The `mmsearch` Metamotif Search Engine Manual and Reference

Thomas Junier
Swiss Institute of Bioinformatics

April 3, 2001

Abstract

Metamotifs are a tool for describing the arrangement of features along sequences. This document describes the metamotif search engine, `mmsearch`, a Python program which allows to retrieve from a database all sequences that match a given metamotif.

Contents

1 Overview

The *arrangement* of features along a sequence, such as domains along a protein or a DNA stretch, is often more characteristic than the presence of any single feature. For example, protein kinase domains occur in a large number of proteins; so do sterile alpha domains; but only ephrin receptors have a protein kinase domain followed by a sterile alpha. In the DNA world, promoters exhibit similar properties.[1] Searching for arrangements may hence enhance the selectivity of searches without lowering their sensitivity, or vice-versa.

The `mmsearch` program works with arrangement descriptions called *metamotifs* (because they are, in a sense, motifs of motifs). Metamotifs look somewhat like character-oriented regular expressions (often called "patterns" in sequence analysis, see for example the PROSITE patterns[2]), but instead of describing arrangements of characters in a text, they describe arrangements of features in a sequence. The syntax of metamotifs (see section 4) is deliberately similar to that of usual regexps (e.g., those of Perl), but due to specialization to a biological problem, there are significant differences. The `mmsearch` engine also works differently from a pure regular expression engine, because it performs some tasks (like comparing numbers extracted from its input string) that are beyond pattern matching.

`mmsearch` does not in itself look for the occurrence of motifs in the sequences (a.k.a. *match data*): this task is left to specialized predictors like profiles, HMMs, patterns, and the like. The match data is supplied to `mmsearch` on standard input, in a tab-separated format (several formats are possible). This may seem a drawback, but in fact it allows `mmsearch` to freely mix match data of any origin, including, for example, database annotations.

2 Command-line Syntax

The call syntax of `mmsearch` is:

```
$ mmsearch [-hOXv][-i <input_format>][-o <output_format>]
    [-n <name>] <metamotif>
```

[1] In promoters, the motifs would be transcription factor binding sites, CpG islands, TATA-boxes and the like; but at other levels it could include exons, genes, terminators, cistrons, etc.

[2] `http://www.expasy.ch/tools/scnpsit3.html`

The match data are read on `stdin`. The options are:

`-h` Prints out a help message.

`-O` Allow matches to overlap. By default, only reports disjoint matches.

`-X` Take into account all features in the input, including those that are not part of the metamotif. Consider a protein which has domains `A`, `B`, and `A`, in that order. Now suppose you're interested in all proteins who have two `A` domains. The corresponding metamotif (see section 4) is '`A = A`', and domain `B` isn't part of this expression. If match data about `B` is present in the string representation (see section 3.2), '`A = A`' will not match (whereas '`A = B = A`' would). This behaviour is generally not desired, so it is off by default. Use `-X` to turn it back on.

`-v` Verbose. Prints out information as to what operation the program is currently performing.

`-i` <*input_format*> Argument is a string that specifies the ordinal number of each field in a line of input. Default is "`s1b2e3f4`", which means sequence (`s`) is field 1, begin of match (`b`) field 2, end of match (`e`) field 3, and feature (`f`) field 4. The letters can appear in any order, but must all be present. Fields begin at 1, like in cut(1). Some frequently used formats have aliases, these are PFF (the default) and GFF (see B.1), so you could say `-i gff` if you don't remember the fields of GFF.

`-o` *<output_format>* Specifies the output format. The *<output_format>* is a white-separated string of format options (see below). The options have a 1-letter and a 3-letter form, and any combination of options can be specified:[3]

`nat` | `n` ("native") the string representation of the sequence (see 3.2). This is reported once per matching sequence, irrespective of the number of times the meta-motif matches in the sequence. Example (this is a single line):

```
## sw:VAV_HUMAN 617-<prf:SH3#1-617 660-prf:SH3>#1-660
671-<prf:SH2#2-671 765-prf:SH2>#2-765 782-<prf:SH3#3-782
842-prf:SH3>#3-842
```

`pff` | `p` each match is presented on a different line, as PFF(see B.2), but the individual component motifs of the match are not shown. Example:

```
sw:VAV_HUMAN 601 660 METAMOTIF - - -
sw:VAV_HUMAN 782 842 METAMOTIF - - -
```

[3]In the examples below, metamotif `[!pfam:SH3|prf:SH3]` was searched in the human Vav oncogene, `VAV_HUMAN`. The names '`pfam:SH3`' and '`prf:SH3`' represents a match of a Src Homology 3 domain motif according to Pfam and PROSITE, respectively.

This option allows the output of a metamotif search to be fed back to `mmsearch` for another metamotif search. This allows the construction of rather complex queries in a single pipeline (or of even more complex ones using scripts). See A for examples of this.

`det | d` ("detailed") each component motif of each match is shown on a separate line, in the original format.

```
sw:VAV_HUMAN 601 658 pfam:SH3 1 57 10.423
sw:VAV_HUMAN 617 660 prf:SH3 20 -1 11.796
sw:VAV_HUMAN 782 842 prf:SH3 1 -1 17.560
sw:VAV_HUMAN 785 840 pfam:SH3 1 57 18.215
```

Obviously this option is most useful when in conjunction with option `pff`.

`spc | s` ("spacers") the spacers between the component motifs of each match are shown as PFF, each one on a line of its own.

```
sw:VAV_HUMAN 602 616 SPACER 1 -1 -
sw:VAV_HUMAN 618 657 SPACER 1 -1 -
sw:VAV_HUMAN 659 659 SPACER 1 -1 -
...
```

This information can be handy when there is suspicion that there is an uncharacterized but conserved region that frequently occurs between, say, two motifs A and B. The spacer sequences could be extracted, aligned, and made into a profile. Again, this isn't very useful without the `det` option.

`sep | h` ("hash marks") lines in native format (option '`nat`' or '`n`') are preceded by "`## `", and different matches in the same protein are separated by a line containing only '`#`'. This makes it easier for parsers to group PFF lines my match, and matches by sequence; while still allowing the output to be piped to, say, GNUplot ('`#`' is a comment, and such lines are ignored).

The default string is '`nat pff det sep`', which results in the following output:

```
## sw:VAV_HUMAN 601-<pfam:SH3#3-601 617-<prf:SH3#1-617
658-pfam:SH3>#3-658
660-prf:SH3>#1-660 782-<prf:SH3#2-782 785-<pfam:SH3#4-785
840-pfam:SH3>#4-840
842-prf:SH3>#2-842
sw:VAV_HUMAN 601 660 METAMOTIF - - -
sw:VAV_HUMAN 601 658 pfam:SH3 1 57 10.423
sw:VAV_HUMAN 617 660 prf:SH3 20 -1 11.796
#
sw:VAV_HUMAN 782 842 METAMOTIF - - -
sw:VAV_HUMAN 782 842 prf:SH3 1 -1 17.560
```

```
sw:VAV_HUMAN 785 840 pfam:SH3 1 57 18.215
#
```

`-n` <*name*> With option `pff`, use <*name*> as feature name (default is '`METAMOTIF`').

3 Implementation

3.1 The Steps of a Metamotif Search

Here's the pseudocode for a metamotif search:

```
1:  get match data about relevant sequences and features -> list
2:  scan metamotif regexp -> tokens
3:  parse tokens -> automaton
4:  for each sequence in list:
5:      convert sequence to string representation
6:      search for regexp over string using automaton
```

Here's what each of these steps does in more detail:

1. **Get match data** The match data are usually looked up in a database (such as Hits[4] [4]) or generated by running the appropriate program(s) (like `pfsearch` [2] or `hmmer`[5]). In any case, one ends up with a list of match data, *i.e.* which motif(s) are present in which sequence(s), and at which position(s). This should be formatted in a line-oriented, tab-separated format with information about one match per line, e.g.

   ```
   sw:VAV_HUMAN 402 504 prf:PH_DOMAIN 1 -1 10.759
   ```

   In this particular format (PFF – see B.2 for details), the first four fields are sequence ID, start of match, end of match, and motif ID. These are the fields needed by `mmsearch`, the others aren't used. Other popular formats, like GFF (see B.1), are also supported. Whatever the origin and format of the data, they are supplied to `mmsearch` on standard input.

2. **Scan metamotif** The user-supplied metamotif is scanned and separated into elementary tokens (smallest meaningful units) like '`(`', '`,`', or '`<`'.

3. **Parse tokens** The tokens are parsed into a meaningful structure composed of substructures, according to the grammar (see 5. For each substructure, a partial automaton is built, and when all the tokens have been successfully parsed, the full automaton is generated.

4. **Loop over all sequences** For each sequence in the list obtained in step 1, do the two following steps:

[4] `http://hits.isb-sib.ch`
[5] `http://hmmer.wustl.edu`

5. **Represent the sequence as a string** The match data is converted into a string which contains the name and position of all motifs, in order of position, (see section 3.2 for the syntax).

6. **Run automaton** Pass this string as input to the automaton built in step 3. When the automaton ends in an acceptor state, report a match. By default, the search resumes from the end of the present match and all matches are reported. This behaviour may be changed by the user: option `-O` (see 2) allows matches to overlap, *i.e.* the search resumes at the next position after the *start* of the match ; the start-anchor '^' causes only matches that occur at the N-terminus of the sequence to be reported; and the end-anchor '$' has the same effect but at the C-terminus of the sequence.

3.2 The String Representation

Metamotif searching is (partially) a pattern matching problem, so the match data pertaining to a sequence must first be converted into some form of string. Here's an example of such a string:

```
181-<SH2#1-181 256-SH2>#1-256
```
(1)

This means that there is an SH2 domain that spans residues 181 to 256. All such strings are made of concatenations of *feature end data*, which are separated by spaces. Expression 1 contains two feature end data, namely

```
181-<SH2#1-181
```
(2)

and

```
256-SH2>#1-256
```
(3)

Expression 2 is composed of three parts, separated by dashes ('-'):

```
181
```
(4)

```
<SH2#1
```
(5)

and

```
181
```
(6)

Expressions 4 and 6, which are identical, are the position of the feature end. It is repeated because there may be a spacer (see 4.4) both before and after a feature end (exception: no spacer at the beginning or end of the metamotif, see 5), and because each character of the input string is only read once. Expression 5 identifies the feature end. It is composed of two parts:

```
<SH2
```
(7)

and

```
#1
```
(8)

Part 7 is the kind of feature end (in this case, the start of an `SH2` domain), and 8 the feature's number, which is unique and is used for identification purposes.

The feature end data appear in order of position. This may lead to ambiguities when two or more features start (resp. end) at the same position. In this case, the longer feature appears *first* (resp. last) in the string. All in all, expression 1 states that the sequence contains the start ('<') of domain `SH2` # 1 at position 181, and the end ('>') of domain `SH2` # 1 (*i.e.*, the same SH2 domain) at position 256.

Note: To speed things up, the whole names of motifs are not used; instead, they are converted to one-letter symbols, e.g. `SH2` ↔ `a`, etc.

This representation is also used in `mmsearch`'s native output (option `nat`, see 2).

3.3 Program Structure

The engine consists of the following Python files:

`mmState.py` Class hierarchy of automaton states: class `State` and its subclasses. All states are linked to one or more next states, and inherit (and possibly override) a method for deciding to which of them (if any) a transition is possible. They also have a '`status`' attribute which is either '`accept`' or '`reject`' and specifies whether a match has been found when no further transition is possible. Some of these states behave like classic finite-state automaton (FSA) states (for more about FSAs see [1], section 3.6), others perform differently. The `Spacer` state, for example, reads from the input until it finds two series of digits, which it then converts to integers; it then allows or rejects transition based on the difference between these two numbers (which represent the end position of a motif and the start position of the next one).

`mmAutomaton.py` This file provides automata, which are built from `State` objects (and subclasses). The main method of an `Automaton` is `run()`, which launches the match process (*cf.* 3.1, pt. 6). The abstract class `Automaton` provides this functionality. Then there are minimal automata for `State` and its subclasses. This is because a single state does not constitute an automaton: there must be at least a start state and an end state. Finally, there are constructors that take a list of automata and link them together (retaining order) into a larger automaton. This process of linking together smaller automata into a larger one is done by the parser, `mmParse`.

`mmScan.py` Metamotif lexer. This splits the metamotif (as supplied by the user) into *tokens*, which are its smallest meaningful units. This corresponds to point 2 in section 3.1.

`mmParse.py` Metamotif parser: a recursive-descent parser that builds the automaton from the elementary sub-automata defined in `mmAutomaton`, ac-

cording to the tokens supplied by `mmScan` (*cf.* 3.1, pt. 3). For the grammar it implements, see section 5. For more about parsing, see [3][6].

`mmsearch` Executable script. This takes care of handling the command-line options, sorting the input data by sequence, building the motif-name ↔ letter conversion tables, converting the match data into string representation (see 3.2), etc. This program performs points 1, 4 and 5 of section 3.1, and drives `mmScan.py`, `mmParse.py`, and `mmAutomaton.py` to perform steps 2, 3 and 6, respectively.

4 Metamotif Syntax

This section describes all the elements of the metamotif syntax.

4.1 Features

A feature is denoted simply by its name, which may include alphanumeric characters (case is significant) as well as ':' and '_'.

```
SH2
PKINASE
```

e.t.c.

4.2 Simple Arrangements and Separators

When two features (or feature ends) directly follow one another, separate them with a *separator*, '␣=␣' (space, 'equal', space). The spaces are in fact optional, but I feel they enhance readability.

`SH2 = SH3` → SH2 followed by SH3

See also note 1 in section 5.

4.3 Feature Ends

It is sometimes necessary to specify feature *ends* rather than whole features, for example when dealing with overlaps or inclusions. The start of a feature is indicated by a '<' preceding the feature, its end by a '>' following the feature:

`<SH2` → start of SH2,
`SH2>` → end of SH2,

In fact, `mmsearch` only deals with feature ends. A "whole" feature, *i.e.* one representing the total extent of the feature, is silently converted to two corresponding ends, e.g. if you say

[6]This book is currently out of print, but the whole text is available from `http://www.cs.vu.nl/~dick/PTAPG.html`

```
SH2
```

the program will convert it to

```
<SH2 = SH2>
```

4.4 Spacers

When the number of residues that separate feature ends (or features) is important, specify the range of acceptable values with a *spacer*: '*m*,*n*'.

```
53EXO_N_DOMAIN = 4,11 = 53EXO_I_DOMAIN
```

This means "from 4 to 11 residues between the Exo-N and the Exo-I domains", and is typical of eubacterial DNA polymerases. Spacers are separated from feature ends by a separator (4.2). They can be open-ended, e.g. ',20' means "at most 20 residues" while '600,' specifies at least 600. Spacers must always be preceded and followed by a feature or feature end.

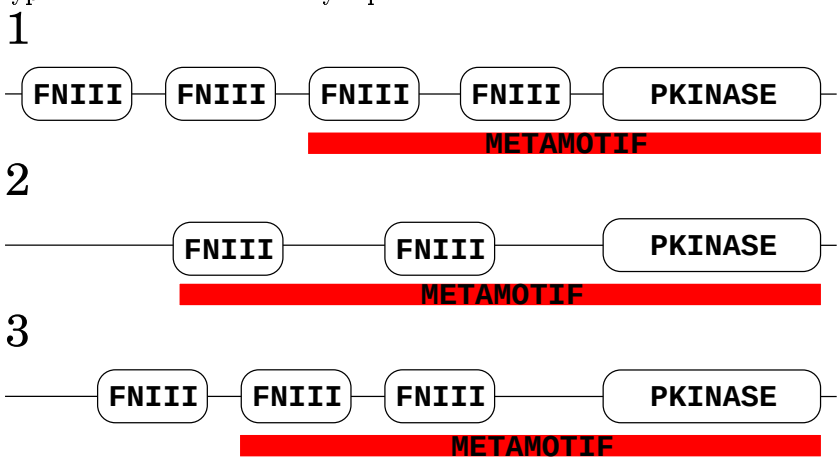
4.5 Anchors

It can be requested that a metamotif occur at the beginning or end of the string. This is done with the usual regexp characters, '^' (start) and '$' (end). Anchoring at the start of the string can speed up the search appreciably, because if the pattern does not match immediately, `mmsearch` does not try to match at other positions in the sequence. See also note 2 in section 5.

Anchors may sometimes 'correct' unexpected (but nevertheless correct) behaviour. For example, say you wish to find this arrangement:

```
FNIII = FNIII = PKINASE
```

Running this will yield all manner of proteins with at least two fibronectin type-III domains followed by a protein kinase:



In this case, what the user wanted was probably only sequence #2. What has gone wrong? Nothing, in fact. It's just that in cases #1 and #3, the match does not start at the beginning of the sequence (the match is indicated by a dashed line). To specify that the match must begin at the start of the sequence, say

```
^FNIII = FNIII = PKINASE
```

The '$' anchor works much in the same way, but restricts the match to the end of the sequence. Thus, if you wished to look for sequences that contain exactly the above pattern, nothing before, nothing after, you would say:

```
^FNIII = FNIII = PKINASE$
```

4.6 Alternatives (logical 'xor')

When there are several mutually exclusive possibilities (or 'branches'), separate them by an *alternative*: '`(`*b1*`|...|`*bn*`)`'. A branch can consist of an arbitrarily long list of features, spacers are allowed except at the beginning or end of a branch. Alternatives can be nested, i.e. a branch can contain an alternative. (In terms of grammar, a branch must be a `FEATURE_LIST` (see section 5)). Examples:

`(SH2|SH3)` → either SH2 or SH3

There may be any number of branches, but only one of the branches may match (if you need to express the possibility of multiple branches simultaneously matching (*i.e.*, overlapping), use an Equivalence (4.8).

4.7 Ranges

It is possible to look for a variable number of occurrences of some arrangement of motifs (again, a `FEATURE_LIST` in grammatical terms), which we call a *range*. Delimit the list with parentheses and specify the maximum and minimum values between braces ('`{}`') just after the closing parenthesis: '`(...){`*m*`,`*n*`}`'. Here's a possible characterization of the nerve growth factor receptor family:

```
(TNFR_NGFR_2){1,4} = DEATH_DOMAIN
```

What this stands for is "one to four (inclusive) tumor necrosis or nerve growth factor receptor domain(s) (`TNFR_NGFR`), then a death domain". Ranges can be open-ended, e.g., `(XY){2,}` means at least two XYs, and `(XY){,3}` means at most 3 XYs.

4.7.1 Negatives

Ranges of the form `{,0}` can be used to indicate a motif that must not occur at this position. Here is an expression for a class of receptor protein kinases:

```
FURIN_LIKE = PKINASE
```

Such proteins fall in two categories: Insulin receptors and related; and ERB-like oncogenes. A discriminating feature is the presence, in the former group, of at least one fibronectin type-III domain (FN3) between the Furin-like and the Protein-kinase domains. To select the oncogenes, *i.e.* those who *don't* have any FN3 domain at this position, use this expression:

```
FURIN_LIKE = (FN3){,0} = PKINASE
```

4.8 Simultaneous Match ("Equivalence")

4.8.1 Variant 1: at least 1 branch must match (logical 'or')

Two different predictors of the same motif (say, Pfam and PROSITE's version of SH2) do not always completely agree : there may be small to medium discrepancies in the start and stop positions, for example. When several motifs can occur at the same position, or at least with some overlap, specify them with an *equivalence class*, [*b1*|...|*bn*]:

```
[PROSITE_SH2|PFAM_SH2]
```

This reads "SH2 from PROSITE, or Pfam, *or both* – in which case they must overlap". The branches of a equivalence class must be FEATURE_LISTs (see the grammar, section 5). There may be any number of branches.

4.8.2 Variant 2: all branches must match (logical 'and')

This variant lets the user specify that *all* branches must match. The matched substring is a contiguous region which has at least one match of each branch. For example, to see where a gene on the minus strand overlaps a gene on the plus strand, you may say:

```
[!PLUS_STRAND_GENE|MINUS_STRAND_GENE]
```

The '!' ensures that only regions with matches of *both* features will be reported. With an ordinary equivalence class, you'd get a report of all genes, because a match of single branch is enough for a match of the equivalence class.

4.8.3 Caveat!

Representing a sequence's features as a string has a potential problem, namely when two features start (or end) at the same position. This would be the case, for example, when two predictors of the same feature are in agreement (this is far from being always the case, but it happens). Suppose a sequence has an SH3 domain, identified both by a PROSITE profile and a Pfam HMM, starting on residue 53. The string representation could be

```
...53-<PROSITE_SH3#1-53 53-<PFAM_SH3#2...
```

or

```
...53-<PFAM_SH3#1-53 53-<PROSITE_SH3#2...
```

In this case, a simple metamotif like

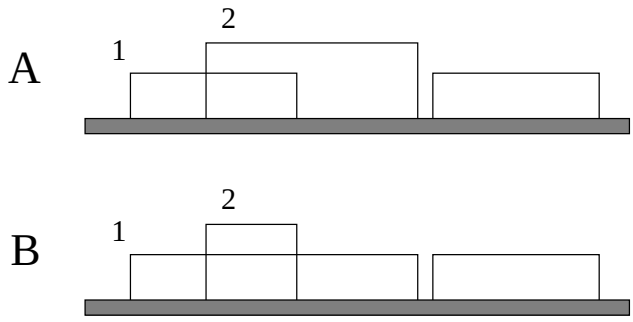
```
^PROSITE_SH3
```

will match only in the first case. The workaround is to use inclusive 'or's, like this:

```
^[PROSITE_SH3|PFAM_SH3]
```

4.9 Identifiers

Sometimes it is necessary to identify feature ends, *i.e.* to know which start corresponds to which stop. Consider the disulfide bridges in this diagram.



Both correspond to the arrangement,

```
<SS = <SS = SS> = SS> = SS
```

Where `SS` is a disulfide bridge. Now case A corresponds to the EGF domain (and a few others). However, the above expression cannot distinguish case A from case B and is thus not suitable for finding EGF domains. It must be modified to

```
<SS#1 = <SS#2 = SS>#1 = SS>#2 = SS
```

Where the '`#1`'s and '`#2`'s identify individual disulfide bridges.

5 Grammar

Here's the metamotif grammar:

```
METAMOTIF     ::= (SEQUENCE|FORK)+
SEQUENCE      ::= (FEATURE_LIST|GROUP)+
FORK          ::= L_BRACKET BANG? FEATURE_LIST { PIPE FEATURE_LIST }* R_BRACKET
GROUP         ::= L_PAREN SEQUENCE { PIPE SEQUENCE }* R_PAREN { RANGE }
```

```
FEATURE_LIST  ::= FEATURE_BLOCK { SPACER FEATURE_BLOCK }*
FEATURE_BLOCK ::= FEATURE_END+
SPACER        ::= INTEGER COMMA INTEGER
|             ::= COMMA INTEGER
|             ::= INTEGER COMMA
FEATURE_END   ::= FEATURE_START | FEATURE_STOP
FEATURE_START ::= L_A_BRACKET LETTER  { HASH ( LETTER | DIGIT ) }?
FEATURE_STOP  ::= LETTER R_A_BRACKET { HASH ( LETTER | DIGIT ) }?
INTEGER       ::= DIGIT+
LETTER        ::= ['A'-'Z'a'-'z']
DIGIT         ::= ['0'-'9']
L_PAREN       ::= '('
R_PAREN       ::= ')'
L_BRACE       ::= '{'
R_BRACE       ::= '}'
L_BRACKET     ::= '['
R_BRACKET     ::= ']'
L_A_BRACKET   ::= '<'
R_A_BRACKET   ::= '>'
COMMA         ::= ','
PIPE          ::= '|'
HASH          ::= '#'
BANG          ::= '!'
```

Note 1: The grammar has no notion of separators. In fact, separators are ignored after converting the feature names to a 1-letter representation (their role is precisely to allow this conversion), and they are not part of the automaton.

Note 2: The grammar has no notion of anchors ('^' and '$'). These characters do not cause different automata to be constructed (this used to be the case in older versions); they simply cause the automaton to behave differently (e.g., by aborting early in the case of '^').

Note 3: The grammar has no notion of "whole" features, because any such names are converted to the equivalent two-ends form (see 4.3).

A Examples of Metamotifs

It is assumed that match data are available, either in a database or by running a search program on-the fly, and that they are passed to `mmsearch` on standard input.

Eubacterial DNA polymerases:
`mmsearch '53EXO_N_DOMAIN = 5,10 = 53EXO_I_DOMAIN = 600, = C_TERM'`
Inclusion: sequences that have XPG_1 embedded in 53EXO_N:
`mmsearch '<53EXO_N_DOMAIN = XPG_1 = 53EXO_N_DOMAIN>'`
superposition: `PROTEIN_KINASE_DOMAIN` or `PKINASE` or both:
`mmsearch '[PKINASE|PROTEIN_KINASE_DOMAIN]'`

alternative:
`mmsearch '(IG|FN3) = PKINASE'`
repetition:
`mmsearch '(IG|FN3){2,4}'`
long repetition:
`mmsearch '(EGF){30,}'`
Tyr PK embedded into PK (Pfam or Prosite) - some are found outside PK domains!
`mmsearch '[<PROTEIN_KINASE_DOM|<PKINASE] = PROTEIN_KINASE_TYR = [PROTEIN_KINASE_DOM>|PKINASE>]'`
Gene with at least 10 exons:
`mmsearch '<gene = (exon){10,} = gene>'`
Gene less than 10 kb long:
`mmsearch '<gene#1 = ,10000 = gene>#1'`
Gene with at least 10 exons *and* less than 10 kb long:
`mmsearch -o pff -n 10_ex_gene '<gene = (exon){10,} = gene>' | mmsearch '<10_ex_gene#1 = ,10000 = 10_ex_gene>#1'`
See also the Hits examples[7] page.

B File Formats

B.1 GFF

GFF (General Feature Format) was originally proposed by Richard Durbin and David Haussler. This is a format for describing features in DNA sequences. A full description is available from `http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml` There is one record per line, each record pertains to one feature in one sequence. The fields are:

<seqname> <source> <feature> <start> <end> <score> <strand> <frame> [attributes]

An here's an example, taken from the above URL:

```
SEQ1    EMBL    exon    103     172     .       + 0
```

B.2 PFF

PFF (Protein Feature Format) is a derivative of GFF, specialized for protein features. It is also able to represent partial matches of a profile or HMM. Like GFF, PFF has one record per line, each record pertaining to one feature in one sequence. The fields are:

<sequence><seq_begin><seq_end><feature><ft_begin><ft_end><score>

[7] `http://hits.isb-sib.ch/doc/wwwmmsearch.shtml`

The *<seq_begin>* and *<seq_end>* fields are the positions of the match in the sequence. The *<ft_begin>* and *<ft_end>* are the positions of the match along the model (from the beginning and end, respectively). For full matches, these are 1 and -1, but when matches are partial, these may differ. If the first five positions of the model are missing in the match, say, then *<ft_begin>* will be 6. If the last five are missing, then *<ft_end>* will be -6.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools.* Addison Wesley, 1986.

[2] Philipp Bucher, Kevin Karplus, Nicolas Moeri, and Kay hofmann. A flexible motif search technique based on generalised profiles. *Computers and Chemistry*, 20:3–23, 1996.

[3] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques – A Practical Guide.* Ellis Horwood, 1990.

[4] Marco Pagni, Christian Iseli, Thomas Junier, Laurent Falquet, Victor Jongeneel, and Philipp Bucher. trEST, trGEN and Hits: access to dabases of predicted protein sequences. *Nucleic Acids Research*, 29:148–151, 2001.