`mmsearch` Implementation Notes

Thomas Junier
Swiss Institute of Bioinformatics

March 15, 2001

Abstract

This is a short expos of the motivations and strategy behind `mmsearch`. Others may argue that anything that doesn't fit well in the manual goes here.

First Attempts in Perl

The first attempt at a metamotif search engine tried to convert metamotifs into Perl regular expressions, and then to evaluate them. This posed a number of problems, due to the fact that regular expressions are a purely syntactic construct: you can check that, say, there are two series of digits separated by a space, but you can't check that their difference lies within a specified range. True, Perl's regexps allow you to save parts of the matches (using parentheses), but you have to refer to the matches by number, and build (and then evaluate) a new, additional expression for each condition. For example, to translate a metamotif such as

```
PKINASE = 10,20 = FN3
```

into a Perl regexp-plus-condition, you'd do something along the lines of

```
#!/usr/bin/perl

my ($min, $max) = (10, 20);
$_ = "128-PKINASE-201 210-FN3-254"; # string representation
my $condition = "\$range_ok = (\$2 - \$1 <= $max) and (\$2 - \$1) >= $min";
my ($pos_1, $pos_2) = /PKINASE-(\d+) (\d+)-FN3/;
if ($&) { # if the regexp matches, see if the range is ok
    eval ($condition);
    print "Match!\n" if $range_ok; # set during eval()
}
```

The trick is to build an appropriate `$condition` expression which references submatches in the regexp (indicated by the parentheses) by number: `$1` and `$2`.

Although this is clumsy, it's feasible. The same trick can be used for checking ranges. But consider this:

```
(PKINASE = 10,20 = FN3){1,3} = IG = 30,40 = EGF
```

now, depending on the number of 'PKINASE = FN3' repeats, the submatch referred to by $2 may refer to a spacer between a protein kinase and a fibronectin type-3 domain (boxed), or beween an immunoglobulin and an EGF domain (underlined). There is no way of knowing this beforehand, and hence a suitable $condition *cannot* be constructed.

The Current Strategy

The solution was to write a new search engine from scratch. The automata built by mmsearch are a little more than FSAs, however, in that some of the states' transitions depend on values parsed from their input. A Spacer state, for example, requires two parameters (the minimum and maximum number of residues allowed) and reads characters from the input string until it has read two sequences of digits separated by a space (see "String Representation" in the manual for the format of the input string). It then converts these sequences to integers, and subtracts the first from the second. The result, which is the number of residues between the adjacent features, is then compared to the minumum and maximum. If the test succeeds, the transition to the next state is allowed, otherwise transition fails.

Such automata could be written in any of several languages, but I chose Python because *i*) I wanted to evaluate it and learn it, *ii*) I needed to quickly find out whether the approach was promising, therefore a scripting language was ideal, and *iii*) although I am aware of the existence of parser generating tools like yacc, I wanted to try my hands at writing one from scratch, since I wanted to understand a bit how they work. Had I chosen C, I would probably still be debugging the lexer.

On the other hand, now that it works, I could rewrite it in C with relative ease, or better yet, rewrite only those parts that slow the program down - most probably the FSA and not the parser. Python makes this surprisingly easy.

Future Developments

Still, thee FSA-*cum-ad-hoc*-enhancements model is not entirely satisfactory. First, it's not very elegant, being a kludgy distortion of FSAs. It works alright, but there are useful operations that can't be done with this model, or at the expense of distorting it yet further. For example, representing a sequence's domains as a simple string becomes unwieldy when two domains start at the same position (see the caveat under "Equivalence classes" in the manual). The current work-around is to use equivalence, but this results in unncessarily complex metamotifs. The engine could of course be modified, but this would inevitably

lead to a point where the FSA part of the code becomes submerged in a jungle of grafts which try to glue other functionality to an FSA. At this point, one should really ask whether we really need the state automaton model, rather than a more general one. This is what I'm trying to do with a new implementation of `mmsearch`, which is based on a (kind of a) stack computer and stores match data as an ordered table rather than a string; and is an opportunity to learn Ruby. ;-)