb`.

4.4.3 A complete example

4.4.3.1 Source

The following description file captures some structural properties of the tRNA.

```
1 TYPE = GRAMMAR
2 SYMBOLS = WORDS
3 START = tRNA
4 RULES =
     tRNA -> A_Stem A_Site ;
     A_Stem -> A_u A_Stem a_U ; A_Stem -> U_a A_Stem u_A ;
     A_Stem -> C_g A_Stem c_G ; A_Stem -> G_c A_Stem g_C ;
     A_Stem -> MultiLoop ;
     A_Site -> A A_Site ; A_Site -> U A_Site ;
     A_Site -> C A_Site ; A_Site -> G A_Site ;
     A_Site -> ;
     MultiLoop -> Fill D_Stem C_Stem Variable_Loop T_Psi_C_Stem ;
     Fill -> A Fill ; Fill -> U Fill ; Fill -> C Fill ;
     Fill -> G Fill ; Fill -> ;
     D_Stem -> A_u D_Stem a_U ; D_Stem -> U_a D_Stem u_A ;
     D_Stem -> C_g D_Stem c_G ; D_Stem -> G_c D_Stem g_C ;
     D_Stem -> D_Loop ;
     D_Loop -> Fill ;
     C_Stem -> A_u C_Stem a_U ; C_Stem -> U_a C_Stem u_A ;
     C_Stem -> C_g C_Stem c_G ; C_Stem -> G_c C_Stem g_C ;
     C_Stem -> C_Loop ;
     C_Loop -> Fill ;
     Variable_Loop -> Fill;
     T_Psi_C_Stem -> A_u T_Psi_C_Stem a_U ; T_Psi_C_Stem -> U_a T_Psi_C_Stem u_A ;
     T_Psi_C_Stem -> C_g T_Psi_C_Stem c_G ; T_Psi_C_Stem -> G_c T_Psi_C_Stem g_C ;
     T_Psi_C_Stem -> T_Psi_C_Loop ;
     T_Psi_C_Loop -> Fill ;
5 WEIGHTS =
    A 0.125 U 0.125 C 0.125 G 0.125
6 ALIASES =
    A_u = A   a_U = U   C_g = C   c_G = G
    U_a = U   u_A = A   G_c = G   g_C = C
```

Figure 4.10: A basic CFG-based model for the tRNA

4.4.3.2 Semantics

In **1**, we choose a context free grammar-based model.

In **2**, we use words, a reader-friendly choice for such a complicated(though still a little simplistic) model.

In **3**, we choose the non-terminal symbol `tRNA` as our starting symbol. Every generated sequence will be obtained from iterative rewritings of this symbol. Here, this clause has no real effect, as the default behaviour of GenRGenS is to choose the first non-terminal to appear in a rule as the starting symbol.

Clause **4** describes the set of rules. The correspondance between non-terminal symbols an substructures is detailled in figure 4.11.

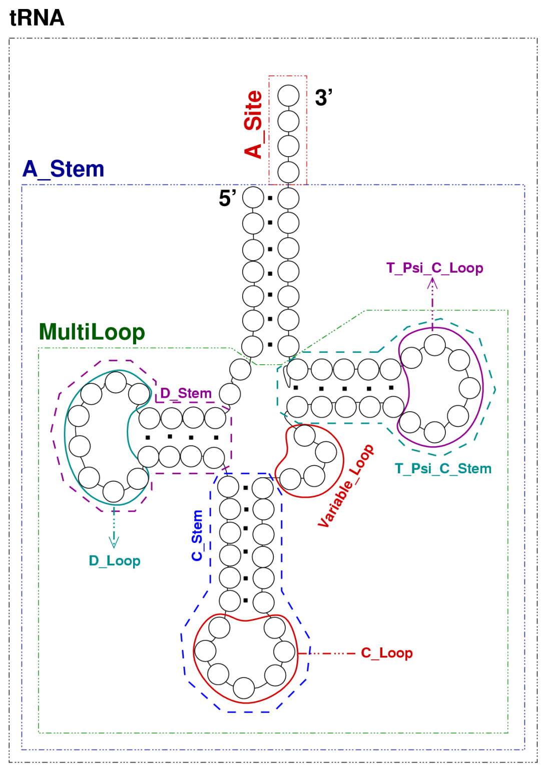


Figure 4.11: Non-terminal symbols and substructures

Note that we use different alphabets to discern paired bases from unpaired ones, which allows discrimination of unpaired bases using weights in clause **5**.
Of course, it is possible to get a more realistic model by substituting the `X_Loop -> Fill` rules with specific set of rules, using different alphabets. Thus, one would be able to control the size and composition of each loop.

In **5**, we discriminate the unpaired bases, which increases size of the helices. Indeed, uniformity tends to favor very small if not empty helices whereas the weighting scheme from this clause penalizes so much the occurences of unpaired bases that the generated structures now have very small loops.

In **6**, the sequence's alphabet is unified by removing traces of the structure.

4.5 Grammar-specific command line options

During the precomputation stage, `GenRGenS` uses arbitrary precision arithmetics to handle numbers that can grow over the size in an exponential way. However, for special languages or small sequences, it is possible to avoid using time-consumming arithmetics, and to limit the computation to *double* variables. This is the purpose of the **-f** option. The **-s** option toggles *on* and *off* the computation and display of frequencies of symbols for each generated sequence. **Grammar specific command line options usage:**

`java -cp .  GenRGenS.GenRGenS -size` *n* `-nb` *k* **-f** [**T**|**F**] **-s** [**T**|**F**] *GrammarGGDFile*

- **-f** [**T**|**F**]: Activates/deactivates the use of light and limited `double` variables instead of heavier but more accurate `BigInteger` ones. Misuse of this option may result in division by zero or non-uniform generation. **Defaults to -f F.**
- **-s** [**T**|**F**]: Enables/disables statistics about each sequence. **Defaults to -s F.**

Chapter 5

The RATIONAL package : Regular Expressions and PROSITE patterns

At the bottom of Chomsky's formal languages hierarchy lies the regular expressions. Taken as languages theory tools, their main purpose is the description of subsets over the whole set of sequences that can be parsed and/or generated by very simple machines called *finite state automata* over finite alphabets. This class is known to be a subclass of the one that is implemented by the context-free grammar, but more efficient random generation algorithms are available for it. Because of their simplicity, they have been extensively used to model families of genomic sequences differing only on a few positions. In the Prosite database, which is well known database of protein families and domains, a pattern (or consensus) is built from a given set of sequences, sharing functional properties. These so-called *Prosite patterns* are related to regular expressions, as shown in 5.1.3.
GenRGenS provides a random generation process both for Prosite patterns and regular expressions. For instance, random sequences drawn with respect to a Prosite pattern supposedly being a fingerprint for a biological property can be used to test its relevance. One can also generate simple mutants from a regular expression by introducing some *choice* places inside the sequence. Such sequences can used for computation of statistical scores, such as Z-Scores, and P-values.

5.1 Some theory

In this chapter, we will describe the syntax and semantics of the regular expressions. We will then explain how GenRGenS turns a ProSite pattern into a regular expression associated with the same language, and show how uniform random generation can be performed from a regular expression.

5.1.1 Regular expressions syntax

A regular expression e is a language description tool, recursively defined as follows :

$$e = \begin{cases} e'^{*} \\ e' \mid e'' \\ e' \ .\ e'' \\ (e') \\ \omega \\ \varepsilon \end{cases}$$

Where e_1 and e_2 are regular expressions, and l stands for any letter among the alphabet.
The ε is a shortcut for an empty word(a word of size 0).

Formally, a language can be seen as a set of words over a given alphabet. The meanings of these alternatives are related to the languages denoted by such expressions.

- The disjonction $e' \mid e''$: The language is the union of the languages associated to e' and e''. Any word of $e' \mid e''$ belongs to e' or e''.
- The concatenation e' . e'': The language is the concatenation of the languages associated to e' and e''. Any word among e' . e'' can be decomposed into a concatenation of words issued from e' and e''.
- The iteration e^*: Each word of the resulting language is a concatenation of a finite set of words issued from e.
- (e'): The language is that of e'. This construction is useful to avoid ambiguity. For instance, the expression $a|b.c$ can denote the language $\{a, bc\}$ or the language $\{ac, bc\}$.
- ω, ε: The only word among the language is the single character l, resp. ε.

Example 1:

$$e = (A.T.G).((A|T|G|C).(A|T|G|C).(A|T|G|C))^*.(T.A.G|T.G.A|T.A.A)$$

The regular expression e describes a simplistic model for an ORF, that is a subsequence of a DNA code starting with a START base triplet ATG and ending with one of the STOP base triplets TAG/TGA/TAA. It should be noticed that anything can happen between the START and STOP codon, as the $((A|T|G|C).(A|T|G|C).(A|T|G|C))^*$ part of e doesn't excludes STOP base triplets.

Example 2:
Such expressions are also perfectly fit to model mutants. Suppose you're given three 5.8S ribosomal RNA sequences close one to another and aligned as follows.

...	C	G	C	C	C	C	G	C	C	G	G	C	G	G	...
...	A	C	G	C	G	A	C	C	C	G	G	U	G	G	...
...	C	C	U	G	U	U	.	G	U	G	G	U	G	G	...

A regular expression e can then be used to model a family of sequences that contains the three original sequences, among with some other sequences close to the originals.

$$e = \cdots (C|A).(C|G).(C|G|U).(C|G).(C|G|U).(C|A|U).(G|C|\varepsilon).(C|G).(C|U).G.G.(C|U).G.G \cdots$$

5.1.2 PROSITE Patterns

5.1.2.1 What are PROSITE Patterns ?

PROSITE is a database of protein families and domains.It consists of biologically significant sites, patterns and profiles that help to identify to which known protein family a new sequence belongs. It was started in 1989 by Amos Bairoch, and is part of the SWISS-PROT program[?]. It features text-files structured by tags and carriage-returns. Among these tags, some are used to define patterns to model sequential aspects behind functional properties. They can be seen as *consensus*, as they capture some sequential similarities of a given entry set of sequences.
Further informations can be found at:

`http://www.expasy.org/prosite/prosuser.html`

Glycine	G
Alanine	A
Valine	V
Leucine	L
Isoleucine	I
Proline	P
Phenylalanine	F
Tyrosine	Y
Cysteine	C
Methionine	M
Histidine	H
Lysine	K
Arginine	R
Tryptophan	W
Serine	S
Threonine	T
Aspartic acid	D
Glutamic acid	E
Asparagine	N
Glutamine	Q
Aspartic acid or asparagine	B
Glutamic acid or glutamine	Z
Unknown	X

Figure 5.1: Standard codes for the amino-acids, as proposed by the IUPAC

5.1.2.2 Syntax

Let $\mathcal{P}$ be the set of all known amino-acids, abbreviated with respect to the standard one-letter IUPAC code[1] given by figure 5.1.2.2.

Then a PROSITE pattern p can then be recursively defined as follows:

$$p = \begin{cases} p'\texttt{-}p'' \\ p'\texttt{(}n\texttt{)} \\ p'\texttt{(}n_{min}\texttt{,}n_{max}\texttt{)} \\ \texttt{(}p'\texttt{)} \\ \texttt{[}l\texttt{]} \\ \texttt{\{}l\texttt{\}} \\ \texttt{x} \\ l \end{cases}$$

where p' and p'' are PROSITE patterns; n, n_{min} and n_{max} are positive integers; l is a sequence of amino-acids encoded with respect to the IUPAC code for the amino-acids. The semantics for the preceeding alternatives is described below :

- The concatenation $p'\texttt{-}p''$: protein codes are derived from patterns p' and p'' and concatenated.
- The strict iteration $p'\texttt{(}n\texttt{)}$: a protein code is derived from pattern p' and copied n times.
- The loose iteration $p'\texttt{(}n_{min}\texttt{,}n_{max}\texttt{)}$: a protein code is derived from pattern p' and copied from n_{min} to n_{max} times.

[1] See http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html for details

- The identity `(`p'`)`: This notation is equivalent to p', and simply means that a sequence is issued from p'. It can be used to resolve some ambiguities.

- The inclusive disjonction `[`l`]`: Any amino-acid code can be choosed from the list l.

- The exclusive disjonction $\{l\}$: Any amino-acid code that is not in the list l can be choosed.

- The wildcard `x`: Any amino-acid, including the unknown X code.

- The amino-acid sequence l: l is a word composed of amino-acid codes.

5.1.3 PROSITE patterns/Rational expressions relationship

As the rational expressions are also defined using concatenations, iterations and disjunctions, a PROSITE pattern can be considered as a rational expression. The correspondance between PROSITE elementary operations and rational ones are listed below:

$$\begin{array}{rcl}
p'\texttt{-}p'' & \Rightarrow & e(p')\,\texttt{.}\,e(p'') \\
p'\texttt{(}n\texttt{)} & \Rightarrow & e(p') \\
p'\texttt{(}n_{min}\texttt{,}n_{max}\texttt{)} & \Rightarrow & e(p')_1 \ldots e(p')_{n_{min}}\,\texttt{.}\,\texttt{(}\varepsilon\,\texttt{|}\,e(p')_{n_{min+1}}\,\texttt{.}\,\texttt{(}\varepsilon\,\texttt{|}\,e(p')_{n_{min+2}}\,\texttt{.}\,\texttt{(}\cdots\texttt{(}\varepsilon\,\texttt{|}\,e(p')_{n_{max}}\texttt{)}\cdots\texttt{)))} \\
\texttt{(}p'\texttt{)} & \Rightarrow & e(p') \\
\texttt{[}l\texttt{]} & \Rightarrow & \texttt{(}l_1\,\texttt{|}\,l_2\,\texttt{|}\ldots\texttt{|}\,l_{|l|}\texttt{)} \\
\{l\} & \Rightarrow & \texttt{(}k_1\,\texttt{|}\,k_2\,\texttt{|}\ldots\texttt{|}\,k_{|l|}\texttt{)}, k_i \in \text{IUPACCodes} - l \\
\texttt{x} & \Rightarrow & \texttt{(}k_1\,\texttt{|}\,k_2\,\texttt{|}\ldots\texttt{|}\,k_{|l|}\texttt{)}, k_i \in \text{IUPACCodes} \bigcup \{X\} \\
l & \Rightarrow & l
\end{array}$$

5.1.4 Uniform random generation among the language denoted by a Regular Expression

Its is a classical result of the formal language theory that any regular expression can be turned into an automaton that recognises/generates the same language. Such an automaton can be turned into a deterministic one, that is an automaton where each accepted word is uniquely coupled with a walk between two special states (start and final). Thus, instead of expanding a initial non-terminal symbol the way we did for the `GRAMMAR` package, we are going to walk inside of this automaton with carefuly chosed probabilities.

For instance, consider the ORFs inside of DNA. They start with a START codon `ATG`, then follows a region of codons that are not STOP-ones, and finally they end with the STOP codon `TAA`. This can be modelled by the following expression, whose deterministic automaton is shown in figure 5.2.

$$(ATG).(T.A.(C|G|T)|T.(C|G|T).(A|C|G|T)|(C|G|T).(A|C|G|T).(A|C|G|T))^*.(TAA)$$

Each sequence is bijectively associated with a single path in the resulting. For instance, the sequence `ATG AAT TCG GAT TAA` corresponds to the state sequence 1-2-3 7-8-9 5-8-9 7-8-9 5-6-10. As explained in the `GRAMMAR` package, the choice of the next state must be made using correct probabilities in order to achieve uniformity. These probabilities are proportional to the amount of words reachable from the destination state. Once again these numbers can be computed using simple linear recurrences before the generation stage.

Controlled non-uniform random generation can also be achieved using the same distributions as within the `GRAMMAR` package. Indeed, each symbol(letter, IUPAC Code) can be associated with a weight inside the `WEIGHT` clause, so that a sequence's probability will the product of all its letters' weights, normalized by the sum of all the weights over the sequence set denoted by the expression. This weight mechanism may be used for instance to increase the proportion of a given base as well as to favor the occurence of a given structured motif.

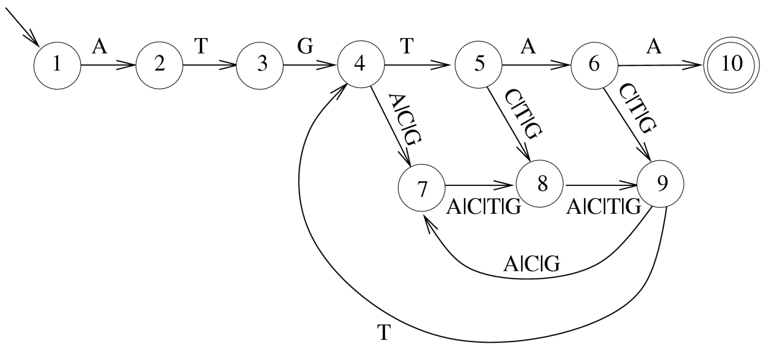


Figure 5.2: Deterministic automaton corresponding to the simple ORF model.

5.2 Implementing a Rational/PROSITE model

5.2.1 Main structure

The main structure of the file is shown in figure 5.3

```
TYPE = RATIONAL
LANGUAGE = RATIONAL | PROSITE
EXPRESSION = e | p
[WEIGHTS] = ...
[ALIASES = ...]
```

Figure 5.3: Main structure of a `RATIONAL` description file

5.2.2 `RATIONAL` generation specific clauses

5.2.2.1 The `LANGUAGE` clause

```
LANGUAGE = RATIONAL | PROSITE
```

Required
Chooses between rational/regular expression and PROSITE pattern syntaxes for the expression.

5.2.2.2 The `EXPRESSION` clause

```
EXPRESSION = e
```

Required
The rational or `PROSITE` expression. The syntax for e depends on the value of the `LANGUAGE` clause, and has always been described in section 5.1.1 for a `RATIONAL` choice, and in section 5.1.2.2 for PROSITE patterns.

5.2.2.3 The `WEIGHTS` clause

$$\texttt{WEIGHTS} \ = \ l_1 \quad w_1 \quad l_2 \quad w_2 \ \ldots$$
$$w_i \in \mathbb{R}$$

Optional, all weights default to 1.
Defines the weights of the terminal letters l_i.
As discussed in section 4.3 for the GRAMMAR description file, assigning a weight w_i to a terminal letter l_i is a way to gain control over the average frequency of l_i.

5.3 Examples

5.3.1 RATIONAL style example

5.3.1.1 Source

We show and explain the GenRGenS description file corresponding to the simple ORF model shown in figure 5.2.

```
1  TYPE = RATIONAL
2  LANGUAGE = RATIONAL
3  EXPRESSION = (A.T.G).(T.A.(C'|G'|T)|T.(C'|G'|T).(A|C'|G'|T)|
     (C'|G'|T).(A|C'|G'|T).(A|C'|G'|T))*.(T.A.A)
4  WEIGHTS =
5    G' 2 C' 2.5
6  ALIASES =
7    C'=C G'=G
```

5.3.1.2 Semantics

On line **1**, we choose the RATIONAL package for random generation.

On line **2**, we select a rational/regular expression.

On line **3**, the rational expression corresponding to the simple ORF model from figure 5.2. Notice that the star operator * has highest precedence, so that "A|B*"$\Leftrightarrow$"A|(B*)"$\neq$"(A|B)*". On lines **4** and **5**, weights are assigned to the C's and G's. A weight higher than 1 for a symbol will increase its number of occurence within generated sequences. Here, we choose to favor C and G among the coding area.

On lines **6** and **7**, the symbols C' and G' were only introduced to allow the assignment of weights within the coding area. Indeed, we needed the weight not to affect the G from the START codon. We don't really need them anymore, so we send them back to their original representations C and G.

5.3.2 PROSITE style example

We illustrate the use of GenRGenS' RATIONAL package to generate protein sequences with respect to a given PROSITE pattern by a *real-life* example. The pattern CBM1_1 can be found under accession code PS00562 on expasy's and is considered as a signature for the carbohydrate binding type-1 domain. Figure 5.4 shows its entry in the PROSITE database.

5.3.2.1 Source

The PA line of the PROSITE file can be inserted directly into the EXPRESSION clause, **without the terminal dot**, resulting in the following description file:

```
ID   CBM1_1; PATTERN.
AC   PS00562;
DT   DEC-1991 (CREATED); DEC-2004 (DATA UPDATE); JAN-2006 (INFO UPDATE).
DE   CBM1 (carbohydrate binding type-1) domain signature.
PA   C-G-G-x(4,7)-G-x(3)-C-x(4,5)-C-x(3,5)-[NHGS]-x-[FYWM]-x(2)-Q-C.
NR   /RELEASE=49.0,207132;
NR   /TOTAL=28(25); /POSITIVE=28(25); /UNKNOWN=0(0); /FALSE_POS=0(0);
NR   /FALSE_NEG=1; /PARTIAL=0;
CC   /TAXO-RANGE=??E??; /MAX-REPEAT=4;
CC   /SITE=1,disulfide; /SITE=7,disulfide; /SITE=9,disulfide;
CC   /SITE=16,disulfide;
CC   /VERSION=1;
DR   Q99034, AXE1_TRIRE , T; Q00023, CEL1_AGABI , T; Q9HE18, FAEB_PENFN , T;
DR   Q12714, GUN1_TRILO , T; P07981, GUN1_TRIRE , T; P07982, GUN2_TRIRE , T;
DR   Q12624, GUN3_HUMIN , T; O14405, GUN4_TRIRE , T; P43317, GUN5_TRIRE , T;
DR   P46236, GUNB_FUSOX , T; P46239, GUNF_FUSOX , T; P45699, GUNK_FUSOX , T;
DR   O59843, GUX1_ASPAC , T; P15828, GUX1_HUMGT , T; Q06886, GUX1_PENJA , T;
DR   P13860, GUX1_PHACH , T; Q9P8P3, GUX1_TRIHA , T; P62695, GUX1_TRIKO , T;
DR   P62694, GUX1_TRIRE , T; P19355, GUX1_TRIVI , T; Q92400, GUX2_AGABI , T;
DR   P07987, GUX2_TRIRE , T; P49075, GUX3_AGABI , T; P46238, GUXC_FUSOX , T;
DR   P50272, PSBP_PORPU , T;
DR   P38676, GUX1_NEUCR , N;
3D   1AZ6; 1AZH; 1AZK; 1CBH; 2CBH;
DO   PDOC00486;
```

Figure 5.4: PROSITE entry for pattern CBM1_1

```
1  TYPE = RATIONAL
2  LANGUAGE = PROSITE
3  EXPRESSION =
4    C-G-G-x(4,7)-G-x(3)-C-x(4,5)-C-x(3,5)-[NHGS]-x-[FYWM]-x(2)-Q-C
```

5.3.2.2 Semantics

On line **1**, we choose the RATIONAL package for random generation.

On line **2**, we select a PROSITE pattern expression.

On line **3** and **4**, the PROSITE pattern.

5.4 Rational expressions-specific command line options

The default behaviour of GenRGenS is to generate sequences of the size provided through the **-size** command line option. However, it can also be useful to generate among every words possibly drawn from a PROSITE expression, regardless of the size. This is the purpose of the **-i** option. The sequences are still drawn at random.

Rational specific command line options usage:

java -cp . GenRGenS.GenRGenS -size *n* -nb *k* **-i** [**T**|**F**] *PrositeGGDFile*

- **-i** [**T**|**F**]:When **T** is selected, ignores the *size* parameter, so that the sequences are drawn at random among the finite set corresponding to the PROSITE pattern defined in the file *PrositeGGDFile*. Generates sequences of the given *size* otherwise.
  Defaults to -i F.

Chapter 6

The MASTER package : Hierarchical models

Sometimes, true uniformity or total control over the distribution of the sequences is not needed, and the hierarchical aspects of some models can be captured. The MASTER generation is then a good tradeoff between computational efficiency and expressivity. The main principle is to generalize the idea behind hidden Markovian to any type of master and hidden states models. At first, a sequence is derived from a master description file, having a length passed to the main generation engine of GenRGenS. Then each of its symbols can either or not be defined *non-terminal* and rewritten using another description file coupled with a size distribution.

6.1 Hierarchical models: Principles

This class of models has been included to allow different models to be combined. For example, a simple model for the Intergenic/ORF alternation could describe the framed aspect of the ORFs (using the Framed Markov model) as well as the lack of a start codon in the intergenic areas (using the Rational expression model). This can be modelled using our *hierarchical models.*

Hierarchical models comprise a main *master* model, a set of auxiliary models and a set of rules defining how to rewrite letters of the master sequence using the auxiliary models. This can also be seen as a generalisation of the HMMs, as the hidden state sequence can be computed independently from the letters arising from each hidden state. A sequence length distribution can be provided with each rule through an expressive language, enabling complex constraints such as *the overall length of the sequence is a multiple of* 3 to be expressed.

6.1.1 Saving time and space

It is noticeable that this approach to random can save some time and space while using models that require more than linear time and space complexities. Indeed, if n_1 and n_2 are the sizes of the two parts of an expected sequence then $n_1^{1+\epsilon} + n_2^{1+\epsilon} < (n_1 + n_2)^{1+\epsilon}$, for all $\epsilon > 0$.

6.1.2 Ambiguity and bias

It also implies a small loss of control over the distribution of sequences. For instance, a model built up by concatenating two sequences chosen uniformly is unlikely to be uniform itself. More generally, let M_1 and M_2 be two models, a sequence s issued from the model $M_1.M_2$ may admit different decompositions $s = \alpha_1.\beta_1 = \alpha_2.\beta_2 = \ldots = \alpha_k.\beta_k$. Its global probability is then $p(s|M_1.M_2) = \sum_{i=1}^{k} p(\alpha_i|M_1)p(\beta_i|M_2)$, which induces a bias if uniformity is aimed at. Furthermore, as it is impossible to constrain the ending of a Markov chain, the concatenation of two sequences issued from some Markovian models may induce some *bias* at the cutpoint.

6.2 Implementing a hierarchical model

This section describes the syntax and semantics of `MASTER` description files.

6.2.1 Main structure

```
TYPE = MASTER
[SYMBOLS = ...]
MAINFILE = "MainFilePath"
WHERE =
  s1 :  "AuxFile1Path" SIZE = law1
  s2 :  "AuxFile2Path" SIZE = law2
  ...
```

Figure 6.1: Main structure of a `MASTER` description file

Clauses nested inside square brackets are optional. The given order for the clauses is **mandatory**.

6.2.2 Master generation specific clauses

A hierarchical model must define the way symbols from the main sequence are to be expanded.

6.2.2.1 The `SYMBOLS` clause

```
SYMBOLS = {WORDS,LETTERS}
```

Optional, defaults to `SYMBOLS = WORDS`
Chooses the type of symbols to be used for random generation.
When `WORDS` is selected, each pair of subsequent symbols is separated with spaces during the generation.

6.2.2.2 The `MAINFILE` clause

```
MAINFILE = "MainFilePath"
```

Required
Defines the main random generation model description file. The file must be accessible by appending the string *MainFilePath* to the current directory.
A so-called *master* sequence is generated from this description file for further rewritings. Its length equals to the `size` provided to GenRGenS.
Remark: The overall size of the sequence generated from a `MASTER` model is usually not related to the `size` parameter provided to GenRGenS.

6.2.2.3 The `WHERE` clause

```
WHERE =
  s_1 :  "AuxPath_1" SIZE = var_1
  s_2 :  "AuxPath_2" SIZE = var_2 ...
```

Required
Defines the way to expand each occurences of a symbol s_i from the main file to sequences issued from $AuxPath_i$, having size given by the distribution law_i.
Each s_i is a symbol that must belong to the vocabulary of the master file, but not necessarily to the sequence generated from it.
$AuxPath_i$ is the path to a description file that must be read-access enabled.
var_i is a description of the random value associated to the size of this symbol's expansion.

6.2.2.4 The `SIZE` definitions

A size can either be a constant, one of the predefined random variate generator, or an arithmetic expression defined recursively.

- integer or float constant: Generates a constant sequence size. If a non-integer size is specified at top-level, the value is rounded to the closest integer value, as it would be have been using `round`.
- `+`,`-`,`*`,`/`,`^` : Classical binary arithmetic operators, acting on real numbers and returning a real value.
- `(`e`)`: An expression nested by parenthesis is evaluated separately and priorly from its context. It can be used to overrule usual operator precedences. Ex : `7+3*5` $\neq$ `(7+3)*5`
- `pow(`e_1`,`e_2`)`: Functional form of the power operator `^` , returns $e_1^{e_2}$. If first argument e_1 is omitted, returns 2^{e_2}.
- `log(`e_1`,`e_2`)`: Returns $\log_{e_1}(e_2)$. If e_1 is omitted, returns $\log_2(e_2)$.
- `floor(`e`)`: Returns the closest integer value smaller than e.
- `round(`e`)`: Returns the closest integer to e.
- `ceil(`e`)`: Returns the closest integer value greater than e.
- `min(`e_1`,`e_2`)`: Returns the smallest number among $\{e_1, e_2\}$.
- `max(`e_1`,`e_2`)`: Returns the smallest number among $\{e_1, e_2\}$.
- `length`: Returns the former length of the *master* sequence. Equivalent syntax: `size` or `N`.
- `normal(`m`,`sd`)`: Generates a pseudorandom real value obeying a normal law having mean m and standard deviation sd. Equivalent syntax: `gaussian(`m`,`sd`)`.
- `uniform(`low`,`up`)`: Generates a pseudorandom integer value uniformly distributed over $[low, up)$ (e.g. $[low, up{-}1]$). If low is omitted, defaults to 0. Equivalent syntax: `random(`$lowm$`,`up`)`.
- `var(`ω`,`e`)`: Declares a variable ω, whose value always equals the most recent evaluation of e. Once declared, ω can be used anywhere among the SIZE declaration parts of the ggd. Each variable carries the value 0 as long as it is not assigned.

6.2.3 Capturing Dependencies between sequences size

`MASTER` package introduces a way to capture the dependency between two subsequences' size. A typical dependency that one may find useful would be : *The total size of the two subsequence is 100nt, although the first's size has been observed to vary with respect to a given distribution.* The `MASTER` package offers the possibility to declare variables, which is a powerful and *delicate* feature. Inside of a `MASTER` description file, simply enclose any expression of a length definition inside a `VAR` declaration. Ex.:

```
TYPE = MASTER
MAINFILE = "MainFile.ggd"
WHERE
A : "AuxFile1.ggd" SIZE VAR(X,UNIFORM(0,5)+1)
B : "AuxFile2.ggd" SIZE MAX(5-X,0)
```

In this example, each evaluation of the random variable `UNIFORM(0,5)+1` is assigned to a variable named `X`. Each evaluation of the expression `MAX(5-X,0)` will then make use of the current value of `X`. This is likely to trick one into bad assumptions over the resulting sequences, for instance if the sequences issued from `MainFile.ggd` do not only present simple `A`-`B` alternations. Suppose that `MainFile.ggd` contains an implementation of a simple `RATIONAL` model, based on the regular expression `(A|B|m)*`, that `AuxFile1.ggd` and `AuxFile2.ggd` are `RATIONAL` models based on the regular expressions `T*` and `U*`, and that a sequence ω is issued from `MainFile.ggd`. Rewritings according to the `SIZE` definitions above result in the following sequence ω'. It can be seen in figure

ω	m A_1 m m m B_1 m m m m A_2 m m B_2 m m $B_{2'}$ m m m A_3 m m A_4 m m m B_4 m
`X`	? **1** $- - - - - - - - \longrightarrow$ **3** $- - - - - - - - \longrightarrow$ **5** $- \longrightarrow$ **1** $- - - - - \longrightarrow$
`MAX(5-X,0)`	? 4 $- - -$ **4** $- - - - \longrightarrow$ 2 $- -$ **2** $- -$ **2** $- - \longrightarrow$ 0 $- \longrightarrow$ 4 $- - -$ **4** $\longrightarrow$

$$\Rightarrow \omega' = \mathrm{m}T_1\mathrm{mmm}UUUU_1\mathrm{mmmm}TTT_2\mathrm{mm}UU_2\mathrm{mm}UU_{2'}\mathrm{mmm}TTTTT_3\mathrm{mm}T_4\mathrm{mmm}UUUU_4\mathrm{m}$$

Figure 6.2: Dependencies during generation stage

6.2 that a variable's assignment can either be used by several subsequent dependant variables assignments, as in $A_2 \rightarrow B_2 \ldots B_{2'}$, or be ignored like A_4. Therefore, this feature should only be used when the sequences issued from `MainFile.ggd` are sufficiently constrained, for instance by showing a strict assignment/use alternation such as the regular expression `m*(A m* B m*)*`.

6.3 A complete example

6.3.1 Source

This example is inspired by a simple model for stem-loop ribosomal -1 frameshifting sites. A frameshitfting RNA causes the ribosome to *slip* over a specific sequence/structure motif and shift to another phase. A model inspired by the HIV-1 stem-loop frameshifting site is detailled in figure 6.3. The `Heptamer` part can be modeled with a regular expression, similar to a mutation model. The `Spacer` will use a Bernoulli model(model of order 0). The `Stem_Loop` requires the full power of a context free grammar. Lastly, the frame shift sites are drowned into an ocean of coding RNA(symbol `Fill`) modelled again by a markov model.

```
TYPE = MASTER
SYMBOLS = WORDS
MAINFILE = "frameshift.ggd"
WHERE =
  Heptamer :  "heptamers.ggd" SIZE = 7
  Spacer :  "spacers.ggd" SIZE = VAR(X,5 + UNIFORM(0,3))
  Stem_Loop :  "stem_loops.ggd" SIZE = VAR(Y,24 + UNIFORM(0,9))
  Fill :  "junk.ggd" SIZE = 3*FLOOR((NORMAL(300 , 100)+X+Y+7)/3)-X-Y-7
```

Figure 6.4: A `MASTER` description file for the frameshift model from figure 6.3.1

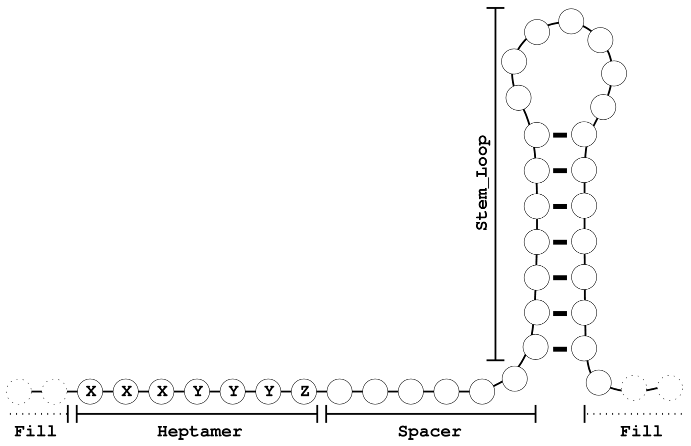


Figure 6.3: A simple stem-loop frameshifting model. `X` stands for *any base*, `Y` means `A` *or* `U` and `Z` is *anything but* `C`

File `frameshift.ggd`

```
TYPE = MARKOV
ORDER = 1
SYMBOLS = WORDS
START = Fill 1
FREQUENCIES =
  Fill Heptamer 1
  Heptamer Spacer 1
  Spacer Stem_Loop 1
  Stem_Loop Fill 1
```

File `heptamers.ggd`

```
TYPE = RATIONAL
LANGUAGE = RATIONAL
SYMBOLS = WORDS
EXPRESSION =
  (A|U|C|G).(A|U|C|G).(A|U|C|G).
  (A|U).(A|U).(A|U).(A|U|G)
```

File `spacers.ggd`

```
TYPE = MARKOV
ORDER = 0
SYMBOLS = WORDS
FREQUENCIES =
  A 10 U 15
  C 20 G 25
```

File `stem_loops.ggd.ggd`

```
TYPE = GRAMMAR
RULES =
  S -> A S U ; S -> U S A ;
  S -> C S G ; S -> G S C ;
  S -> L ;
  L -> A' L ; L -> C' L;
  L -> G' L ; L -> U' L;
  L -> ;
WEIGHTS =
  A' 0.4 C' 0.5 G' 0.5 U' 0.3
ALIASES =
  A'=A G'=G C'=C U'=U
```

File `junk.ggd`

```
TYPE = MARKOV
ORDER = 0
SYMBOLS = WORDS
FREQUENCIES =
  A 20 C 25
  C 30 U 35
```

Figure 6.5: Auxiliary description files

6.3.2 Semantics

On line **1**, a `MASTER` generator is choosed.

On line **2**, we use words as symbols to avoid confusion.

On line **3**, we define the main description file, whose content is listed in figure 6.5.

On line **5**, we ask the generator to rewrite each occurence of the symbol `Heptamer` with a sequence issued from `heptamers.ggd`, having size 7.

On line **6**, we ask the generator to rewrite each occurence of `Spacer` with a sequence issued from `spacers.ggd`, using a size uniformly distributed over $[5, 7)$. Note that `UNIFORM(5,8)` is an equivalent syntax for `5 + UNIFORM(0,3)`. After each evaluation, the resulting size is stored inside a variable `X`.

On line **7**, we ask the generator to rewrite each occurence of `Stem_Loop` with a sequence issued from `stem_loops.ggd`, using a size uniformly distributed over $[24, 33)$. After each evaluation, the resulting size is stored inside a variable `Y`.

On line **8**, we ask the generator to rewrite each occurence of `Fill` using a size that depends on the previous assignments. The somehow cabalistic formula for the size of `Fill`'s expansion is just a little trick to ensure that the `Heptamer` motif always occurs on phase 0. That is, `NORMAL(300 , 100)+X+Y+7` must be a multiple of 3. So, by dividing by 3 and rounding to the closest smaller integer and then multiplying by 3, we get the closest sum of the sizes of `Heptamer`, `Spacer`, `Stem_Loop` and `Fill` that is divided by 3 without a remainder. It suffices then to substract 7 (`Heptamer`'s size), `X` (`Spacer`'s size) and `Y` (`Stem_Loop`'s size), to get an elligible size for `Fill`.