INFERNAL User's Guide

Sequence analysis using profiles of RNA secondary structure consensus

http://infernal.janelia.org/
Version 0.71; December 2006

Sean Eddy
HHMI Janelia Farm
19700 Helix Drive
Ashburn VA 20147
http://selab.janelia.org/

Contents

1 Introduction

INFERNAL is a software package that allows you to make consensus RNA secondary structure profiles, and use them to search nucleic acid sequence databases for homologous RNAs, or to create new structure-based multiple sequence alignments.

To make a profile, you need to have a multiple sequence alignment of an RNA sequence family, and the alignment must be annotated with a consensus RNA secondary structure. The program **`cmbuild`** takes an annotated multiple alignment as input, and outputs a profile.

You can then use that profile to search a sequence database for homologs, using the program **`cmsearch`**.

You can also use the profile to align a set of unaligned sequences to the profile, producing a structural alignment, using the program **`cmalign`**. This allows you to build hand-curated representative alignments of RNA sequence families, then use a profile to automatically align any number of sequences to that profile. This seed alignment/full alignment strategy combines the strength of stable, carefully human-curated alignments with the power of automated updating of complete alignments as sequence databases grow. This is the strategy used to maintain the RFAM database of RNA multiple alignments and profiles.

INFERNAL is comparable to HMMER (hmmer.janelia.org). The HMMER software package builds profile hidden Markov models (profile HMMs) of multiple sequence alignments. Profile HMMs capture only primary sequence consensus features. INFERNAL models are profile stochastic context-free grammars (profile SCFGs). Profile SCFGs include both sequence and RNA secondary structure consensus information.

Currently INFERNAL is really just an algorithm testbed. Output is rudimentary, and some desired features are missing. Most importantly, INFERNAL is very slow and CPU-intensive. You will probably need a large number of CPUs in order to use it for serious work. Planned algorithmic improvements should make it more practical in the future. We are making it available as a fully documented package now, only because INFERNAL has been pressed prematurely into service as the basis for constructing and maintaining the RFAM database of structurally annotated RNA multiple alignments (Griffiths-Jones et al., 2003). When we assign a 1.0 release number, that's when we'll think INFERNAL is ready for prime time. Until then, please bear with us.

2 Installation

Quick installation instructions

Download the source tarball (**infernal.tar.gz**) from ftp://selab.janelia.org/pub/software/infernal/ or http://infernal.janelia.org

Unpack the software:

```
> tar xvf infernal.tar.gz
```

Go into the newly created top-level directory (named either **infernal**, or **infernal-xx** where **xx** is a release number:

```
> cd infernal
```

Configure for your system, and build the programs:

```
> ./configure
> make
```

Run the automated testsuite. This is optional. All these tests should pass:

```
> make check
```

The programs are now in the **src/** subdirectory. The user's guide (this document) is in the **documentation/userguide** subdirectory. The man pages are in the **documentation/manpages** subdirectory. You can manually move or copy all of these to appropriate locations if you want. You will want the programs to be in your $PATH.

Optionally, you can install the man pages and programs in system-wide directories. If you are happy with the default (programs in **/usr/local/bin/** and man pages in **/usr/local/man/man1**), do:

```
> make install
```

That's all. More complete instructions follow, including how to change the default installation directories for **make install**.

More detailed installation notes

INFERNAL is distributed as ANSI C source code. It is designed to be built and used on UNIX platforms. It is developed on Intel GNU/Linux systems, and intermittently tested on a variety of vendor-donated UNIX platforms including Sun/Solaris, HP/UX, Digital Tru64, Silicon Graphics IRIX, IBM/AIX, and Intel/FreeBSD. It is not currently tested on either Microsoft Windows or Apple OS/X. It should be possible to build it on any platform with an ANSI C compiler. The software itself is vanilla POSIX-compliant ANSI C. You may need to work around the configuration scripts and Makefiles to get it built on a non-UNIX platform.

The GNU configure script that comes with INFERNAL has a number of options. You can see them all by doing:

```
> ./configure --help
```

All customizations can and should be done at the **./configure** command line, unless you're a guru delving into the details of the source code.

setting installation targets

The most important options are those that let you set the installation directories for **make install** to be appropriate to your system. What you need to know is that INFERNAL installs only two types of files: programs and man pages. It installs the programs in **--bindir** (which defaults to **/usr/local/bin**), and the man pages in the **man1** subdirectory of **--mandir** (default **/usr/local/man**). Thus, say you want **make**

install to install programs in **/usr/bioprogs/bin/** and man pages in **/usr/share/man/man1**; you would configure with:

```
> ./configure --mandir=/usr/share/man --bindir=/usr/bioprogs/bin
```

That's really all you need to know, since INFERNAL installs so few files. But just so you know; GNU configure is very flexible, and has shortcuts that accomodates several standard conventions for where programs get installed. One common strategy is to install all files under one directory, like the default **/usr/local**. To change this prefix to something else, say **/usr/mylocal/** (so that programs go in **/usr/mylocal/bin** and man pages in **/usr/mylocal/man/man1**, you can use the **--prefix** option:

```
> ./configure --prefix=/usr/mylocal
```

Another common strategy (especially in multiplatform environments) is to put programs in an architecture-specific directory like **/usr/share/Linux/bin** while keeping man pages in a shared, architecture-independent directory like **/usr/share/man/man1**. GNU configure uses **--exec-prefix** to set the path to architecture dependent files; normally it defaults to being the same as **--prefix**. You could change this, for example, by:

```
> ./configure --prefix=/usr/share --exec-prefix=/usr/share/Linux/
```

In summary, a complete list of the **./configure** installation options that affect INFERNAL:

Option	Meaning	Default
--prefix=PREFIX	architecture independent files	**/usr/local/**
--exec-prefix=EPREFIX	architecture dependent files	PREFIX
--bindir=DIR	programs	EPREFIX/bin/
--mandir=DIR	man pages	PREFIX/man/

setting compiler and compiler flags

By default, **configure** searches first for the GNU C compiler **gcc**, and if that is not found, for a compiler called **cc**. This can be overridden by specifying your compiler with the **CC** environment variable.

By default, the compiler's optimization flags are set to **-g -O2** for **gcc**, or **-g** for other compilers. This can be overridden by specifying optimization flags with the **CFLAGS** environment variable.

For example, to use an Intel C compiler in **/usr/intel/ia32/bin/icc** with optimization flags **-O3 -ipo**, you would do:

```
> env CC=/usr/intel/ia32/bin/icc CFLAGS="-O3 -ipo" ./configure
```

which is the one-line shorthand for:

```
> setenv CC /usr/intel/ia32/bin/icc
> setenv CFLAGS "-O3 -ipo"
> ./configure
```

If you are using a non-GNU compiler, you will almost certainly want to set **CFLAGS** to some sensible optimization flags for your platform and compiler. The **-g** default generated unoptimized code. At a minimum, turn on your compiler's default optimizations with **CFLAGS=-O**.

turning on Large File Support (LFS)

INFERNAL has one optional feature: support for Large File System (LFS) extensions that allow programs to access files larger than 2 GB. LFS is rapidly becoming standard, but not yet standard enough to be default. If you do anything with Genbank files or large genome files (like the 3 GB human genome), you will need LFS support. LFS is enabled with the **--enable-lfs** option:

```
> ./configure --enable-lfs
```

installing rigorous filters

INFERNAL includes programs by Zasha Weinberg that implement rigorous filtering. This software requires a C++ compiler, and also relies on an external library, CFSQP, that is not included in this distribution, but can be obtained by request from http://www.aemdesign.com/. To build the executables, include these two options to configure:

```
> ./configure --with-rigfilters --with-cfsqp=/path/to/cfsqp
```

Example configuration

The Intel GNU/Linux version installed at Janelia Farm is configured as follows:

```
> env CFLAGS="-O3" ./configure --enable-lfs --prefix=/usr/local/infernal-xx
```

3 Getting started

Here's a quick walk-through of the package. Here, we will a) build a model of an RNA multiple alignment using **cmbuild**; b) use that model to search for new homologs using **cmsearch**; and c) use that model to align new sequences, and create a new multiple alignment, using **cmalign**.

We'll use the following files, all of which are in the **intro/** subdirectory of the distribution:

tutorial.sto A multiple alignment of five tRNA sequences. This file is a simple example of *Stockholm format* that INFERNAL uses for structurally-annotated alignments.

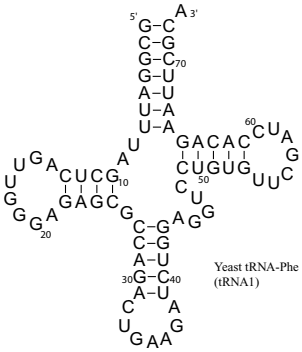
tutorial.db A tiny sequence "database"; a 300 nt sequence that contains a tRNA. The file is in FASTA format, which INFERNAL uses for unaligned sequence data.

tutorial.fa The same sequences as in **tutorial.sto**, plus one more tRNA with an internal deletion (to demonstrate local alignment), in unaligned FASTA format.

Format of a simple input RNA alignment file

Look at the alignment file **tutorial.sto** in the **intro/** subdirectory of the INFERNAL distribution. It is shown below, with a secondary structure of the first sequence shown to the right for reference (yeast Phe tRNA, labeled as "tRNA1" in the file):

```
# STOCKHOLM 1.0

tRNA1             GCGGAUUUAGCUCAGUUGGG.AGAGCGCCAGACUGAAGAUCUGGAGGUCC
tRNA2             UCCGAUAUAGUGUAAC.GGCUAUCACAUCACGCUUUCACCGUGGAGA.CC
tRNA3             UCCGUGAUAGUUUAAU.GGUCAGAAUGGGCGCUUGUCGCGUGCCAGA.UC
tRNA4             GCUCGUAUGGCGCAGU.GGU.AGCGCAGCAGAUUGCAAAUCUGUUGGUCC
tRNA5             GGGCACAUGGCGCAGUUGGU.AGCGCGCUUCCCUUGCAAGGAAGAGGUCA
#=GC SS_cons      <<<<<<<..<<<<.........>>>>.<<<<<.......>>>>>.....<

tRNA1             UGUGUUCGAUCCACAGAAUUCGCA
tRNA2             GGGGUUCGACUCCCCGUAUCGGAG
tRNA3             GGGGUUCAAUUCCCCGUCGCGGAG
tRNA4             UUAGUUCGAUCCUGAGUGCGAGCU
tRNA5             UCGGUUCGAUUCCGGUUGCGUCCA
#=GC SS_cons      <<<<.......>>>>>>>>>>>>.
//
```



This is a simple example of a multiple RNA sequence alignment with secondary structure annotation, in *Stockholm format*. Stockholm format, the native alignment format used by HMMER and INFERNAL and the PFAM and RFAM databases, is documented in detail later in the guide.

For now, what you need to know about the key features of the input file is:

- The alignment is in an interleaved format, like other common alignment file formats such as CLUSTALW. Lines consist of a name, followed by an aligned sequence; long alignments are split into blocks separated by blank lines.
- Each sequence must have a unique name. (This is important!)
- For residues, any one-letter IUPAC nucleotide code is accepted, including ambiguous nucleotides. Case is ignored; residues may be either upper or lower case.
- Gaps are indicated by the characters ., _, -, or ~. (Blank space is not allowed.)

- A special line starting with `#=GC SS_cons` indicates the secondary structure consensus. Gap characters annotate unpaired (single-stranded) columns. Base pairs are indicated by any of the following pairs: `<>`, `()`, `[]`, or `[]`. No pseudoknots are allowed; the open/close-brackets notation is only unambiguous for strictly nested base-pairing interactions.

- The file begins with the special tag line `# STOCKHOLM 1.0`, and ends with `//`.

Building a model with `cmbuild`

To build a model from this alignment, do:

```
> cmbuild my.cm tutorial.sto
```

Almost instantly, **`cmbuild`** reads in the alignment, constructs a model, and saves that model to the new file **`my.cm`**. It is a convention to use the **`.cm`** suffix for model files; CM stands for "covariance model", another name for the profile SCFG architecture used by INFERNAL (Eddy and Durbin, 1994).

The output **`cmbuild`** contains information about the size of your input alignment (in aligned columns and # of sequences), and about the size of the resulting model. You don't need to understand this to use the model, so for now we'll skip describing the output.

The result, the model file in **`my.cm`** is a text file. You can look at it (e.g. **`> more my.cm`**) if you like, but it isn't really designed to be human-interpretable. You can treat **`.cm`** files as compiled models of your RNA alignment.

Searching a sequence database with `cmsearch`

You can use your model to search for new homologues of your RNA family. The file **`tutorial.db`** contains an example sequence "database": one 300 nt sequence, with yeast tRNA-Phe embedded at position 101...173. To search it, do:

```
> cmsearch my.cm tutorial.db
```

`cmsearch` now searches both strands of each sequence in the target database, and returns alignments for high scoring hits. In this case, it returns one hit:

```
sequence: example
hit 0   :    101    173    22.29 bits
           (((((((,,<<<<___.____>>>>,<<<<<_______>>>>>,,,,,<<<<<_______
         1 gccgauaUagcgcAgU.GGuAgcgcgccacccUgucaagguggAGgUCcggggUUCGAUu 59
           GC:+AU:UAGC:CAGU GG AG:GCGCCA::CUG ++A::UGGAGGUCC:G:GUUCGAU+
       101 GCGGAUUUAGCUCAGUuGGGAGAGCGCCAGACUGAAGAUCUGGAGGUCCUGUGUUCGAUC 160

           >>>>>)))))))):
        60 ccccguaucggcg 72
           C:C:G:AU+:GC+
       161 CACAGAAUUCGCA 173
```

Hits are numbered starting from 0. This one shows hit 0, from position 101 to 173 on the sequence named "example", with a score of 22.29 bits. There are no E-value statistics yet in INFERNAL, so this score is all you have to go by to determine significance of a hit. Larger scores are better. As a rough guide, scores greater than the log (base two) of the target database size are significant. Here, given a 600 nt target (300 nt $\times$ 2 strands), scores over 9-10 bits are significant - so a score of 22.29 is a good hit.

The alignment is shown in a BLAST-like format, augmented by secondary structure annotation.

The top line shows the predicted secondary structure of the target sequence. The format is a little fancier and more informative than the simple least-common-denominator format we used in the input alignment

file. It's designed to make it easier to "see" the secondary structure by eye. The format is described in detail later; for now, here's all you need to know. Base pairs in simple stem loops are annotated with `<>` characters. Base pairs enclosing multifurcations (multiple stem loops) are annotated with `()`, such as the tRNA acceptor stem in this example. In more complicated structures, `[]` and `{}` annotations also show up, to reflect deeper nestings of multifurcations. For single stranded residues, _ characters mark hairpin loops; characters mark interior loops and bulges; , characters mark single-stranded residues in multifurcation loops; and : characters mark single stranded residues external to any secondary structure. Insertions relative to this consensus are annotated by a `.` character.

The second line shows that consensus of the query model. The highest scoring residue sequence is shown. Upper case residues are highly conserved. Lower case residues are weakly conserved or unconserved.

The third line shows where the alignment score is coming from. For a consensus base pair, if the observed pair is the highest-scoring possible pair according to the consensus, both residues are shown in upper case; if a pair has a score of ≥ 0, both residues are annotated by : characters (indicating an acceptable compensatory base pair); else, there is a space, indicating that a negative contribution of this pair to the alignment score. For a single-stranded consensus residue, if the observed residue is the highest scoring possibility, the residue is shown in upper case; if the observed residue has a score of ≥ 0, a `+` character is shown; else there is a space, indicating a negative contribution to the alignment score.

Finally, the fourth line is the target sequence.

Creating new multiple alignments with `cmalign`

You can also use a model to structurally align any number of new RNA sequences to your consensus structure. This is how the RFAM database is constructed: we start with "seed" alignment, build a CM of it, and use that CM to align all known members of the sequence family and create a "full" alignment. This allows us to maintain representative seed alignments that are stable and small enough to be human-curated, while still being able to automatically incorporate and align all homologues detected in the rapidly growing public sequence databases.

An example of some unaligned tRNA sequences are in the file **`tutorial.fa`**. (In fact, these are the same sequences that are in **`tutorial.sto`**, reformatted into unaligned FASTA format; plus a new sequence, tRNA6, which was created by deleting some residues out of the middle of tRNA1. tRNA6 will be used a little later to demonstrate local alignment.)

To align these sequences to the model we made in **`my.cm`**, do:

```
> cmalign my.cm tutorial.fa
```

This results in an alignment that looks like:

```
# STOCKHOLM 1.0
#=GF AU    Infernal 0.54

tRNA1             GCGGAUUUAGCUCAGUuGGG.AGAGCGCCAGACUGAAGAUCUGGAGGUCC
tRNA2             UCCGAUAUAGUGUAAC.GGCuAUCACAUCACGCUUUCACCGUGGAGA-CC
tRNA3             UCCGUGAUAGUUUAAU.GGUcAGAAUGGGCGCUUGUCGCGUGCCAGA-UC
tRNA4             GCUCGUAUGGCGCAGU.GGU.AGCGCAGCAGAUUGCAAAUCUGUUGGUCC
tRNA5             GGGCACAUGGCGCAGUuGGU.AGCGCGCUUCCCUUGCAAGGAAGAGGUCA
tRNA6             GCGGAUUUAGCUCAGUuGGG.AGAGCGCC-------AGA---CGAGGUCC
#=GC SS_cons      (((((((,,<<<<___.___._>>>>,<<<<<_______>>>>>,,,,,<
#=GC RF           gccgauaUagcgcAgU.GGu.AgcgcgccacccUgucaagguggAGgUCc

tRNA1             UGUGUUCGAUCCACAGAAUUCGCA
tRNA2             GGGGUUCGACUCCCCGUAUCGGAG
tRNA3             GGGGUUCAAUUCCCCGUCGCGGAG
tRNA4             UUAGUUCGAUCCUGAGUGCGAGCU
tRNA5             UCGGUUCGAUUCCGGUUGCGUCCA
tRNA6             UGUGUUCGAUCCACAGAAUUCGCA
#=GC SS_cons      <<<<_______>>>>>)))))))):
#=GC RF           ggggUUCGAUuccccguaucggcg
//
```

In the aligned sequences, a `.` character indicates an inserted column relative to consensus; the `.` character is an alignment pad. A `-` character is a deletion relative to consensus.

The symbols in the consensus secondary structure annotation line have the same meaning that they did in a pairwise alignment from **`cmsearch`**.

The `#=GC RF` line is *reference annotation*. Non-gap characters in this line mark consensus columns; **`cmalign`** uses the residues of the consensus sequence here, with upper case denoting strongly conserved residues, and lower case denoting weakly conserved residues. Gap characters (specifically, the `.` pads) mark insertions relative to consensus. As described below, **`cmbuild`** is capable of reading these RF lines, so you can specify which columns are consensus and which are inserts (otherwise, **`cmbuild`** makes an automated guess, based on the frequency of gaps in each column).

If you want to save the alignment to a file, you can use the **`-o`** option:

```
> cmalign -o my.sto my.cm tutorial.fa
```

We'll use this **`my.sto`** alignment file in the next section.

Using optional annotation to completely specify model architecture to `cmbuild`

`cmbuild` needs to know two things to convert your alignment into a profile SCFG.

First, it needs to know the consensus secondary structure. It reads this from the `#=GC SS_cons` line, as described above. This annotation is mandatory.

It also needs to know which columns are consensus, and which columns are insertions relative to consensus. By default, it will determine this by a simple rule: if a column contains more than a certain fraction of gap characters (default $>$50%), the column is called an insertion. This may not be what you want; for instance, maybe you are trying to iteratively build models based on larger and larger numbers of sequences (based on an RFAM seed, say), but you don't want the curated consensus model architecture to change just because you added some new sequences to the alignment.

You can optionally override that default and specify the complete architecture of the model, using both a `#=GC SS_cons` structure annotation line and a `#=GC RF` reference column annotation line. To do this, you use the **`--rf`** flag to **`cmbuild`**.

For example, to build a model called **second.cm** from **my.sto** that has the same architecture as **my.cm**, you would do:

```
> cmbuild --rf second.cm my.sto
```

Since **cmalign** leaves an RF line on the alignments it generates, the **--rf** option allows you to propagate your consensus structure into new, larger alignments. The RF line is also handy when you want the model's coordinate system to be the same as a canonical, well-studied single sequence: you can simply use that sequence as the RF line, or manually create any consensus coordinate system you like. (This is the origin of RF as the "reference line", e.g. giving a reference coordinate system.) The only thing that matters in the RF line is nongap versus gap characters: the line can be as simple as x's marking consensus columns, `.`'s for insert columns.

Using local alignment in `cmsearch` and `cmalign`

The examples above required the entire model to match a subsequence of the target: so-called *glocal* alignment (global with respect to the query model, local with respect to the target sequence). But in many cases, a homologous RNA structure has undergone enough changes that parts of its structure cannot be aligned to the consensus. *Local* alignment, in which only part of the query model needs to match the target to detect a hit, can be a more sensitive searching strategy.

In primary sequence alignment, local alignment means an alignment of two subsequences of the query and target. In aligning a query RNA structure to a target sequence, local alignment means starting and ending at points inside the query structure – which, when you map that idea onto linear sequence, means an alignment that may consist of more than one discontinuous subsequence. We'll demonstrate this by example for now, and describe local alignment is described in detail later. For the purposes of the tutorial, all you really need to know is how to activate it. It is not the default behavior for either **cmsearch** or **cmalign**.

Local alignment is activated for **cmsearch** by using the **--local** option. For example:

```
> cmsearch --local my.cm tutorial.fa
```

Look at the last alignment in this output, the alignment for tRNA6:

```
sequence: tRNA6
hit 0   :      1     63    11.52 bits
           (((((((,,<<<<___.____>>>>,<~~~~~~>,,,,,<<<<<_______>>>>>))))
         1 gccgauaUagcgcAgU.GGuAgcgcgc*[15]*gAGgUCcggggUUCGAUuccccguauc 68
           GC:+AU:UAGC:CAGU GG AG:GCGC      GAGGUCC:G:GUUCGAU+C:C:G:AU+
         1 GCGGAUUUAGCUCAGUuGGGAGAGCGC*[ 5]*GAGGUCCUGUGUUCGAUCCACAGAAUU 59

           ))):
        69 ggcg 72
           :GC+
        60 CGCA 63
```

The `*[15]*` and `*[5]*` in the query and target, respectively, indicate that 15 consensus residues and 5 target residues were left unaligned; the target does not appear to have the consensus structure in this region. (No kidding, since I made the tRNA6 example sequence by deleting part of the anticodon stem.) The structure annotation line is marked with `~~~~~~` to indicate the gap in the alignment, and to distinguish local alignment induced gaps from normal insertions (which are marked with `.` characters).

You can activate local alignment in **cmalign** with the **-l** option:

```
> cmalign -l my.cm tutorial.fa
```

This results in the following alignment: [1]

```
# STOCKHOLM 1.0
#=GF AU    Infernal 0.54

tRNA1             GCGGAUUUAGCUCAGUuGGG.AGAGCGCCAGACUGAAGA.....UCUGGA
tRNA2             UCCGAUAUAGUGUAAC.GGCuAUCACAUCACGCUUUCAC.....CGUGGA
tRNA3             UCCGUGAUAGUUUAAU.GGUcAGAAUGGGCGCUUGUCGC.....GUGCCA
tRNA4             GCUCGUAUGGCGCAGU.GGU.AGCGCAGCAGAUUGCAAA.....UCUGUU
tRNA5             GGGCACAUGGCGCAGUuGGU.AGCGCGCUUCCCUUGCAA.....GGAAGA
tRNA6             GCGGAUUUAGCUCAGUuGGG.AGAGCGC-----------cagac----GA
#=GC SS_cons      (((((((,,<<<<___.___._>>>>,<<<<<_______~~~~~>>>>>,
#=GC RF           gccgauaUagcgcAgU.GGu.Agcgcgccacc Ugucaa~~~~~gguggA

tRNA1             GGUCCUGUGUUCGAUCCACAGAAUUCGCA
tRNA2             GA-CCGGGGUUCGACUCCCCGUAUCGGAG
tRNA3             GA-UCGGGGUUCAAUUCCCCGUCGCGGAG
tRNA4             GGUCCUUAGUUCGAUCCUGAGUGCGAGCU
tRNA5             GGUCAUCGGUUCGAUUCCGGUUGCGUCCA
tRNA6             GGUCCUGUGUUCGAUCCACAGAAUUCGCA
#=GC SS_cons      ,,,,<<<<<_______>>>>>))))))):
#=GC RF           GgUCcggggUUCGAUuccccguaucggcg
//
```

Note how the local alignment is represented for tRNA6. The deleted consensus columns are marked by - characters. The unaligned "insertion" is shown in its own columns; those columns are again marked with ~ characters in the consensus secondary structure annotation and the reference (RF) annotation lines.

An important limitation to `cmsearch`: the -W option

`cmsearch` implements a "scanning CYK" dynamic programming algorithm (Durbin et al., 1998) that looks for high-scoring alignments of the structural model to any subsequence of the target sequence. It works even if the target sequence is very large (e.g. a whole chromosome) – though it is slow, like all SCFG algorithms. An important feature of the scanning CYK algorithm is that it needs to know the *maximum length* of an aligned target subsequence, e.g. the size of its scanning window on the target sequence.

By default, the scanning window is calculated by **`cmbuild`** and stored in the model. The calculation is fairly rigorous, based on the expected probability density over possible lengths (Nawrocki and Eddy, 2007). The result will almost certainly be fine for your purposes. However, you can override it and set the scanning window width yourself to `<n>` with the **`-W <n>`** option.

Even if the target RNA is larger than the scanning window, there is some chance of finding part of it – particularly in combination with local alignment (**`--local`**), which explicitly allows partial matches to the query model. Thus, it may be possible to speed up a search with a large model by deliberately scanning a genome with **`--local`** and a small window size, looking for possible hits, then realigning candidate regions with a more appropriate value of **`-W`**. This strategy has not been systematically tested.

[1] The discontinuity of structural local alignment presents a quandary for representing multiple alignments. On the one hand, you might not want to even show the unaligned target residues in the gap (e.g., cagac) – they aren't aligned to the model. On the other hand, you sort of expect that if you pull an RNA sequence out of a multiple alignment, it represents a true subsequence of a larger sequence, not a concatenation of disjoint subsequences – you'd at least like some indication of where some residues have gone missing. One option would be to leave a *[5]* in the gap, as in the pairwise representation; but one of the nice properties of Stockholm format is that it's easy to interconvert it to other alignment formats just by stripping off everything by the name/sequence part of the alignment, and sticking non-sequence characters like *[5]* in the alignment would prevent that.

Getting more information

For a quick refresher on the command line usage of any program and its commonly used options, just type the name of the program with no other arguments: e.g.

```
> cmbuild
```

and you'll get a brief help:

```
FATAL: Incorrect number of arguments.
Usage: cmbuild [-options] <cmfile output> <alignment file>
The alignment file is expected to be in Stockholm format.
  Available options are:
   -h     : help; print brief help on version and usage
   -n <s> : name this CM <s>
   -A     : append; append this CM to <cmfile>
   -F     : force; allow overwriting of <cmfile>
```

For version information and a complete listing of options, use the **-h** option with any program, e.g.

```
> cmsearch -h
```

and you'll see something like:

```
Infernal 0.54 (Jan 2003)
Copyright (C) 2002-2003 Washington University, Saint Louis
Freely distributed under the GNU General Public License (GPL)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Usage: cmsearch [-options] <cmfile> <sequence file>
The sequence file is expected to be in FASTA format.
  Most commonly used options are:
   -h     : help; print brief help on version and usage
   -W <n> : set scanning window size to <n> (default: 200)

  Expert, in development, or infrequently used options are:
   --informat <s>: specify that input alignment is in format <s>, not FASTA
   --toponly     : only search the top strand
   --local       : do local alignment
   --dumptrees   : dump verbose parse tree information for each hit
```

More detailed information on usage and command line options is available in UNIX manual pages. If they have been installed for your system, you can see this information with, e.g.:

```
> man cmalign
```

Copies of the man pages are also provided at the end of this guide.

4 Profile SCFG construction: the `cmbuild` program

INFERNAL builds a model of consensus RNA secondary structure using a formalism called a *covariance model* (CM), which is a type of *profile stochastic context-free grammar* (profile SCFG) (Eddy and Durbin, 1994; Durbin et al., 1998; Eddy, 2002).

What follows is a technical description of what CM is, how it corresponds to a known RNA secondary structure, and how it is built and parameterized.[2] You certainly don't have to understand the technical details of CMs to understand **`cmbuild`** or INFERNAL, but it will probably help to at least skim this part. After that is a description of what the **`cmbuild`** program does to build a CM from an input RNA multiple alignment, and how to control the behavior of the program.

Technical description of a covariance model

Definition of a stochastic context free grammar

A stochastic context free grammar (SCFG) consists of the following:

- M different nonterminals (here called *states*). I will use capital letters to refer to specific nonterminals; V and Y will be used to refer generically to unspecified nonterminals.
- K different terminal symbols (e.g. the observable alphabet, a,c,g,u for RNA). I will use small letters a, b to refer generically to terminal symbols.
- a number of *production rules* of the form: $V \rightarrow \gamma$, where γ can be any string of nonterminal and/or terminal symbols, including (as a special case) the empty string ϵ.
- Each production rule is associated with a probability, such that the sum of the production probabilities for any given nonterminal V is equal to 1.

SCFG productions allowed in CMs

A CM is a specific, repetitive SCFG architecture consisting of groups of model states that are associated with base pairs and single-stranded positions in an RNA secondary structure consensus. A CM has seven types of states and production rules:

State type	Description	Production	Emission	Transition
P	(pair emitting)	$P \rightarrow aYb$	$e_v(a,b)$	$t_v(Y)$
L	(left emitting)	$L \rightarrow aY$	$e_v(a)$	$t_v(Y)$
R	(right emitting)	$R \rightarrow Ya$	$e_v(a)$	$t_v(Y)$
B	(bifurcation)	$B \rightarrow SS$	1	1
D	(delete)	$D \rightarrow Y$	1	$t_v(Y)$
S	(start)	$S \rightarrow Y$	1	$t_v(Y)$
E	(end)	$E \rightarrow \epsilon$	1	1

Each overall production probability is the independent product of an emission probability e_v and a transition probability t_v, both of which are position-dependent parameters that depend on the state v (analogous

[2] Much of this text is taken from (Eddy, 2002).

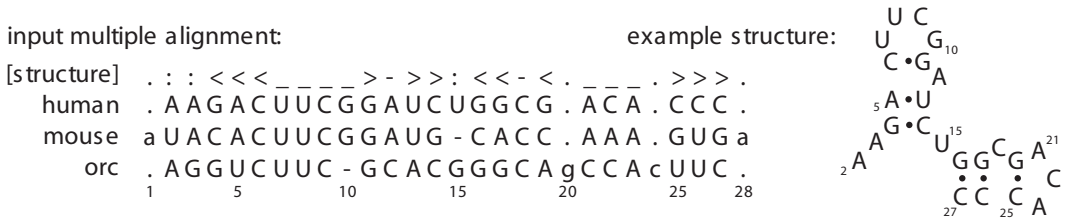


Figure 1: **An example RNA sequence family.** Left: a toy multiple alignment of three sequences, with 28 total columns, 24 of which will be modeled as consensus positions. The [structure] line annotates the consensus secondary structure in WUSS notation. Right: the secondary structure of the "human" sequence.

to hidden Markov models). For example, a particular pair (P) state v produces two correlated letters a and b (e.g. one of 16 possible base pairs) with probability $e_v(a, b)$ and transits to one of several possible new states Y of various types with probability $t_v(Y)$. A bifurcation (B) state splits into two new start (S) states with probability 1. The E state is a special case ϵ production that terminates a derivation.

A CM consists of many states of these seven basic types, each with its own emission and transition probability distributions, and its own set of states that it can transition to. Consensus base pairs will be modeled by P states, consensus single stranded residues by L and R states, insertions relative to the consensus by more L and R states, deletions relative to consensus by D states, and the branching topology of the RNA secondary structure by B, S, and E states. The procedure for starting from an input multiple alignment and determining how many states, what types of states, and how they are interconnected by transition probabilities is described next.

From consensus structural alignment to guide tree

Figure 1 shows an example input file: a multiple sequence alignment of homologous RNAs, with a line in WUSS notation that describes the consensus RNA secondary structure. The first step of building a CM is to produce a binary *guide tree* of *nodes* representing the consensus secondary structure. The guide tree is a parse tree for the consensus structure, with nodes as nonterminals and alignment columns as terminals.

The guide tree has eight types of nodes:

Node	Description	Main state type
MATP	(pair)	P
MATL	(single strand, left)	L
MATR	(single strand, right)	R
BIF	(bifurcation)	B
ROOT	(root)	S
BEGL	(begin, left)	S
BEGR	(begin, right)	S
END	(end)	E

These consensus node types correspond closely with the CM's final state types. Each node will eventually contain one or more states. The guide tree deals with the consensus structure. For individual sequences, we will need to deal with insertions and deletions with respect to this consensus. The guide tree is the skeleton on which we will organize the CM. For example, a MATP node will contain a P-type state to model a

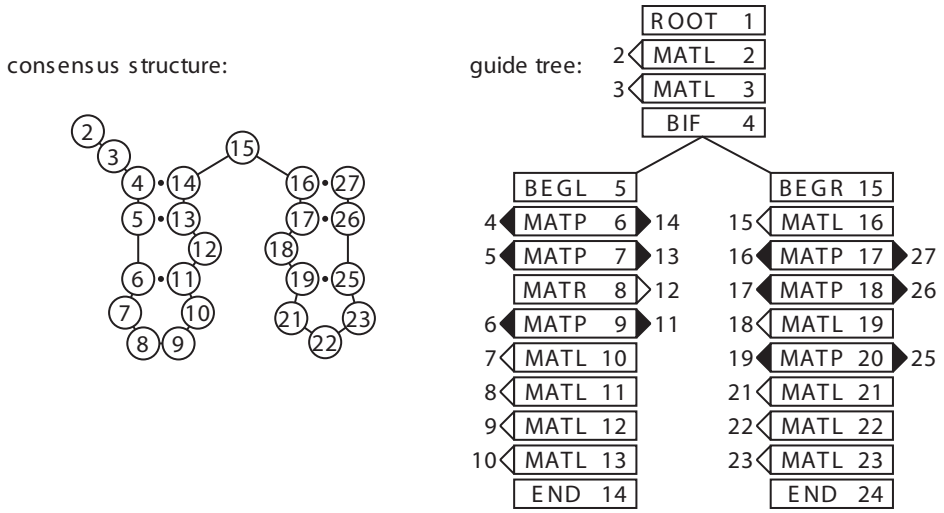


Figure 2: **The structural alignment is converted to a guide tree.** Left: the consensus secondary structure is derived from the annotated alignment in Figure 1. Numbers in the circles indicate alignment column coordinates: e.g. column 4 base pairs with column 14, and so on. Right: the CM guide tree corresponding to this consensus structure. The nodes of the tree are numbered 1..24 in preorder traversal (see text). MATP, MATL, and MATR nodes are associated with the columns they generate: e.g., node 6 is a MATP (pair) node that is associated with the base-paired columns 4 and 14.

consensus base pair; but it will also contain several other states to model infrequent insertions and deletions at or adjacent to this pair.

The input alignment is first used to construct a consensus secondary structure (Figure 2) that defines which aligned columns will be ignored as non-consensus (and later modeled as insertions relative to the consensus), and which consensus alignment columns are base-paired to each other. For the purposes of this description, I assume that both the structural annotation and the labeling of insert versus consensus columns is given in the input file, as shown in the alignment in Figure 1, where both are are indicated by the WUSS notation in the [structure] line (where, e.g., insert columns are marked with `.`). (In practice, **`cmbuild`** does need secondary structure annotation, but it doesn't require insert/consensus annotation or full WUSS notation in its input alignment files; this would require a lot of manual annotation. More on this later.)

Given the consensus structure, consensus base pairs are assigned to MATP nodes and consensus unpaired columns are assigned to MATL or MATR nodes. One ROOT node is used at the head of the tree. Multifurcation loops and/or multiple stems are dealt with by assigning one or more BIF nodes that branch to subtrees starting with BEGL or BEGR head nodes. (ROOT, BEGL, and BEGR start nodes are labeled differently because they will be expanded to different groups of states; this has to do with avoiding ambiguous parse trees for individual sequences, as described below.) Alignment columns that are considered to be insertions relative to the consensus structure are ignored at this stage.

In general there will be more than one possible guide tree for any given consensus structure. Almost all of this ambiguity is eliminated by three conventions: (1) MATL nodes are always used instead of MATR nodes where possible, for instance in hairpin loops; (2) in describing interior loops, MATL nodes are used before MATR nodes; and (3) BIF nodes are only invoked where necessary to explain branching secondary structure stems (as opposed to unnecessarily bifurcating in single stranded sequence). One source of ambiguity remains. In invoking a bifurcation to explain alignment columns $i..j$ by two substructures on columns

$i..k$ and $k+1..j$, there will be more than one possible choice of k if $i..j$ is a multifurcation loop containing three or more stems. The choice of k impacts the performance of the divide and conquer algorithm; for optimal time performance, we will want bifurcations to split into roughly equal sized alignment problems, so I choose the k that makes $i..k$ and $k+1..j$ as close to the same length as possible.

The result of this procedure is the guide tree. The nodes of the guide tree are numbered in preorder traversal (e.g. a recursion of "number the current node, visit its left child, visit its right child": thus parent nodes always have lower indices than their children). The guide tree corresponding to the input multiple alignment in Figure 1 is shown in Figure 2.

From guide tree to covariance model

A CM must deal with insertions and deletions in individual sequences relative to the consensus structure. For example, for a consensus base pair, either partner may be deleted leaving a single unpaired residue, or the pair may be entirely deleted; additionally, there may be inserted nonconsensus residues between this pair and the next pair in the stem. Accordingly, each node in the master tree is expanded into one or more *states* in the CM as follows:

Node	States	total # states	# of split states	# of insert states
MATP	[MP ML MR D] IL IR	6	4	2
MATL	[ML D] IL	3	2	1
MATR	[MR D] IR	3	2	1
BIF	[B]	1	1	0
ROOT	[S] IL IR	3	1	2
BEGL	[S]	1	1	0
BEGR	[S] IL	2	1	1
END	[E]	1	1	0

Here we distinguish between consensus ("M", for "match") states and insert ("I") states. ML and IL, for example, are both L type states with L type productions, but they will have slightly different properties, as described below.

The states are grouped into a *split set* of 1-4 states (shown in brackets above) and an *insert set* of 0-2 insert states. The split set includes the main consensus state, which by convention is first. One and only one of the states in the split set must be visited in every parse tree (and this fact will be exploited by the divide and conquer algorithm). The insert state(s) are not obligately visited, and they have self-transitions, so they will be visited zero or more times in any given parse tree.

State transitions are then assigned as follows. For bifurcation nodes, the B state makes obligate transitions to the S states of the child BEGL and BEGR nodes. For other nodes, each state in a split set has a possible transition to every insert state in the *same* node, and to every state in the split set of the *next* node. An IL state makes a transition to itself, to the IR state in the same node (if present), and to every state in the split set of the next node. An IR state makes a transition to itself and to every state in the split set of the next node.

This arrangement of transitions guarantees that (given the guide tree) there is unambiguously one and only one parse tree for any given individual structure. This is important. The algorithm will find a maximum likelihood parse tree for a given sequence, and we wish to interpret this result as a maximum likelihood

structure, so there must be a one to one relationship between parse trees and secondary structures (Giegerich, 2000).

The final CM is an array of M states, connected as a directed graph by transitions $t_v(y)$ (or probability 1 transitions $v \rightarrow (y, z)$ for bifurcations) with the states numbered such that $(y, z) \geq v$. There are no cycles in the directed graph other than cycles of length one (e.g. the self-transitions of the insert states). We can think of the CM as an array of states in which all transition dependencies run in one direction; we can do an iterative dynamic programming calculation through the model states starting with the last numbered end state M and ending in the root state 1. An example CM, corresponding to the input alignment of Figure 1, is shown in Figure 3.

As a convenient side effect of the construction procedure, it is guaranteed that the transitions from any state are to a *contiguous* set of child states, so the transitions for state v may be kept as an offset and a count. For example, in Figure 3, state 12 (an MP) connects to states 16, 17, 18, 19, 20, and 21. We can store this as an offset of 4 to the first connected state, and a total count of 6 connected states. We know that the offset is the distance to the next non-split state in the current node; we also know that the count is equal to the number of insert states in the current node, plus the number of split set states in the next node. These properties make establishing the connectivity of the CM trivial. Similarly, all the parents of any given state are also contiguously numbered, and can be determined analogously. We are also guaranteed that the states in a split set are numbered contiguously. This contiguity is exploited by the divide and conquer implementation.

Parameterization

Using the guide tree and the final CM, each individual sequence in the input multiple alignment can be converted unambiguously to a CM parse tree, as shown in Figure 4. Weighted counts for observed state transitions and singlet/pair emissions are then collected from these parse trees. These counts are converted to transition and emission probabilities, as maximum *a posteriori* estimates using mixture Dirichlet priors.

Comparison to profile HMMs

The relationship between an SCFG and a covariance model is analogous to the relationship of hidden Markov models (HMMs) and profile HMMs for modeling multiple sequence alignments (Krogh et al., 1994; Durbin et al., 1998; Eddy, 1998). A comparison may be instructive to readers familiar with profile HMMs. A profile HMM is a repetitive HMM architecture that associates each consensus column of a multiple alignment with a single type of model node – a MATL node, in the above notation. Each node contains a "match", "delete", and "insert" HMM state – ML, IL, and D states, in the above notation. The profile HMM also has special begin and end states. Profile HMMs could therefore be thought of as a special case of CMs. An unstructured RNA multiple alignment would be modeled by a guide tree of all MATL nodes, and converted to an unbifurcated CM that would essentially be identical to a profile HMM. (The only difference is trivial; the CM root node includes a IR state, whereas the start node of a profile HMM does not.) All the other node types (especially MATP, MATR, and BIF) and state types (e.g. MP, MR, IR, and B) are SCFG augmentations necessary to extend profile HMMs to deal with RNA secondary structure.

The cmbuild program, step by step

The **`cmbuild`** command line syntax is:

```
> cmbuild <options> [cmfile] [alifile]
```

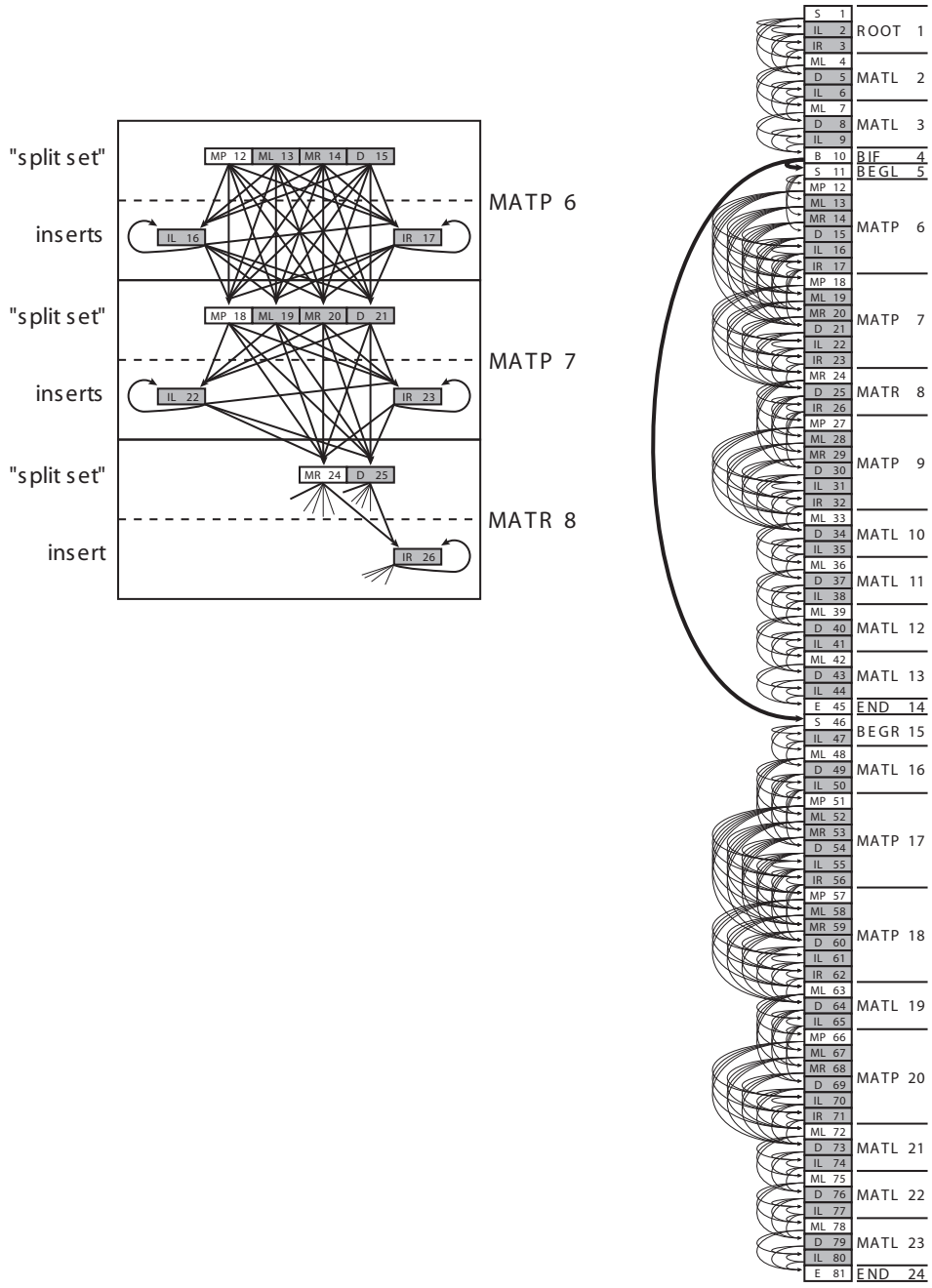


Figure 3: **A complete covariance model.** Right: the CM corresponding to the alignment in Figure 1. The model has 81 states (boxes, stacked in a vertical array). Each state is associated with one of the 24 nodes of the guide tree (text to the right of the state array). States corresponding to the consensus are in white. States responsible for insertions and deletions are gray. The transitions from bifurcation state B10 to start states S11 and S46 are in bold because they are special: they are an obligate (probability 1) bifurcation. All other transitions (thin arrows) are associated with transition probabilities. Emission probability distributions are not represented in the figure. Left: the states are also arranged according to the guide tree. A blow up of part of the model corresponding to nodes 6, 7, and 8 shows more clearly the logic of the connectivity of transition probabilities (see main text), and also shows why any parse tree must transit through one and only one state in each "split set".

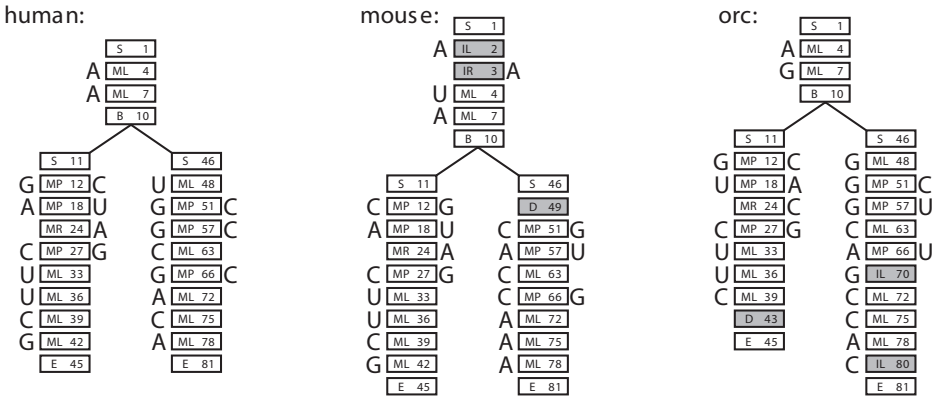


Figure 4: **Example parse trees.** Parse trees are shown for the three sequences/structures from Figure 1, given the CM in Figure 3. For each sequence, each residue must be associated with a state in the parse tree. (The sequences can be read off its parse tree by starting at the upper left and reading counterclockwise around the edge of parse tree.) Each parse tree corresponds directly to a secondary structure – base pairs are pairs of residues aligned to MP states. A collection of parse trees also corresponds to a multiple alignment, by aligning residues that are associated with the same state – for example, all three trees have a residue aligned to state ML4, so these three residues would be aligned together. Insertions and deletions relative to the consensus use nonconsensus states, shown in gray.

where **[alifile]** is the name of the input alignment file, and **[cmfile]** is the name of the output CM file. What follows describes the steps that **cmbuild** goes through, and the most important options that can be chosen to affect its behavior.

Alignment input file

The input alignment file must be in Stockholm format, and it must have a consensus secondary structure annotation line (`#=GC SS_cons`).

The program is actually capable of reading many common multiple alignment formats (ClustalW, PHYLIP, GCG MSF, and others) but no other format currently supports consensus RNA secondary structure annotation. This may change in the future, either when other formats allow structure annotation, or when **cmbuild** is capable of inferring consensus structure from the alignment by automated comparative analysis, as the earlier COVE suite was capable of (Eddy and Durbin, 1994).

If the file does not exist, is not readable, or is not in a recognized format, the program exits with a "could not be opened for reading" error. If the file does not have consensus secondary structure annotation, the program exits with a "no consensus structure annotation" error. This includes all non-Stockholm alignment files.

> ▷ ***Why does cmbuild have a `--informat` option, if it only accepts Stockholm?*** *If you don't specify* ***`--informat`****, the software has to autodetect the file format. Autodetection of file formats doesn't work in certain advanced/nonstandard cases, for instance if you're reading the alignment from standard input instead of from a file. The* ***`--informat`*** *allows you to override autodetection; e.g.* ***`cat my.sto | cmbuild --informat Stockholm my.cm -`*** *is an example of reading the alignment from piped standard input.*

Parsing secondary structure annotation

The structure annotation line only needs to indicate which columns are base paired to which. It does not have to be in full WUSS notation. Even if it is, the details of the notation are largely ignored. Nested pairs of `<>`, `()`, `[]`, or `{}` symbols are interpreted as base paired columns. All other columns marked with the symbols `:,_-.~` are interpreted as single stranded columns.

A simple minimal annotation is therefore to use `<>` symbols to mark base pairs and `.` for single stranded columns.

If a secondary structure annotation line is in WUSS notation and it contains valid pseudoknot annotation (e.g. additional non-nested stems marked with AAA,aaa or BBB,bbb, etc.), this annotation is removed and a warning is printed. INFERNAL cannot handle pseudoknots. Internally, these columns are treated as if they were marked with `.` symbols.

> ▷ ***How should I choose to annotate pseudoknots?*** INFERNAL *can only deal with nested base pairs. If there is a pseudoknot, you have to make a choice of which stem to annotate as normal nested structure (thus including it in the model) and which stem to call additional "pseudoknotted" structure (thus ignoring it in the model). For example, for a simple two-stem pseudoknot, should you annotate it as* `AAAA.<<<<aaaa....>>>>`*, or* `<<<<.AAAA>>>>....aaaa`*? From an RNA structure viewpoint, which stem I label as the pseudoknotted one is an arbitrary choice; but since one of the stems in the pseudoknot will have to be modeled as a single stranded region by* INFERNAL*, the choice makes a slight difference in the performance of your model. You want your model to capture as much information content as possible. Thus, since the information content of the model is a sum of the sequence conservation plus the additional information contributed by pairwise correlations in base-paired positions, you should tend to annotate the shorter stem as the "pseudoknot" (modeling as many base pairs as possible), and you should also annotate the stem with the more conserved primary sequence as the "pseudoknot" (if one stem is more conserved at the sequence level, you won't lose as much by modeling that one as primary sequence consensus only).*

If (aside from any ignored pseudoknot annotation) the structure annotation line contains characters other than `<>()[]{}:_-,.~` then those characters are ignored (treated as `.`) and a warning is printed.

If, after this "data cleaning", the structure annotation is inconsistent with a secondary structure (for example, if the number of `<` and `>` characters isn't the same), then the program exits with a "failed to parse consensus structure annotation" error.

Sequence weighting

By default, the input sequences are weighted in two ways to compensate for biased sampling (phylogenetic correlations). Relative sequence weights are calculated by the Gerstein/Chothia/Sonnhammer method (Gerstein et al., 1994). (The **`--wgsc`** option forces GSC weights, but is redundant since that's the default.) To turn relative weighting off (e.g. set all weights to 1.0), use the **`--wnone`** option.

Some alignment file formats allow relative sequence weights to be given in the file. This includes Stockholm format, which has `#=GS WT` weight annotations. Normally **`cmbuild`** ignores any such input weights. The **`--wgiven`** option tells **`cmbuild`** to use them. This lets you set the weights with any external procedure you like; for example, the **`weight`** utility program in SQUID implements some common weighting algorithms, including the fast $O(N)$ Henikoff position-based weights (Henikoff and Henikoff, 1994).

If for some reason you put more than one relative weighting option on the command line, the last one you give is used.

> ▷ ***Why is cmbuild taking so much time?*** *The GSC weighting algorithm scales as $O(N^2)$ with the number of sequences N. Weighting may become rate-limiting for* **`cmbuild`** *if your alignment contains*

many sequences. Model construction itself is fast. You might want to turn weighting off, or pre-calculate the weights by a faster algorithm.

Absolute weights (the "effective sequence number") is calculate by "entropy weighting" (Karplus et al., 1998). This sets the balance between the prior and the data, and affects the information content of the model. Entropy weighting reduces the effective sequence number (the total sum of the weights) and increases the entropy (degrading the information content) of the model until a threshold is reached. The default entropy is 1.46 bits per position (roughing 0.54 bits of information, relative to uniform base composition). This threshold can be changed with the **`--etarget <x>`** option. Entropy weighting may be turned off entirely with the **`--effnone`** option.

Architecture construction

The CM architecture is now constructed from your input alignment and your secondary structure annotation, as described in the previous section.

The program needs to determine which columns are consensus (match) columns, and which are insert columns. (Remember that although WUSS notation allows insertions to be annotated in the secondary structure line, **`cmbuild`** is only paying attention to annotated base pairs.) By default, it does this by a simple rule based on the frequency of gaps in a column. If the frequency of gaps is greater than a threshold, the column is considered to be an insertion.

The threshold defaults to 0.5. It can be changed to another number `<x>` (from 0 to 1.0) by the **`--gapthresh <x>`** option. The higher the number, the more columns are included in the model. At **`--gapthresh 1.0`**, all the columns are considered to be part of the consensus. At **`--gapthresh 0.0`**, none of them are (probably not a good idea).

You can also manually specify which columns are consensus versus insert by including reference coordinate annotation (e.g. a `#=GC RF` line, in Stockholm format) and using the **`--rf`** option. Any columns marked by non-gap symbols become consensus columns. (The simplest thing to do is mark consensus columns with x's, and insert columns with `.`'s. Remember that spaces aren't allowed in alignments in Stockholm format.) If you set the **`--rf`** option but your file doesn't have reference coordinate annotation, the program exits with an error.

Parameterization

Weighted observed emission and transition counts are then collected from the alignment data. These count vectors c are then converted to estimated probabilities p using mixture Dirichlet priors. The default mixture priors are described in (Nawrocki and Eddy, 2007). You can provide your own prior as a file, using the **`--priorfile <f>`** option.

Naming the model

Each CM gets a name. Stockholm format allows the alignment to have a name, provided in the `#=GF ID` tag. If this name is provided, it is used as the CM name.

If a name is not provided, the name is the input filename, without any extension – for example, if you build a model from the alignment file `RNaseP.sto`, the model will be named RNaseP.

You can override this and provide your own name with the **`-n <s>`** option, where `<s>` is any string.

Stockholm format also allows more than one alignment per file, and **cmbuild** supports this: CM files can contain more than one model, and if you say e.g. **cmbuild Rfam Rfam.sto** where Rfam.sto contains a whole database of alignments, **cmbuild** will create a database of CMs in the Rfam file, one per alignment. But in this case, obviously you don't want them all to have the same name! Therefore when running **cmbuild** on a multi-multiple alignment database file, the alignment database file *must* provide #=GF ID tags with names for each alignment. If any alignment is found to not have one, the program exits at that point with an error. Attempting to set the **--n** option for an alignment database also results in an error.

Saving the model

The model is now saved to a file, according to the filename specified on the command line. By default, a new file is created, and the model is saved in a portable ASCII text format.

If the cmfile already exists, the program exits with an error. The **--F** option causes the new model to overwrite an existing cmfile. The **--A** option causes the new model to be appended to an existing cmfile (creating a growing CM database, perhaps).

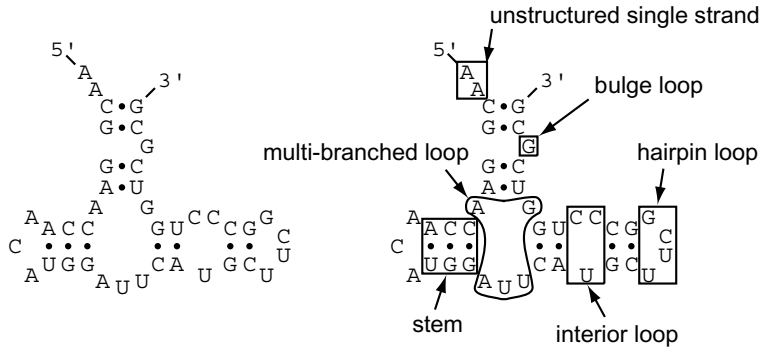
5 File and output formats

RNA secondary structures: WUSS notation

INFERNAL annotates RNA secondary structures using a linear string representation called "WUSS notation" (Washington University Secondary Structure notation).

The symbology is extended from the common bracket notation for RNA secondary structures, where open- and close-bracket symbols (or parentheses) are used to annotate base pairing partners: for example, `((((...))))` indicates a four-base stem with a three-base loop. Bracket notation is difficult for humans to interpret, for anything much larger than a simple stem-loop. WUSS notation makes it somewhat easier to interpret the annotation for larger structures.

The following figure shows an example with the key elements of WUSS notation. At the top left is an example RNA structure. At the top right is the same structure, with different RNA structural elements marked. Below both structure pictures : the WUSS notation string for the structure.




```
::((((,<<<___>>>,,,<<-<<____>>-->>,))-))
AACGGAACCAACAUGGAUUCAUGCUUCGGCCCUGGUCGCG
```

Full (output) WUSS notation

In detail, symbols used by WUSS notation in *output* structure annotation strings are as follows:

Base pairs Base pairs are annotated by nested matching pairs of symbols `<>`, `()`, `[]`, or `{}`. The different symbols indicate the "depth" of the helix in the RNA structure as follows: `<>` are used for simple terminal stems; `()` are used for "internal" helices enclosing a multifurcation of all terminal stems; `[]` are used for internal helices enclosing a multifurcation that includes at least one annotated `()` stem already; and `{}` are used for all internal helices enclosing deeper multifurcations.

Hairpin loops Hairpin loop residues are indicated by underscores, `_`. Simple stem loops stand out as, e.g. `<<<<_____>>>>`.

Bulge, interior loops Bulge and interior loop residues are indicated by dashes, `-`.

Multifurcation loops Multifurcation loop residues are indicated by commas, `,`. The mnemonic is "stem 1, stem2", e.g. `<<<___>>>,,<<<___>>>`.

External residues Unstructured single stranded residues completely outside the structure (unenclosed by any base pairs) are annotated by colons, `:`.

Insertions Insertions relative to a known structure are indicated by periods, `.`. Regions where local structural alignment was invoked, leaving regions of both target and query sequence unaligned, are indicated by tildes, `~`. These symbols only appear in alignments of a known (query) structure annotation to a target sequence of unknown structure.

Pseudoknots WUSS notation allows pseudoknots to be annotated as pairs of upper case/lower case letters: for example, `<<<<_AAAA____>>>>aaaa` annotates a simple pseudoknot; additional pseudoknotted stems could be annotated by `Bb`, `Cc`, etc. INFERNAL cannot handle pseudoknots, however; pseudoknot notation never appears in INFERNAL output; it is accepted in input files, but ignored.

An example of WUSS notation for a complicated structure (*E. coli* RNase P) is shown in Figure 5. An example of WUSS notation for a local INFERNAL alignment of *B. subtilis* RNase P to *E. coli* RNase P, illustrating the use of local alignment annotation symbols, is in Figure 6.

Shorthand (input) WUSS notation

While WUSS notation makes it easier to visually interpret INFERNAL *output* structural annotation, it would be painful to be required to *input* all structures in full WUSS notation. Therefore when INFERNAL reads input secondary structure annotation, it uses simpler rules:

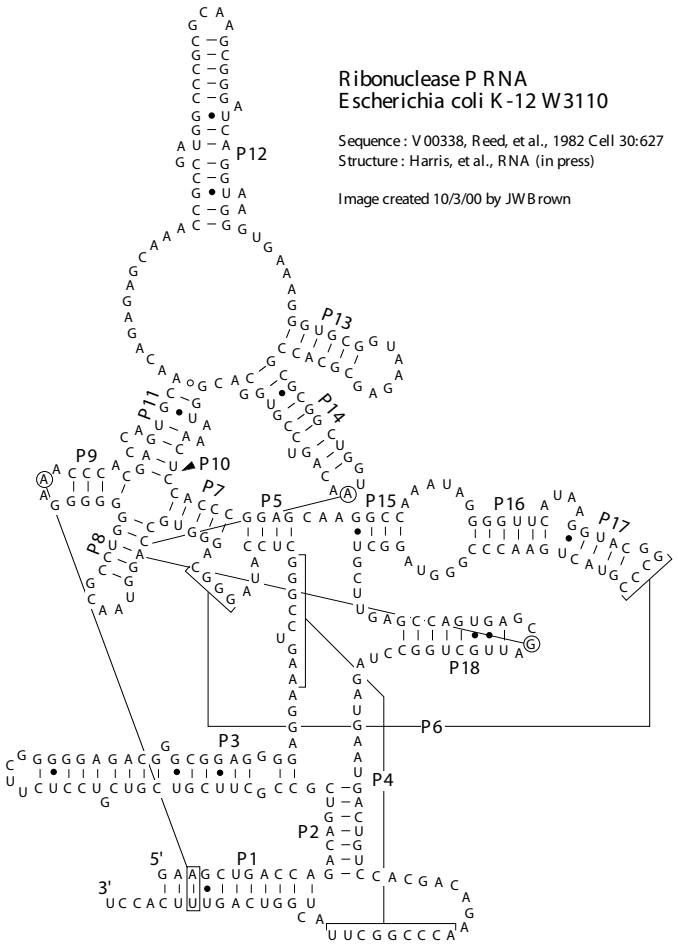
Base pairs Any matching nested pair of `()`, `()`, `[]`, `{}` symbols indicates a base pair; the exact choice of symbol has no meaning, so long as the left and right partners match up.

Single stranded residues All other symbols `_-,:.~` indicate single stranded residues. The choice of symbol has no special meaning. Annotated pseudoknots (nested matched pairs of upper/lower case alphabetic characters) are also interpreted as single stranded residue in INFERNAL input.

Thus, for instance, `<<<<....>>>>` and `((((____))))` and `<(<(._._)>)>` all indicate a four base stem with a four base loop (the last example is legal but weird).

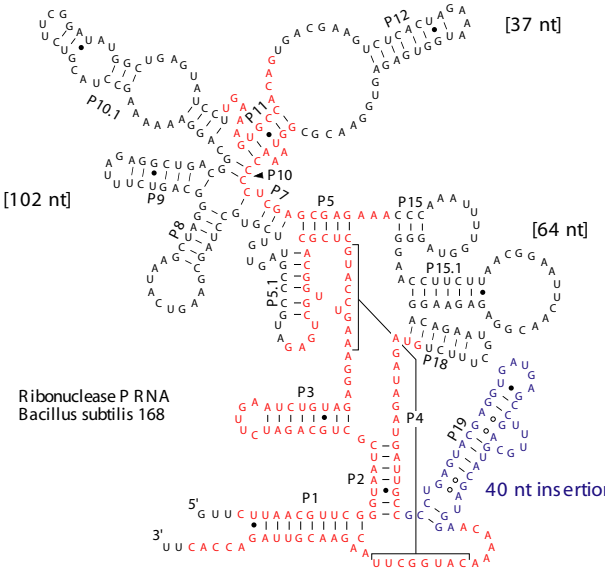
Remember that the key property of canonical (nonpseudoknotted) RNA secondary structure is that the pairs are *nested*. `((<<....))>>` is not a legal annotation string: the pair symbols don't match up properly. INFERNAL will reject such an annotation and report an input format error, suspecting a problem with your annotation. If you want to annotate pseudoknots, WUSS notation allows alphabetic symbols Aa, Bb, etc. see above; but remember that INFERNAL ignores pseudoknotted stems and treats them as single stranded residues.

Because many other RNA secondary structure analysis programs use a simple bracket notation for annotating structure, INFERNAL's ability to input this format makes it easier to use data generated by other RNA software packages. Conversely, converting INFERNAL output WUSS notation to simple bracket notation is a matter of a simple Perl or sed script, substituting the symbols appropriately.




```
    {{{{{{{{{{{{{{{{{{,<<<<<<<<<<<<<-<<<<<_____>>>>>>>>>->>>>>>>>
  1 GAAGCUGACCAGACAGUCGCCGCUUCGUCGUCGUCCUCUUCGGGGGAGACGGGCGGAGGG 60

    >,,,,,,,,,,,,,[[[[--------[[[[[<<<<<_____>>>>>>>>>____>>>->(
 61 GAGGAAAGUCCGGGCUCCAUAGGGCAGGGUGCCAGGUAACGCCUGGGGGGGGAAACCCACG 120

    (---(((((,,,,,,,,,,,,<<<<<--<<<<<<<<_____>>>>>->>>>>>-->>,,,,
121 ACCAGUGCAACAGAGAGCAAACCGCCGAUGGCCCGCGCAAGCGGGAUCAGGUAAGGGUGA 180

    ,,,<<<<<<_______>>>>>>>>>>>>>>>>____>>>->>>>>->,,)))--))))]]]
181 AAGGGUGCGGUAAGAGCGCACCGCGCGGCUGGUAACAGUCCGUGGCACGGUAAACUCCAC 240

    ]]]]]],,,<<<<------<<<<<<----<<<<<_____>>>>>>>>>>>------>>>>,
241 CCGGAGCAAGGCCAAAUAGGGGUUCAUAAGGUACGGCCCGUACUGAACCCGGGUAGGCUG 300

    ,,,,,<<<<<<<<____>>>>>>>>,,,,,,,,,,}}}}}}}----------------
301 CUUGAGCCAGUGAGCGAUUGCUGGCCUAGAUGAAUGACUGUCCACGACAGAACCCGGCUU 360

    -}-}}}}}}}}}}::::
361 AUCGGUCAGUUUCACCU 377
```

Figure 5: **Example of WUSS notation.** Top: Secondary structure of *E. coli* RNase P, from Jim Brown's RNase P database (Brown, 1999). Bottom: WUSS notation for the same structure, annotating the *E. coli* RNase P sequence. The P4 and P6 pseudoknots are not annotated in this example.




```
hit 0   :      4    399    52.56 bits
          {{{{{{{{{{{{{{{{{{,<<<<<<<<<<<<<-<<<<<____>>>>>>>>>->>>>>>>>
           1 ggAGuggGgcaGgCaguCGCugcuucggccuuGuucaguuaacugaaaaggAccgaagga 60
             +: :::G::C:GG:A:UCGCU+C::::            U+            ::::G+A
           4 CUUAACGUUCGGGUAAUCGCUGCAGAUC----------UUG----------AAUCUGUA 42

             >,,,,,,,,,,,,,[[[.[--------[[[[[~~~~~~~((---(((((,,,,~~~~~~)
          61 GAGGAAAGUCCGGGCUC.CACAGGGCAgGGUG*[ 29]*GGAAAGUGCCACAG*[96]*G 229
             GAGGAAAGUCC  GCUC C  A GG   :G G       :GAAAGUGCCACAG      G
          43 GAGGAAAGUCCAUGCUCgC--ACGGUGCUGAG*[102]*UGAAAGUGCCACAG*[37]*G 226

             ))--))))]]]]]].]]],,,~~~~~~,,,,,,,,,}}}}}}}--..............
         230 GUAAACCCCACCcG.GAGCAA*[77]*CuAGAUGAAUGacuGcCCA.............. 344
             GUAAACC:C C: G GAG AA       UAGAU++AUGA:U:CC
         227 GUAAACCCCUCGAGcGAGAAA*[64]*GUAGAUAGAUGAUUGCC--gccugaguacgagg 342

             ........................----------}-}}}}}}}}}}:::::
         345 ........................CGACAGAACCCGGCUUAuagcCccaCUccucuu 377
                                     ACA AAC  GGCUUA:AG::C::: :+ C
         343 ugaugagccguuugcaguacgaugga--ACAAAACAUGGCUUACAGAACGUUAGACCAC 399
```

Figure 6: **Local alignment annotation example.** Top: Secondary structure of *B. subtilis* RNase P, from Jim Brown's RNase P database (Brown, 1999). Residues in red are those that INFERNAL aligns to a CM of *E. coli* type RNase P's. The local structural alignment is in four pieces; three regions of the structure (102, 37, and 64 nt long) are skipped over. One additional stem is treated as a 40 nt insertion. Bottom: the INFERNAL output, showing the *E. coli* query structure aligned to the *B. subtilis* sequence.

Multiple alignments: Stockholm format

The Pfam consortium developed an annotated alignment format called "Stockholm format", and this format has been adopted as the standard alignment format in HMMER and INFERNAL, and by the Rfam consortium. The reasons for inventing a new alignment format were two-fold. First, there really is no standard accepted format for multiple sequence alignment files, so we don't feel guilty about inventing a new one. Second, the formats of popular multiple alignment software (e.g. CLUSTAL, GCG MSF, PHYLIP) do not support rich documentation and markup of the alignment. Stockholm format was developed to support extensible markup of multiple sequence alignments, and we use this capability extensively in both RNA work (with structural markup) and the Pfam database (with extensive use of both annotation and markup).

A minimal Stockholm file

```
# STOCKHOLM 1.0

seq1  ACDEF...GHIKL
seq2  ACDEF...GHIKL
seq3  ...EFMNRGHIKL

seq1  MNPQTVWY
seq2  MNPQTVWY
seq3  MNPQT...
```

The simplest Stockholm file is pretty intuitive, easily generated in a text editor. It is usually easy to convert alignment formats into a "least common denominator" Stockholm format. For instance, SELEX, GCG's MSF format, and the output of the CLUSTAL multiple alignment programs are all similar interleaved formats.

The first line in the file must be `# STOCKHOLM 1.x`, where `x` is a minor version number for the format specification (and which currently has no effect on my parsers, other than identifying the file as Stockholm format). This line allows a parser to instantly identify the file format.

In the alignment, each line contains a name, followed by the aligned sequence. A dash or period denotes a gap. If the alignment is too long to fit on one line, the alignment may be split into multiple blocks, with blocks separated by blank lines. The number of sequences, their order, and their names must be the same in every block. Within a given block, each (sub)sequence (and any associated `#=GR` and `#=GC` markup, see below) is of equal length, called the *block length*. Block lengths may differ from block to block; the block length must be at least one residue, and there is no maximum.

The sequence names must be unique. (They are used to associate markup tags with the sequences.)

Other blank lines are ignored. You can add comments to the file on lines starting with a #.

All other annotation is added using a tag/value comment style. The tag/value format is inherently extensible, and readily made backwards-compatible; unrecognized tags will simply be ignored. Extra annotation includes consensus and individual RNA or protein secondary structure, sequence weights, a reference coordinate system for the columns, and database source information including name, accession number, and coordinates (for subsequences extracted from a longer source sequence) See below for details.

Syntax of Stockholm markup

There are four types of Stockholm markup annotation, for per-file, per-sequence, per-column, and per-residue annotation:

#=GF <tag> <s> Per-file annotation. **<s>** is a free format text line of annotation type **<tag>**. For example, **#=GF DATE April 1, 2000**. Can occur anywhere in the file, but usually all the **#=GF** markups occur in a header.

#=GS <seqname> <tag> <s> Per-sequence annotation. **<s>** is a free format text line of annotation type **tag** associated with the sequence named **<seqname>**. For example, **#=GS seq1 SPECIES_SOURCE Caenorhabditis elegans**. Can occur anywhere in the file, but in single-block formats (e.g. the Pfam distribution) will typically follow on the line after the sequence itself, and in multi-block formats (e.g. HMMER output), will typically occur in the header preceding the alignment but following the **#=GF** annotation.

#=GC <tag> <s> Per-column annotation. **<s>** is an aligned text line of annotation type **<tag>**. #=GC lines are associated with a sequence alignment block; **<s>** is aligned to the residues in the alignment block, and has the same length as the rest of the block. Typically #=GC lines are placed at the end of each block.

#=GR <seqname> <tag> <s> Per-residue annotation. **<s>** is an aligned text line of annotation type **<tag>**, associated with the sequence named **<seqname>**. #=GR lines are associated with one sequence in a sequence alignment block; **<s>** is aligned to the residues in that sequence, and has the same length as the rest of the block. Typically #=GR lines are placed immediately following the aligned sequence they annotate.

Semantics of Stockholm markup

Any Stockholm parser will accept syntactically correct files, but is not obligated to do anything with the markup lines. It is up to the application whether it will attempt to interpret the meaning (the semantics) of the markup in a useful way. At the two extremes are the Belvu alignment viewer and the HMMER profile hidden Markov model software package.

Belvu simply reads Stockholm markup and displays it, without trying to interpret it at all. The tag types (**#=GF**, etc.) are sufficient to tell Belvu how to display the markup: whether it is attached to the whole file, sequences, columns, or residues.

HMMER and INFERNAL use Stockholm markup to pick up a variety of information from the multiple alignment files. The Pfam and Rfam consortiums therefore agree on additional syntax for certain tag types, so software can parse some markups for useful (or necessary) information. This additional syntax is imposed by Pfam, HMMER, INFERNAL, and other software of mine, not by Stockholm format per se. You can think of Stockholm as akin to XML, and what my software reads as akin to an XML DTD, if you're into that sort of structured data format lingo.

The Stockholm markup tags that are parsed semantically by my software are as follows:

Recognized #=GF annotations

ID <s> Identifier. **<s>** is a name for the alignment; e.g. "RNaseP. Mandatory, if the file is an alignment database used as input for **cmbuild**, because each CM must get a unique name. One word. Unique in file.

AC <s> Accession. **<s>** is a unique accession number for the alignment; e.g. "PF00001". Used by the Rfam database, for instance. Often a alphabetical prefix indicating the database (e.g. "RF") followed by a unique numerical accession. One word. Unique in file.

DE <s> Description. **<s>** is a free format line giving a description of the alignment; e.g. "Ribonuclease P RNA". One line. Unique in file.

AU <s> Author. **<s>** is a free format line listing the authors responsible for an alignment; e.g. "Bateman A". One line. Unique in file.

Recognized #=GS annotations

WT <f> Sequence weight. **<f>** is a positive real number giving the relative weight for a sequence, usually used to compensate for biased representation by downweighting similar sequences. Usually the weights average 1.0 (e.g. the weights sum to the number of sequences in the alignment) but this is not required. Either every sequence must have a weight annotated, or none of them can.

AC <s> Accession. **<s>** is a database accession number for this sequence. (Compare the **#=GF AC** markup, which gives an accession for the whole alignment.) One word.

DE <s> Description. **<s>** is one line giving a description for this sequence. (Compare the **#=GF DE** markup, which gives a description for the whole alignment.)

Recognized #=GC annotations

RF Reference line. Any character is accepted as a markup for a column. The intent is to allow labeling the columns with some sort of mark. **cmbuild** uses this annotation to determine which columns are consensus versus insertion; insertion columns are annotated by a gap symbol, and consensus columns by any non-gap symbol.

SS_cons Secondary structure consensus. When this line is generated by INFERNAL, it is generated in full WUSS notation. When it is read by **cmbuild**, it is interpreted more loosely, in shorthand (input) WUSS notation: pairs of symbols <>, `()`, `[]`, or `[]` mark consensus base pairs, and symbols `:_-,.~` mark single stranded columns.

Recognized #=GR annotations

SS Secondary structure for this sequence. See **#=GC SS_cons** above.

Sequence files: FASTA format

FASTA is probably the simplest of formats for unaligned sequences. FASTA files are easily created in a text editor. Each sequence is preceded by a line starting with >. The first word on this line is the name of the sequence. The rest of the line is a description of the sequence (free format). The remaining lines contain the sequence itself. You can put as many letters on a sequence line as you want. For example:

```
>seq1 This is the description of my first sequence.
AGTACGTAGTAGCTGCTGCTACGTGCGCTAGCTAGTACGTCA CGACGTAGATGCTAGCTGACTCGATGC
>seq2 This is a description of my second sequence.
CGATCGATCGTACGTCGACTGATCGTAGCTACGTCGTACGTAG CATCGTCAGTTACTGCATGCTCG
CATCAGGCATGCTGCTGACTGATCGTACG
```

For better or worse, FASTA is not a documented standard. Minor (and major) variants are in widespread use in the bioinformatics community, all of which are called "FASTA format". My software attempts to cater to all of them, and is tolerant of common deviations in FASTA format. Certainly anything that is accepted by the database formatting programs in NCBI BLAST or WU-BLAST (e.g. setdb, pressdb, xdformat) will also be accepted by my software. Blank lines in a FASTA file are ignored, and so are spaces or other gap symbols (dashes, underscores, periods) in a sequence. Other non-amino or non-nucleic acid symbols in the sequence are also silently ignored, mostly because some people seem to think that "*" or "." should be added to protein sequences to (redundantly) indicate the end of the sequence. The parser will also accept unlimited line lengths, which allows it to accomodate the enormous description lines in the NCBI NR databases.

(On the other hand, any FASTA files *generated* by my software adhere closely to community standards, and should be usable by other software packages (BLAST, FASTA, etc.) that are more picky about parsing their input files. That means you can run a sloppy FASTA file thru the **`sreformat`** utility program to clean it up.)

Partly because of this tolerance, the software may have a difficult time dealing with files that are *not* in FASTA format, especially if you're relying on file format autodetection (the "Babelfish"). Some (now mercifully uncommon) file formats are so similar to FASTA format that they be erroneously called FASTA by the Babelfish and then quietly and lethally misparsed. An example is the old NBRF file format. If you're afraid of this, you can use the **`--informat fasta`** option to bypass the Babelfish and improve robustness. However, it is still possible to construct files perversely similar to FASTA that will still confuse the parser. (The gist of these caveats applies to all formats, not just FASTA.)

CM file format

The default CM file format is a simple, extensible tag-value format. The format being used right now is tentative and likely to change. Therefore, it is not currently documented here. If you absolutely need to interpret it, see the file `cmio.c` in the source code.

Dirichlet prior files

A prior file is parsed into a number of whitespace-delimited, non-comment fields. These fields are then interpreted in order. The order and number of the fields is important. This is not a robust, tag-value save file format.

All whitespace is ignored, including newlines. The number of fields per line is unimportant.

Comments begin with a # character. The remainder of any line following a # is ignored.

The Infernal source distribution includes an example prior file, **`default.pri`**. This prior is identical to the hardcoded default prior used by Infernal. The following text may only make sense if you're looking at that example while you read.

The order of the fields in the prior file is as follows:

Strategy. The first field is the keyword **`Dirichlet`**. Currently Dirichlet priors (mixture or not) are the only prior strategy used by Infernal.

Transition prior section. The next field is the number **`74`**, the number of different types of transition distributions. (See Figure 7 for an explanation of where the number 74 comes from.) Then, for each of these 74 distributions:

<from-uniqstate> <to-node>: Two fields give the transition type: from a unique state identifier, to a node identifier. Example: **MATP_MP MATP**.

<n>: One field gives the number of transition probabilities for this transition type; that is, the number of Dirichlet parameter vector $\alpha_1^q..\alpha_n^q$ for each mixture component q.

<nq>: One field gives the number of mixture Dirichlet components for this transition type's prior. Then, for each of these **nq** Dirichlet components:

p(q): One field gives the mixture coefficient $p(q)$, the prior probability of this component q. For a single-component "mixture", this is always 1.0.

$\alpha_{\mathbf{1}}^{\mathbf{q}}..\alpha_{\mathbf{n}}^{\mathbf{q}}$: The next n fields give the Dirichlet parameter vector for this mixture component q.

Base pair emission prior section. This next section is the prior for MATP_MP emissions. One field gives **<K>**, the "alphabet size" – the number of base pair emission probabilities – which is always 16 (4x4), for RNA. The next field gives **<nq>**, the number of mixture components. Then, for each of these **nq** Dirichlet components:

p(q): One field gives the mixture coefficient $p(q)$, the prior probability of this component q. For a single-component "mixture", this is always 1.0.

$\alpha_{\mathbf{AA}}^{\mathbf{q}}..\alpha_{\mathbf{UU}}^{\mathbf{q}}$: The next 16 fields give the Dirichlet parameter vector for this mixture component, in alphabetical order (AA, AC, AG, AU, CA ... GU, UA, UC, UG, UU).

Consensus singlet base emission prior section. This next section is the prior for MATL_ML and MATR_MR emissions. One field gives **<K>**, the "alphabet size" – the number of singlet emission probabilities – which is always 4, for RNA. The next field gives **<nq>**, the number of mixture components. Then, for each of these **nq** Dirichlet components:

p(q): One field gives the mixture coefficient $p(q)$, the prior probability of this component q. For a single-component "mixture", this is always 1.0.

$\alpha_{\mathbf{A}}^{\mathbf{q}}..\alpha_{\mathbf{U}}^{\mathbf{q}}$: The next 4 fields give the Dirichlet parameter vector for this mixture component, in alphabetical order (A, C, G, U).

Nonconsensus singlet base emission prior section. This next section is the prior for insertions (MATP_IL, MATP_IR, MATL_IL, MATR_IR, ROOT_IL, ROOT_IR, BEGR_IL) as well as nonconsensus singlets (MATP_ML, MATP_MR). One field gives **<K>**, the "alphabet size" – the number of singlet emission probabilities – which is always 4, for RNA. The next field gives **<nq>**, the number of mixture components. Then, for each of these **nq** Dirichlet components:

p(q): One field gives the mixture coefficient $p(q)$, the prior probability of this component q. For a single-component "mixture", this is always 1.0.

$\alpha_{\mathbf{A}}^{\mathbf{q}}..\alpha_{\mathbf{U}}^{\mathbf{q}}$: The next 4 fields give the Dirichlet parameter vector for this mixture component, in alphabetical order (A, C, G, U).

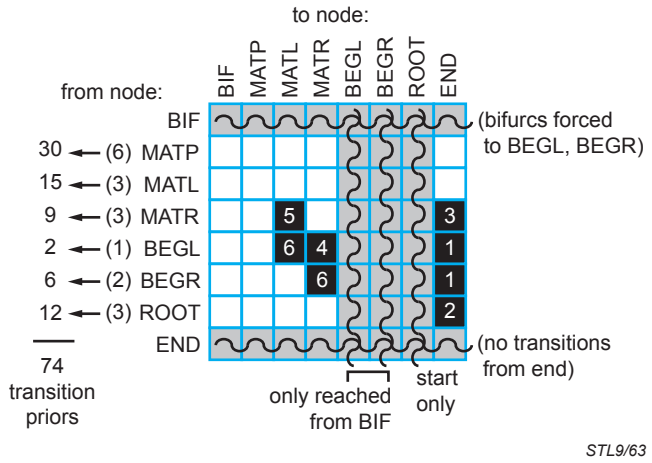


Figure 7: **Where does the magic number of 74 transition distribution types come from?** The transition distributions are indexed in a 2D array, from a unique statetype (20 possible) to a downstream node (8 possible), so the total conceivable number of different distributions is $20 \times 8 = 160$. The grid represents these possibilities by showing the 8×8 array of all node types to all node types; each starting node contains 1 or more unique states (number in parentheses to the left). Two rows are impossible (gray): bifurcations automatically transit to determined BEGL, BEGR states with probability 1, and end nodes have no transitions. Three columns are impossible (gray): BEGL and BEGR can only be reached by probability 1 transitions from a bifurcation, and the ROOT node is special and can only start a model. Eight individual cells of the grid are unused (black) because of the way **`cmbuild`** (almost) unambiguously constructs a guide tree from a consensus structure. These cases are numbered as follows. (1) BEGL and BEGR never transit to END; this would imply an empty substructure. A bifurcation is only used if both sides of the split contain at least one consensus pair (MATP). (2) ROOT never transits to END; this would imply an alignment with zero consensus columns. Infernal models assume ≥ 1 consensus columns. (3) MATR never transits to END. Infernal always uses MATL for unpaired columns whenever possible. MATR is only used for internal loops, multifurcation loops, and 3' bulges, so MATR must always be followed by a BIF, MATP, or another MATR. (4) BEGL never transits to MATR. The single stranded region between two bifurcated stems is unambiguously assigned to MATL nodes on the right side of the split, not to MATR nodes on the left. (5) MATR never transits to MATL. The only place where this could arise (given that we already specified that MATL is used whenever possible) is in an interior loop; there, by unambiguous convention, MATL nodes precede MATR nodes. (6) BEGL nodes never transit to MATL, and BEGR nodes never transit to MATR. By convention, at any bifurcated subsequence i, j, i and j are paired but not to each other. That is, the smallest possible subsequence is bifurcated, so that any single stranded stretches to the left and right are assigned to MATL and MATR nodes above the bifurcation, instead of MATL nodes below the BEGL and MATR nodes below the BEGR. Thus, the total number 74 comes from multiplying, for each row, the number of unique states in each starting node by the number of possible downstream nodes (white), and summing these up, as shown to the left of the grid.

6 Manual pages

`cmalign` - use a CM to make a structure RNA multiple alignment

Synopsis

cmalign *[options] cmfile seqfile*

Description

cmalign aligns the RNA sequences in *seqfile* to the covariance model (CM) in *cmfile*, and outputs a multiple sequence alignment.

The sequence file can be in most any common biosequence format, including alignment file formats (in which case the sequences will be read as if they were unaligned). FASTA format is recommended.

CM files are profiles of RNA consensus secondary structure. A CM file is produced by the **cmbuild** program, from a given RNA sequence alignment of known consensus structure.

The alignment that **cmbuild** makes is written in Stockholm format. It can be redirected to a file using the *-o* option.

Options

-h Print brief help; includes version number and summary of all options, including expert options.

-l Turn on the local alignment algorithm, which allows the alignment to span two or more subsequences if necessary (e.g. if the structures of the query model and target sequence are only partially shared), allowing certain large insertions and deletions in the structure to be penalized differently than normal indels. The default is to globally align the query model to the target sequences.

-o <*f*> Save the alignment in Stockholm format to a file <*f*>. The default is to write it to standard output.

-q Quiet; suppress the verbose banner, and only print the resulting alignment to stdout. This allows piping the alignment to the input of other programs, for example.

Expert Options

--informat <*s*> Assert that the input *seqfile* is in format <*s*>. Do not run Babelfish format autodection. This increases the reliability of the program somewhat, because the Babelfish can make mistakes; particularly recommended for unattended, high-throughput runs of Infernal. <*s*> is case-insensitive; valid formats include FASTA, GENBANK, EMBL, GCG, PIR, STOCKHOLM, SELEX, MSF, CLUSTAL, and PHYLIP. See the User's Guide for a complete list.

--nosmall Use the normal CYK alignment algorithm. The default is to use the divide and conquer algorithm described in SR Eddy, BMC Bioinformatics 3:18, 2002. This is useful for debugging, and checking that the two algorithms give identical results. The ”normal” algorithm requires too much memory for most uses.

--regress <*f*> Save regression test information to a file <*f*>. This is part of the automated testing procedure at each release.

`cmbuild` - construct a CM from an RNA multiple sequence alignment

Synopsis

cmbuild *[options] cmfile alifile*

Description

cmbuild reads an RNA multiple sequence alignment from *alifile,* constructs a covariance model (CM), and saves the CM to *cmfile.*

The alignment file must be in Stockholm format, and must contain consensus secondary structure annotation. **cmbuild** uses the consensus structure to determine the architecture of the CM.

The alignment file may be a database containing more than one alignment. If it does, the resulting *cmfile* will be a database of CMs, one per alignment. In this case, each alignment must have a name (a #=GF ID tag, in Stockholm format).

Options

-h Print brief help; includes version number and summary of all options, including expert options.

-n <*s*> Name the covariance model <*s*>. (Does not work if *alifile* contains more than one alignment.) The default is to use the name of the alignment (given by the #=GF ID tag, in Stockholm format), or if that is not present, to use the name of the alignment file minus any file type extension (that is, a file ”myrnas.sto” would give a CM named ”myrnas”).

-A Append the CM to *cmfile,* if *cmfile* already exists.

-F Allow *cmfile* to be overwritten. Normally, if *cmfile* already exists, **cmbuild** exits with an error unless the *-A* or *-F* option is set.

Expert Options

--binary Save the model in a compact binary format. The default is a more readable ASCII text format.

--rf Use reference coordinate annotation (#=GC RF line, in Stockholm) to determine which columns are consensus, and which are inserts. Any non-gap character indicates a consensus column. (For example, mark consensus columns with ”x”, and insert columns with ”.”.) The default is to determine this automatically; if the frequency of gap characters in a column is greater than a threshold, gapthresh (default 0.5), the column is called an insertion.

--gapthresh <*x*> Set the gap threshold (used for determining which columns are insertions versus consensus; see above) to <*x*>. The default is 0.5.

--informat <*s*> Assert that the input *alifile* is in format <*s*>. Do not run Babelfish format autodection. This increases the reliability of the program somewhat, because the Babelfish can make mistakes; particularly recommended for unattended, high-throughput runs of Infernal. <*s*> is case-insensitive. This option is a bit forward-looking; **cmbuild** currently only accepts Stockholm format, but this may not be true in the future.

--wgiven Use sequence weights as given in annotation in the input alignment file. If no weights were given, assume they are all 1.0. The default is to determine new sequence weights by the Gerstein/Sonnhammer/Chothia algorithm, ignoring any annotated weights.

--wnone Turn sequence weighting off; e.g. explicitly set all sequence weights to 1.0.

--wgsc Use the Gerstein/Sonnhammer/Chothia weighting algorithm. This is the default, so this option is probably useless.

--cfile <*f*> Save a file containing observed count vectors (both emissions and transitions) to a counts file <*f*>. One use for this file is as the starting point for estimating Dirichlet priors from observed RNA structure data.

--cmtbl <*f*> Save a tabular description of the CM's topology to a file <*f*>. Primarily useful for debugging CM architecture construction.

--emap <*f*> Save a consensus emission map to a file <*f*>. This file relates the numbering system of states in the CM's tree-like directed graph to the linear numbering of consensus columns. Primarily useful for debugging.

--gtree <*f*> Save an ASCII picture of the high level structure of the CM's guide tree to a file <*f*>. Primarily useful for debugging.

--gtbl <*f*> Save a tabular description of the nodes in CM's guide tree to a file <*f*>. Primarily useful for debugging.

--tfile <*f*> Dump tabular inferred sequence tracebacks for each individual training sequence to a file <*f*>. Primarily useful for debugging.

--nobalance Turn off the architecture "rebalancing" algorithm. The nodes in a CM are initially numbered in standard preorder traversal. The rebalancing algorithm is an optimizer that reorders the numbering of the CM in order to absolutely guarantee certain algorithmic performance bounds. However, it is a stylistic riff that has almost no real empirical impact on performance, and is a tricky algorithm to get right. This option was inserted for debugging purposes. It is sometimes also useful to obtain a simple preorder traversal numbering system in the CM architecture (for illustrative purposes, for example).

--regress <*f*> Save regression test information to a file <*f*>. This is part of the automated testing procedure at each release.

--treeforce After building the model, score the first sequence in the alignment using its inferred parsetree, and show both the score and the parsetree. This is a debugging tool, used to specify and score a particular desired parsetree.

`cmscore` - align and score one or more sequences to a CM

Synopsis

cmscore *[options] cmfile seqfile*

Description

cmscore uses the covariance model (CM) in *cmfile* to align and score the sequences in *seqfile*, and output the score and optimal alignment for each one. **cmscore** is a testbed for new CM alignment algorithms, and it is also used by the testsuite. It is not intended to be particularly useful in the real world. Documentation is provided for completeness, and to aid my own memory.

Currently, **cmscore** aligns the sequence(s) by both the full CYK algorithm and the divide and conquer variant (SR Eddy, BMC Bioinformatics 3:18, 2002), and outputs both parse trees.

Usually, the two parse trees should be identical for any sequence. However, there can be cases of ties, where two or more different parse trees have identical scores. In such cases, it is possible for the two parse trees to differ. The parse tree selected as ”optimal” from amongst the ties is arbitrary, dependent on order of evaluation in the DP traceback, and the order of evaluation for D&C vs. standard CYK is different. Thus, in its testsuite role, **cmscore** checks that the scores are within 0.01 bits of each other, but does not check that the parse trees are absolutely identical; identity can be checked for using the **--stringent** option.

The sequence file can be in most any common biosequence format, including alignment file formats (in which case the sequences will be read as if they were unaligned). FASTA format is recommended.

The sequences are treated as single stranded RNAs; that is, only the given strand of each sequence is aligned and scored, and no reverse complementing is done.

CM files are profiles of RNA consensus secondary structure. A CM file is produced by the **cmbuild** program, from a given RNA sequence alignment of known consensus structure.

Options

-h Print brief help; includes version number and summary of all options, including expert options.

Expert Options

--informat <*s*> Assert that the input *seqfile* is in format <*s*>. Do not run Babelfish format autodection. This increases the reliability of the program somewhat, because the Babelfish can make mistakes; particularly recommended for unattended, high-throughput runs of Infernal. <*s*> is case-insensitive; valid formats include FASTA, GENBANK, EMBL, GCG, PIR, STOCKHOLM, SELEX, MSF, CLUSTAL, and PHYLIP. See the User’s Guide for a complete list.

--local Turn on the local alignment algorithm, which allows the alignment to span two or more subsequences if necessary (e.g. if the structures of the query model and target sequence are only partially shared), allowing certain large insertions and deletions in the structure to be penalized differently than normal indels. The default is to globally align the query model to the target sequences.

--regress <*f*> Save regression test information to a file <*f*>. This is part of the automated testing procedure at each release.

--scoreonly Do the small memory ”score only” variant of the standard CYK alignment algorithm, and don’t recover a parse tree.

--smallonly Skip the standard CYK algorithm; do only the divide and conquer algorithm.

--stringent Require the two parse trees to be identical; fail and return a non-zero exit code if they are not. Normally, **cmscore** only requires that the two parse trees have identical scores (within a floating point tolerance of 0.01 bits), because it is possible to have more than one parse tree with the same score.

--X Project X. Undocumented. No serviceable parts inside. Using this option voids your warranty. Do not attempt. Professional driver on a closed course. May induce dizziness and vomiting.

`cmsearch` - search a sequence database for RNAs homologous to a CM

Synopsis

cmsearch *[options] cmfile seqfile*

Description

cmsearch uses the covariance model (CM) in *cmfile* to search for homologous RNAs in *seqfile,* and outputs high-scoring alignments.

The sequence file can be in most any common biosequence format, including alignment file formats (in which case the sequences will be read as if they were unaligned). FASTA format is recommended.

CM files are profiles of RNA consensus secondary structure. A CM file is produced by the **cmbuild** program, from a given RNA sequence alignment of known consensus structure.

The output that **cmsearch** produces is currently extremely rudimentary. All hits of score greater than zero bits are output as alignments, in the order they are found. Niceties like ranking hits by their score, E-values, and reporting thresholds will come later.

Options

-h Print brief help; includes version number and summary of all options, including expert options.

-W <*n*> Set the scanning window width to <*n*>. This is the maximum length of a homologous sequence. By default, this is set to 200, which will be too small for many RNAs. In the future, this number will be automatically set to something sensible, instead of relying on you setting it sensibly on the command line.

Expert Options

--informat <*s*> Assert that the input *seqfile* is in format <*s*>. Do not run Babelfish format autodection. This increases the reliability of the program somewhat, because the Babelfish can make mistakes; particularly recommended for unattended, high-throughput runs of Infernal. <*s*> is case-insensitive; valid formats include FASTA, GENBANK, EMBL, GCG, PIR, STOCKHOLM, SELEX, MSF, CLUSTAL, and PHYLIP. See the User's Guide for a complete list.

--toponly Only search the top (Watson) strand of the sequences in *seqfile.* By default, both strands are searched.

--local Turn on the local alignment algorithm, which allows the alignment to span two or more subsequences if necessary (e.g. if the structures of the query model and target sequence are only partially shared), allowing certain large insertions and deletions

in the structure to be penalized differently than normal indels. The default is to globally align the query model to the target sequences.

--dumptrees Dump verbose, ugly parse trees for each hit. Useful only for debugging purposes.

References

Brown, J. W. (1999). The ribonuclease P database. *Nucl. Acids Res.*, 27:314.

Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. J. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge UK.

Eddy, S. R. (1998). Profile hidden Markov models. *Bioinformatics*, 14:755–763.

Eddy, S. R. (2002). A memory-efficient dynamic programming algorithm for optimal alignment of a sequence to an RNA secondary structure. *BMC Bioinformatics*, 3:18.

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

Gerstein, M., Sonnhammer, E. L. L., and Chothia, C. (1994). Volume changes in protein evolution. *J. Mol. Biol.*, 235:1067–1078.

Giegerich, R. (2000). Explaining and controlling ambiguity in dynamic programming. In Giancarlo, R. and Sankoff, D., editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848, pages 46–59, Montréal, Canada. Springer-Verlag, Berlin.

Griffiths-Jones, S., Bateman, A., Marshall, M., Khanna, A., and Eddy, S. R. (2003). Rfam: an RNA family database. *Nucl. Acids Res.*, 31:439–441.

Henikoff, S. and Henikoff, J. G. (1994). Position-based sequence weights. *J. Mol. Biol.*, 243:574–578.

Karplus, K., Barrett, C., and Hughey, R. (1998). Hidden Markov models for detecting remote protein homologies. *Bioinformatics*, 14:846–856.

Krogh, A., Brown, M., Mian, I. S., Sjolander, K., and Haussler, D. (1994). Hidden Markov models in computational biology: Applications to protein modeling. *J. Mol. Biol.*, 235:1501–1531.

Nawrocki, E. P. and Eddy, S. R. (2007). Query-dependent banding (QDB) for faster RNA similarity searches. Manuscript submitted.