A System for Pattern Matching Applications on Biosequences†

*Gerhard Mehldau and Gene Myers**

Abstract

ANREP is a system for finding matches to patterns composed of (1) spacing constraints called ‘‘spacers’’, and (2) approximate matches to ‘‘motifs’’ that are, recursively, patterns composed of ‘‘atomic’’ symbols. A user specifies such patterns via a declarative, free-format, and strongly typed language called A that is presented here in a tutorial style through a series of progressively more complex examples. The sample patterns are for protein and DNA sequences, the application domain for which ANREP was specifically created. ANREP provides a unified framework for almost all previously proposed biosequence patterns and extends them by providing approximate matching, a feature heretofore unavailable except for the limited case of individual sequences. The performance of ANREP is discussed and an appendix gives a concise specification of syntax and semantics. A portable C software package implementing ANREP is available via anonymous remote file transfer.

Introduction

In this paper we present a prototype system, *ANREP*, for defining and searching for a class of patterns that should facilitate research into patterns ranging from relatively simple descriptions of promotors (Hawley and McClure, 1983) and zinc fingers (Miller *et al.*, 1985), to more complex patterns, such as those recently posed for cytosine methyltransferases (Posfai *et al.*, 1989) and α-hemoglobins (Barton and Sternberg, 1990). The system is capable of searching sequences for patterns that are a series of approximate pattern matches separated by specifiable distance ranges. The patterns are communicated to the system in a small language called *A*.

ANREP extends previous work by providing a unified framework for concepts found in isolation in earlier systems: QUEST (Abarbanel *et al.*, 1984), a system for specifying character patterns built around exact matching of regular expressions, Staden's software (1988), that allows for the logical combination of sequence similarities via a list of spatial constraints; and the package by Gribskov *et al.* (1988), that computes and analyzes profiles that are positionally weighted

*Department of Computer Science, University of Arizona, Tucson, AZ 85721.
†Supported in part by NLM grant R01-4960 and the Aspen Center for Physics

consensus matrices describing protein domains. Furthermore, *ANREP* is the first system capable of handling approximate matches to network expressions rather than just keywords (as is possible in some earlier systems).

At the top level, our system supports patterns that are *network expressions* built from *motif/threshold pairs* and *spacers*. A network expression is a regular expression without Kleene closure†, or equivalently, any expression built up of concatenation and union operators. An example of such a pattern is given by the following specification:

```
net pattern = {A,1.0}(<0,20>{B,1.0}|<-5,5>{C,1.0})<25>{D,1.0};
```

The letter/number pairs in curly-brackets are motif/threshold pairs, and the numbers in angle-brackets are spacers. This network expression matches sequences consisting of some motif `A`, followed by either motif `B` (starting somewhere between zero and twenty symbols downstream) or motif `C` (five symbols upstream to five symbols downstream), followed by motif `D` exactly twenty five symbols downstream from the right end of the match for motif `B` or `C`. The threshold 1.0 paired with each motif specifies the degree of precision required to match it, a 1.0 asserting in this context that the match must be exact.

ANREP is a two-level pattern matching system in that motifs are in turn patterns, conceptually at a lower, second level. More specifically, a motif is a network expression of ‘‘atoms’’ that are either ASCII symbols or consensus symbols that generalize symbols to weighted and/or tailored combinations of symbol sets. For example, the motif `A` in the pattern above might be defined as the following simple network expression over the ASCII symbols `a`, `c`, `g`, and `t`:

```
motif A = "ac(gg|ta)ct(gt)?";
```

The motif exactly matches any of and only the four strings `acggct`, `acggctgt`, `actact`, and `actactgt`. Formally one says that the motif `A` *specifies* the set of strings it exactly matches.

Matches to motifs can be approximate as well as exact. A string is said to approximately match a pattern if it can be aligned, in the sense of traditional sequence comparison, with one of the strings specified by the pattern, in a manner that scores above a given threshold under some scoring scheme for sequence alignments. Note that an approximate match requires the specification of a threshold as well as a pattern. Any string can be aligned to any of the strings specified by the pattern, but not necessarily in a fashion whose score is high enough to make it interesting or sufficiently close to exact. In *ANREP*, each motif is coupled with a threshold that defines the stringency of the approximate match as seen in the example above. Moreover, the scoring scheme used to evaluate approximate matches is user-definable, and different scoring schemes may be associated with different motifs or even subparts of motifs. All approximate matches to a motif which exceed the associated threshold are considered by the system in attempting to match the top-level network expression of spacers and motif/threshold pairs.

---

† Kleene closure is the repetition (zero or more times) of some substring within a motif. Kleene closure is commonly denoted by ‘‘`*`’’, e.g., the motif `ac*a` would match strings `aa`, `aca`, `acca`, `accca`, ...

Given that the matching of motifs is approximate, it follows that during the search process each atom of a motif is conceptually scored against the search string symbol (if any) with which it is aligned. For ASCII symbols these scores are determined by a user-defined scoring scheme. But there is also the possibility of having an atom whose alignment scores are tailored specifically for it, or are some admixture of the schema scores for a set of symbols. *ANREP* provides such a capability in the form of *consensus symbols* that effectively generalize Staden's weight matrix concept (Staden, 1988). Furthermore, the fact that motif matching is based on alignment scores also provides an opportunity to value some portions of a match as being more important than others, i.e., one can *positionally weight* various parts of a motif. *ANREP*'s positional weighting feature generalizes the capability embodied in Gribskov's profile analysis software (Gribskov *et al.*, 1988).

The language *A* is free-format, declarative, and strongly typed. It permits patterns to be hierarchically defined and parameterized, supports the development of pattern libraries, and provides both enumerative and functional mechanisms for specifying scoring schemes. At the theoretical level, *ANREP* is based on a new algorithm for finding approximate matches to motifs, and a new optimized-backtracking method for matching network expressions of motifs and spacers. These algorithms are described in a forthcoming paper by Myers (1992) currently available as a technical report. The approximate motif matching algorithm is output sensitive in that its efficiency is proportionally to the stringency of the match required: the more exact the match, the more efficient it is. The algorithm generalizes the work of Ukkonen (1985) and specializes that of Myers and Miller (1989). The backtracking algorithm matches networks of spacers and motifs by first finding an optimal order of evaluations of its elements. Because of this it is particularly robust: in most cases search time is relatively independent of pattern complexity, as it is dominated by the time spent in an approximate search for the least frequently occurring motif.

The next section describes important attributes of the software and how to get a copy. Then the remainder of the paper introduces the system via a series of examples. Appendix A contains a quick reference guide. A complete description of the syntax and semantics for the language *A* are presented in Appendix B.

System and Methods

The *ANREP* system is implemented in C and was written and tested on a SUN 4/490 running Sun OS Version 4.1.1. The program is portable: the requirements are an ANSI-standard C compiler, accompanying standard I/O library, and UNIX™ operating system. The dependence on UNIX is in the communication of command line arguments and the ability to invoke commands from within the program. The software may be obtained by `ftp` to `cs.arizona.edu`. For a login use `anonymous` and descend to subdirectory `anrep`. There one will find a `README` file and a compressed tar file of the system software. The `README` file contains directions for uncompressing and unbundling the compressed tar file, and a `makefile` in the bundle constructs the system in an executable file called `anrep`. As is customary for UNIX tools, the

program `anrep` reads a pattern specification from the standard input and echos the specification and the results of pattern searches on the standard output. In this way, `anrep` can be part of a UNIX command "pipeline".

Like most systems, *ANREP* has a number of system defined limits. All of these are setup as defined constants in the file `anrep.h`. For example, the maximum number of parameters that may appear in a motif or network reference is set to 50 (`PARMAX`) but may be increased or decreased by simply changing the definition and recompiling the system. One important constraint is that the definition of `MAXMATCH` as 10000 limits the length of the longest *match* that can be reported. This should not be confused with a limit on the length of the longest *sequence* that can be searched. *ANREP* appropriately buffers its scan of a sequence so that such a limit is not required.

ANREP currently comprehends two very simple formats for files that are to be searched. The first is a plain file format where the file is assumed to be just a sequence of characters. The second is the Lipman/Pearson FASTA format (Lipman and Pearson, 1985) where the file is assumed to be a series of header/sequence pairs. A header is a consecutive sequence of lines whose first character is `>` and the sequence associated with that header is the subsequent lines to the next header. For example:

```
> CCHU (PIR) Cytochrome c - Human
GDVEKGKKIFIMKCSQCHTVEKGGKHKTGPNLHGLFGRKTGQAPGYSYTAANKNKGIIWG
EDTLMEYLENPKKYIPGTKMIFVGIKKKEERADLIAYLKKATNE
> SYBEHS (PIR) Thymidylate synthase - Herpesvirus saimiri
>       127.0,1.0,4.0,1.0,1.0
MSTHTEEQHGEHQYLSQVQHILNYGSFKNDRTGTGTLSIFGTQSRFSLENEFPLLTTKRV
FWRGVVEELLWFIRGSTDSKELSAAGVHIWDANGSRSFLDKLGFYDRDEGDLGPVYGFQW
RHFGAEYKGVGRDYKGEGVDQLKQLIDTIKTNPTDRRMLMCAWNVSDIPKMVLPPCHVLS
QFYVCDGKLSCQLYQRSADMGLGVPFNIASYSLLTCMIAHVTNLVPGEFIHTIGDAHIYV
DHIDALKMQLTRTPRPFPTLRFARNVSCIDDFKADDIILENYNPHPIIKMHMAV
```

Which format is to be used in a given search is specifiable in the **`search`** statement invoking the scan. This is done by giving the option — `PLAIN` or `FASTA` — immediately following the **`search`** keyword in parentheses. If left unspecified, the plain format is assumed. Examples illustrating this feature are included later in the paper.

Rather than try to accomodate the proliferation of file formats currently used in practice, we chose to confine *ANREP*'s access to a file during a search to three routines that one can augment to support their local needs. These three routines, described below, are the contents of the file `iopack.c`, and they are currently coded to support the two formats described above.

- `int Start_scan(format,file) char *format; FILE *file;`

  ANREP calls this routine once just before initiating a search. The parameter `format` is the string between parenthesis immediately following the **`search`**-keyword of the statement invoking the scan. This parameter is the empty string if a format does not follow the keyword. The parameter `file` gives the opened file to be scanned. `Start_scan` should return a 0 if the format string is not recognized/supported, and 1 otherwise.

- `char *Next_header()`

  When invoked, this routine is to return a string to the "header" for the next entry to be searched. If a match is found to the ensuing entry (fetched via `Get_sequence` below), then this header string is displayed along with a depiction of the match. If there is no next entry, then a `NULL` pointer should be returned indicating the end of the search.

- `void Get_sequence(where,amnt) char *where; int amnt;`

  The routine is to place the next `amnt` characters of the current entry at the location given by `where`. If there are fewer characters than this left in the entry, than the remainder should be transferred. In particular, *ANREP* determines that the entry is exhausted when this routine transfers zero characters. The string transferred must be null-terminated.

The software currently embodying this simple interface should suffice as an example to the programmer who wants to extend or alter the available formats. For example, GENBANK (Burks *et al.*, 1991), EMBL (Stoehr and Cameron, 1991), PIR (Barker *et al.*, 1991), and PROSITE (Bairoch, 1991) formats could all be added as desired.

Simple Patterns

ANREP's central function is to search a database file for instances of user defined patterns, called *networks*. A network is a network expression of *motif/threshold* pairs and *spacers*. Each motif/threshold pair, in turn, specifies an approximate match to a network expression over a set of symbols. Given a network and the name of a database file, *ANREP* searches the database for matches to the network and reports them in order of appearance.

The most simple motif is a keyword or sequence of symbols. An example of such an elementary pattern is the Pribnow box, "tata", found in many promotor sequences. To search for exact matches to this keyword, the following statement suffices:

```
search DATABASE for {"tata",1.0};  # A very simple example
```

The **`search`** statement initiates a scan of the contents of the file `DATABASE` for all occurrences of the network specified after the keyword **`for`**. The search views the file as a single long sequence of ASCII characters, new-line symbols included. In this example, the network expression is as simple as possible: a motif/threshold pair. The motif matches the literal string `tata`. We defer the treatment of approximate matching to the next section. For now it is enough to know that a default scoring scheme is automatically in effect and is such that a threshold of 1.0 requires that the matches to the motif be exact. Thus the example above searches `DATABASE` for all occurrences of the string `tata`. The example further illustrates that comments may appear between statements, begin with a pound-sign (‘‘#’’), and terminate at the end of the current line. In summary, the search statement operates on a file, specified directly after the keyword **`search`**, and looks for matches to the network specified after the keyword **`for`**.

In general, an *ANREP* network pattern is a network expression of motif/threshold pairs and spacers. The concatenation of two network expressions P and Q, denoted PQ, matches if there is a match to P, directly followed by a match to Q. The alternation of network expressions P and Q, denoted $P \mid Q$, matches if there is a match to either P or Q, and a match to a network expression P can be made optional by following the P with a question mark, i.e., $P?$. Alterna-

tion has the lowest precedence†, concatenation the next lowest, and optionality the highest. Parentheses may be used for grouping, or to change the order of precedence. A search for a pattern combining the so-called ‘‘-35 sequence’’ with a Pribnow box 15 to 20 base pairs downstream, can be written as:

```
search(FASTA) DATABASE for {"ttgac",1.0} <15,20> {"tata",1.0};
```

where the network consists of motif `ttgac`, concatenated with a spacer of the appropriate length, concatenated with motif `tata`. In this example, the inclusion of the format `FASTA` in parentheses after the **`search`**-keyword indicates that the file `DATABASE` is assumed to be a series of Lipman/Pearson FASTA formatted entries and will be searched as such. The default format is `PLAIN` and may either be specified explicitly or, as in our first example, implicitly. See the preceding section for details on these two supported file formats.

A spacer $<i,j>$ matches at least i and at most j symbols. The special spacer $<i,i>$ that matches exactly i symbols may be written more simply as $<i>$. Moreover, both i and j may be negative to indicate upstream rather than downstream offsets. For example:

```
search DATABASE for {P,1.0} <-5,10> {Q,1.0};
```

searches for an exact match to motif `P` followed by an exact match to motif `Q`, where the left end of the match to `Q` must start anywhere from ten symbols after, to five symbols before the right end of the match to `P`. Note that negative spacing permits matches to overlap, e.g., in the example matches to `P` and `Q` may overlap by as many as 5 symbols. The pattern `{P,1.0} <-x> {Q,1.0}` where `x` is the average length of a match to `P` may be used as an approximation to a request that a given stretch match both `P` and `Q`, but is not logically equivalent when the length of matches to `P` varies.

Motifs are not restricted to keywords, but can be network expressions of ‘‘atomic’’ symbols. Regular ASCII symbols (like `a`, `c`, `g`, or `t`) are the most simple and conventional atomic symbols. Examples of more complex atomic symbols (wild cards, classes, and consensus symbols) are given in a later section.

Networks and motifs (as well as alphabets, scoring schemes and numbers to be introduced) can be assigned to variables, which may then be used in subsequent statements. Variable names consist of any number of letters, digits, and underscores, but must begin with a letter or an underscore. For example, the sequence of statements:

```
motif Pribnow = "tat?aa?t";
motif M35Seq  = "(tc)?ttgac(a|t)";
net Promotor = {M35Seq,1.0} <15,20> {Pribnow,1.0};
search DATABASE for Promotor;
```

---

† If an operator `*` has higher precedence than operator `+`, what this means is that `a*b+c*d` is interpreted to mean `(a*b)+(c*d)` and not `a*(b+c)*d`. As another example, `PQ|R?S` is interpretated as `(PQ)|(R(S?))`. The precedence concept is a useful way to avoid having to use a lot of parentheses to make the interpretation of an expression unambiguous.

assigns motifs to the variables `Pribnow` and `M35Seq`; then assigns a network involving these two motif variables to the variable `Promotor`; and finally searches `DATABASE` for a match to the network denoted by `Promotor`. The motif `Pribnow` matches one of the four strings `taat`, `tatat`, `taaat`, and `tataat`, and the motif `M35Seq` matches one of the strings `ttgaca`, `ttgact`, `tcttgaca`, and `tcttgact`. The network `Promotor` then matches an exact match to `M35Seq` followed 15 to 20 symbols later by an exact match to `Pribnow`.

Variables are defined hierarchically (i.e., they must be defined before they are used), and may be parameterized. For instance, using the definitions for `Pribnow` and `M35Seq` above, one can define and use a parameterized network `Par_Promotor` as follows:

```
net Par_Promoter{T,S} = {M35Seq,T} S {Pribnow,T};
number cutoff = 0.75;
search DATABASE for Par_Promoter{cutoff,<15,20>};
```

`Par_Promotor` is parameterized with an approximate match threshold, `T`, for the two motifs in its network expression, and a network, `S`, to place between them. It is ‘‘called’’ in the subsequent **`search`** statement with a threshold of .75 for each approximate motif match, and a spacer of 15 to 20 symbols to place between the motifs. Any aspect of a motif or net can be parameterized. Under the scoring scheme that is in effect by default, the threshold of .75 permits one symbol to be mismatched in a match to `Pribnow`, and up to two in a match to `M35Seq`. The exact nature of approximate matching will be explained in the next section.

Alphabets, Scoring Schemes, and Approximate Matches

Alphabets are sets of symbols. They are used to declare the range of symbols over which a scoring scheme, and, by extension, a pattern is defined. Examples are:

```
alphabet DNA = [acgt];
alphabet Protein = [ARNDCQEGHILKMFPSTWYV];
```

which specify alphabets for DNA and protein sequences, respectively. The order of symbols between square brackets is not important as one is specifying a *set* of symbols. When searching a database, if a symbol is encountered that is not in the alphabet of a motif, then the symbol cannot be involved in an approximate match to the motif. Such symbols may, however, be spanned by a spacer.

In the context of approximately matching a motif, one must specify for each atomic motif symbol, what it costs to delete that symbol, substitute some other symbol for it, and insert symbols in the database sequence after it. Traditionally these *edit costs* are specified by a single global scoring scheme. In most previous pattern matching systems, this underlying scoring scheme was ‘‘hard-wired’’ into the program code. However, for many applications it is important to be able to experiment with different scoring schemes, either between different searches for the same pattern, or between different sub-patterns over the length of a pattern. For this reason, *ANREP* provides for the explicit specification of scoring schemes.

A scoring scheme consists of two components: an alphabet Σ, and a $(|\Sigma|+1)$ by $(|\Sigma|+1)$ scoring matrix δ whose indices range over the symbols in Σ and the special character $\varepsilon \notin \Sigma$. The entries of δ define the cost of the various editing operations as follows. For $x, y \in \Sigma$, the cost of replacing x with y is $\delta(x, y)$, the cost of deleting x is $\delta(x, \varepsilon)$, and the cost of inserting y is $\delta(\varepsilon, y)$. The single entry $\delta(\varepsilon, \varepsilon)$ is not used. Figure 1 shows a scoring matrix for DNA sequences that scores insertions and deletions, called *indels*, −1, unequal substitutions −.33, and equal substitutions 1. This scoring scheme will be referred to as the `Standard` scheme. Another scoring scheme, `Hybrid`, for DNA sequences is shown in Figure 2. Note that `Hybrid` gives indels such large negative scores (denoted $-\infty$) that they effectively cannot occur in any approximate match scored with respect to this scheme.

	ε	a	c	g	t
ε		−1	−1	−1	−1
a	−1	1	−.33	−.33	−.33
c	−1	−.33	1	−.33	−.33
g	−1	−.33	−.33	1	−.33
t	−1	−.33	−.33	−.33	1

Figure 1: Sample scoring scheme, `Standard`

Consider the motif `tata(aaaa)?` which exactly matches either `tata` or `tataaaaa`. Call these strings the sequences of the motif. An approximate match to a motif is a substring that aligns to one of the motif's sequences with a score that is sufficiently large. For example, in scanning a database we may find the matches:

```
... agttcagcatct-aataatccgacgacgatatcgttactcgg ...
              tataaa-aa           tata
```

to the substrings `tctaataa` and `tatc`. The score of a match is the sum of the scores δ assigns to each column of the alignment where insertions and deletions are introduced by placing dashes in the appropriate sequence. Thus, the score of the first match under the `Standard` scoring scheme is $2\delta(t,t)+\delta(a,c)+\delta(a,\varepsilon)+4\delta(a,a)+\delta(\varepsilon,t)$ = 3.67 and the score of the second match is $2\delta(t,t)+\delta(a,a)+\delta(a,c)$ = 2.67. Under the `Hybrid` scheme, the first match scores $-\infty$ (because indels are not allowed) and the second −8.

As seen from the `Hybrid`-based scores, a motif can actually match any substring of the database provided one is not concerned with the resulting score of the necessary alignment. Thus one is generally looking for matches whose score is above some threshold of interest. However, with motifs whose sequences vary in length, giving an absolute threshold is problematic. For example, if a threshold of 5.0 were required to report matches to our sample motif under the `Standard` scheme, then only matches to `tataaaaa` would ever be found since the best possible match to `tata` scores 4.0. In general, matches to long sequences of a motif are always

favored over the shorter ones. To rectify this, *ANREP*, compares a match's *length relative score* to the given threshold. The length relative score is obtained by dividing the score of the alignment by the length of the motif sequence involved in it. For example, the length relative scores of the matches above are .459 and .667, respectively. Further, the motif/threshold pair, `{tata(aaaa)?,0.4}` reports matches to `tataaaaa` whose absolute scores are above 3.2, reports matches to `tata` scoring above 1.6, and matches both of the substrings in the example above.

A **`score`** statement can be used to name and define a scoring scheme over a given alphabet. Two formats are provided for defining the scoring matrix δ — an *enumerative* and a *functional* format. Both formats may be combined within the same score declaration statement. All entries not set explicitly have a value of $-\infty$, i.e., not permitted. The enumerative format requires giving a list of matrix entries to be set, and the values they are to be set to. This format is useful for scoring schemes like the Dayhoff metric (Dayhoff 1978), where values do not have an easily characterizable pattern. An example of such a specification is the fragment:

```
score Dayhoff = Protein { <A.A>  #  0.90;
                          <A.R>  #  1.09;
                          <A.N>  #  0.99;
                          <A.D>  #  0.98;
                          <A.C>  #  1.12;
                            ...
                        };
```

which declares a variable, `Dayhoff`, of type scoring scheme over the alphabet `Protein`, and assigns values to entries in the semi-colon separated list between curly braces. Each item in the list consists of a set of matrix entries to the left of a pound sign and the value to assign to them to its right.

Within a pair of angle brackets, one can specify a set of matrix entries in a number of ways. The most basic specification is to give a comma-separated pair of characters, `<x,y>`, which denotes the individual entry, $\delta(x, y)$. Both x and y must be from the underlying alphabet of the scoring scheme, or they must be the ASCII symbol `$` which stands for ε. Either x or y (or both) can be replaced by a list of characters between square brackets, in which case the denoted set of matrix entries is the cross product of the sets of characters in each list, e.g., `<[ac],[gt]>` denotes the set of entries `<a,g>`, `<a,t>`, `<c,g>`, and `<c,t>`. If a period separates the pair instead of a comma, then the set denoted is the ''symmetric'' cross product of the left and right parts, e.g., `<A.R>` denotes the entries `<A,R>` and `<R,A>`; and the notation `<[ag].t>` denotes the entries `<a,t>`, `<g,t>`, `<t,a>`, and `<t,g>`. Finally, the notation `<x>` denotes the entries $<x, y>$ where y ranges over all the symbols in Σ, i.e., it denotes the entire row of the matrix labeled x except for the entry $<x, \varepsilon>$. An example making use of some of these forms is:

```
score Hybrid = DNA { <[acgt]>  #  -2;
                     <c.g>   #   3;
                     <a.t>   #   2;
                     <g.t>   #   1;
                   };
```

that creates a ‘‘hybridization’’ matrix where each pair of nucleotides that can stack is scored according to the number of hydrogen bonds it contributes to the duplex, and non-stacking pairs are scored –2. Note that each specification should be thought of as being ‘‘executed’’ in order of appearance. Thus the entry for $\delta(a,t)$ is first set to –2 in the first line, and later set to 2 in the third line. The matrix resulting from the above definition is shown in Figure 2.

	ε	a	c	g	t
ε		–∞	–∞	–∞	–∞
a	–∞	–2	–2	–2	2
c	–∞	–2	–2	3	–2
g	–∞	–2	3	–2	1
t	–∞	2	–2	1	–2

Figure 2: Hybridization scoring scheme, `Hybrid`

For the functional format, the part to the left of the pound sign is a boolean expression in two variables x and y that can take on values from $\Sigma \cup \{\varepsilon\}$. For every pair of symbol choices for x and y for which this expression is true, the entry $\delta(x,y)$ is set to the value given by the expression in x and y to the right of the pound sign. Both these expression may be arbitrary expressions in the C programming language. As such the ‘‘boolean’’ expression to the left of the pound sign is considered false if its value is 0 and true otherwise. For example:

```
score Standard = [acgt]
     { x!='$' && y!='$' # (4*(x==y) - 1) / 3.;
       < [acgt] . $ >   # -1.;
     };
```

defines the `Standard` scoring scheme for DNA sequences shown in Figure 1. All entries of the matrix δ, except for the first row and the first column (where the expression `x != '$' && y != '$'` evaluates to false) are set to the value of the expression to the right of the pound sign which evaluates to 1.0 when $x = y$ and –.33 otherwise. The ε entries are set by the second, enumerative clause to –1.

In an *ANREP* specification there is the concept of the *current scoring scheme*. Initially, it is set by default to the *unitary scoring scheme*, `Unitary`, over the alphabet of ASCII characters. The δ-matrix for this scheme does not permit indels, scores identical substitutions 1, and mismatches 0. Defining a scoring scheme has the effect of making it the current one. It then applies to all motif definitions until another **`score`** statement is encountered. Moreover, at the time of its definition, a motif is bound to the current scoring scheme in the sense that it is the one

used to score matches to the motif, regardless of any resettings of the current scheme subsequent to the definition. The abbreviated statement:

```
score MyScheme;
```

assumes `MyScheme` is a previously defined scoring scheme variable, and has the effect of making it the current one. This permits one to define a number of schemes and have different ones be in effect for different portions of the specification.

Recall that in the examples of the first section, it was asserted that patterns of the form `{P,1.0}` required exact matches to the network expression of the motif for `P`. Since no scoring schemes are defined in those examples, the default unitary scoring scheme was in effect throughout. Under this scheme the only way a match can have a length relative score of 1.0 is for its absolute score to be equal to the length of the motif word it matches, and the only way this can happen is if the match is exact because only equal substitutions score at least 1. Thus in the default case, `{P,1.0}` denotes an exact match to `P`. This is also true for the `Standard` scheme above, but not for schemes like `Hybrid`.

While the current scoring scheme is generally used to determine the scoring of a motif declared within its scope, any part of a motif may be scored under any other scheme by following the part in question with an exclamation mark (‘‘`!`’’) and the name of the scoring scheme to be applied to it. The exclamation mark operator has the same precedence as optionality (‘‘`?`’’). For example, for the motif `M35Seq` defined as follows:

```
score Unitary;
motif M35Seq = "(tc)? (ttgac)!Standard ((a|t)t)?";
```

an approximate match is scored with the default unitary scheme except for the part of the match to the submotif `ttgac` which is scored with the sample scoring scheme of Figure 1. For example, if the motif sequence `tcttgacat` is matched to the sequence `actgccaat` as follows:

```
ac | t-gcca | at
tc | ttgac- | at
```

then the part of the alignment between the bars is scored with `Standard` and the rest with `Unitary`. The former scores .67 and the latter two parts score 3 for a length relative score of .408. Note that the unaligned `a` is considered to be a part of the alignment scored with `Standard`. In general, an unaligned character in the database is scored according to the scheme associated with the first motif symbol to its left. If there is no such motif symbol, then the unaligned database character is scored according to the current scheme.

Positional Weights

The concept of positional weights is introduced into *ANREP* to let the user express the relative importance of different parts of a motif. With any submotif, one can associate a factor or weight by placing a colon after the subexpression in question, followed by the weight, an integer or real number in the range $\pm 10.0^{10}$. The precedence of the colon operator is the same as that of ‘‘`!`’’

and ‘‘?’’. The score for aligning the given submotif is multiplied by the score of the associated weight. Thus weights of greater than 1 increase the importance of a portion of a motif, and weights less than 1 decrease their relative importance. In addition, nested weight factors multiply. That is, if a subexpression weighted by α is nested within another that is weighted by β, then the scores of the alignment for the inner motif are multiplied by $\alpha\beta$. As an example of positional weighting consider:

```
motif M35Seq = "(tc):.5 ttgac ((a|t):.5 t):.5";
```

which simply ‘‘downgrades’’ the less important parts of motif `M35Seq` as opposed to making them entirely optional, as was done previously. The first sub-motif, `tc`, has a weight of 0.5; the second sub-motif, `ttgac`, has the default weight of 1.0; the third sub-motif, `a|t`, has a weight of 0.25 (nested weights), and the last sub-motif, `t`, has a weight of 0.5.

When using positional weights, the relationship between scores, weights and length-relative thresholds should be kept in mind. For example:

```
net Pribnow1 = {"ta t:.5 a a:.5 t",1.0};
net Pribnow2 = {"(ta):2 t a:2 a t:2",1.0};
```

are not equivalent patterns. Assuming the default unitary scoring scheme and an input string (database) of `...tatact...`, net `Pribnow1` matches with an absolute score of 4.5, which translates into a length-relative score of 0.75. For `Pribnow2`, however, we get an absolute score of 9.0, and a relative score of 1.50. The threshold of 1.0 is therefore appropriate only for `Pribnow2`. To obtain identical results with `Pribnow1`, its threshold should be set to 0.5.

Atoms

Motifs are network expressions built up with the operators `|`, `!`, `:`, concatenation, and `?` from a collection of *atoms* which in all the examples thus far have simply been ASCII characters. The set of ASCII symbols permitted in a motif are restricted to be those in the alphabet of the scoring scheme associated with part of the motif the symbol is in. For instance, in the case of the `Standard` scoring scheme defined earlier, the legal symbols are those of the alphabet `DNA` = {`a`,`c`,`g`,`t`}. In this section, we specify and illustrate the ways *ANREP* extends the notion of atom in a number of powerful ways.

The Wild Card

The wild card symbol, a period (‘‘`.`’’) in *ANREP*, is an atom which is defined for all alphabets and matches any symbol from the alphabet of the current scoring scheme. An example is:

```
motif wild = "ca.t";
```

which matches any one of the four strings `caat`, `cact`, `cagt`, and `catt` (provided `DNA` is the alphabet of the current scoring scheme). The wild card scores 0 when substituted for any character and may not be deleted. But the most unique characteristic of a wild card is that it does not

contribute to the length of the motif sequence matched to the text even though it must be substituted for. For example, if `wild` is matched to the text string `ctgt` under the `Unitary` scoring scheme, the length of the match is 3 (not 4), and the length relative score is .667 (not .5). The reason for this feature is that it is desired that wild cards be usable as intra-motif spacers.† For example, `PartA"....?.?.?"PartB` is a motif where a match to the pattern for `PartA` is separated from a match to the pattern for `PartB` by 3 to 6 symbols. The critical semantic difference between the above and putting a spacer between motifs for the two parts, is that the example is considered a single motif. Thus the length-relative score of a match to this single motif is the sum of the scores of the two subpart matches divided by the sum of the lengths of the two subpart motif sequences involved.

Symbol Classes

A symbol class is a generalization from a single-symbol to a multi-symbol atom. A symbol class is specified as a set of symbols in square brackets, and it matches any of the symbols in the set. For example:

```
motif class = "a[act]a";
```

matches any one of the strings `aaa`, `aca`, and `ata`. Of course, this motif could also be written as:

```
motif class = "a(a|c|t)a";
```

but not only is the symbol class easier to specify than the alternation of symbols, it also is much more efficiently searched for.

It should be noted that the declaration:

```
motif semi_wild = "ca[acgt]t";
```

is *not* equivalent to the wild card example above for three reasons: (1) if substituted for, the class has zero score only if the maximum substitution cost against one of the characters is zero, (2) the class may be deleted if any of the class' characters can be deleted, and (3) the class counts in the length of a match. For example, under the `Unitary` scoring scheme `semi_wild` matches `ctgt` with an absolute score of 3, and a length relative score of .75.

Consensus Symbols

In an effort to provide the user of the system with even finer control over the motif to be matched, *ANREP* further generalizes symbol classes to consensus symbols. Consensus symbols provide a way of specifying, at a given position in a motif, exactly which symbols do or do not

---

† Another reason for this choice is that the alternative — counting a wild card in the length of a match — is attainable with an appropriately designed consensus symbol. The definition chosen gives *ANREP* a capability it would not otherwise have.

match, and their scores. That is, they allow one to create a ‘‘symbol’’ whose edit costs are either explicitly specified or are a function of the edit costs for a set of characters.

In order to define the effect of consensus symbols and to show that they generalize symbol classes, the concept of a *cost vector* is needed. How a specific symbol, say x, of a motif is aligned in a match is completely determined by the scores for replacing it, deleting it, and inserting characters after it. These scores are all contained in just two rows of the underlying scoring scheme in effect for the portion of the motif containing x. Namely, the ε-row, $\delta(\varepsilon, \Sigma)$, which gives the cost of inserting symbols after x, and the x-row, $\delta(x, \Sigma \cup \{\varepsilon\})$, which gives the cost for deleting x ($\delta(x, \varepsilon)$), and substituting another symbol for x (the rest of the vector). We call these two vectors the cost vectors associated with symbol x. Figure 3 shows the cost vectors for symbol `g`, again assuming the `Standard` scoring scheme for DNA sequences.

	ε	a	c	g	t	
ε		−1	−1	−1	−1	$\delta(\varepsilon, \Sigma)$
a	−1	1	−.33	−.33	−.33	
c	−1	−.33	1	−.33	−.33	
g	−1	−.33	−.33	1	−.33	$\delta(g, \Sigma \cup \{\varepsilon\})$
t	−1	−.33	−.33	−.33	1	

Figure 3: Cost vectors for symbol `g`

Now note that if one knows the two cost vectors for a symbol then one no longer needs to know the symbol in order to correctly align and score the position it occupies in a motif. That is one may think of a symbol as just being a shorthand for the specification of two cost vectors. A consensus symbol is *ANREP*’s mechanism for creating ‘‘symbols’’ whose cost vectors are under explicit user control. There are two different kinds of consensus symbols. For a *reduction* consensus symbol, the cost vectors are determined by some functional combination of the cost vectors of the symbols in the character class notation specifying the consensus symbol. The combination function, or so-called reduction, to be employed is specified by the first symbol after the left square bracket: ‘‘`>`’’ for maximum, ‘‘`~`’’ for average, and ‘‘`<`’’ for minimum. For a *tailored* consensus symbol, to be discussed latter, the elements of its cost vectors are explicitly set via a syntax that is a variant of the class notation.

The cost vectors of a reduction consensus symbol are obtained by applying the appropriate reduction function, element-wise, to the cost vectors of the set of symbols in its specification. For example:

```
motif ReductionSymbols = "[~at][>at]" ! Standard;
```

consists of two consensus symbols `[~at]` and `[>at]`, whose cost vectors are the average and maximum reductions, respectively, of the cost vectors for the symbols `a` and `t`. Figure 4 shows how these new cost vectors are computed from those of the symbols. The immediate question is, ‘‘Why are the deletion and insertion scores $-\infty$?’’ In order to increase flexibility we chose to

make indels prohibited as the default, i.e., the reduction function is applied to all substitution costs, but indels are automatically set to $-\infty$. If it is desired that either or both insertion and deletion scores be set according to the reduction of the contributing symbol cost vectors, then one should include a ‘‘-’’ for deletions, and ‘‘+’’ for insertions, into the character class of the consensus symbol. Figure 4 illustrates this as well. It should be clear that the idea of reductions could employ other functions. Minimum, maximum, and average were those that we thought clearly useful.

X	$\delta(\varepsilon, \Sigma)$				$\delta(X, \Sigma \cup \{\varepsilon\})$				
	a	c	g	t	ε	a	c	g	t
a	–1	–1	–1	–1	–1	1.0	–.33	–.33	–.33
t	–1	–1	–1	–1	–1	–.33	–.33	–.33	1.0
[~at]	–∞	–∞	–∞	–∞	–∞	.33	–.33	–.33	.33
[>at]	–∞	–∞	–∞	–∞	–∞	1.0	–.33	–.33	1.0
[~at+]	–1	–1	–1	–1	–∞	.33	–.33	–.33	.33
[>at+-]	–1	–1	–1	–1	–1	1.0	–.33	–.33	1.0

Figure 4: Computation of cost vectors for several consensus symbols

The special symbols + and -, when introduction into a consensus symbol’s class, ‘‘turn on’’ the application of the reduction function to the insertion and deletion scores, respectively. The addition of another special symbol, ^, has the effect of introducing every symbol not in the class, into the reduction computation. Alone this last feature is not of much value because it implies that the reduction is over every symbol in the alphabet. But it is also possible to weight the contribution of each symbol’s vector by placing a weight after the symbol separated with a colon. The notation for these weights is similar to that for positional weighting but the syntax is simpler: a given weight applies to all symbols to the left of it, but right of the previous weight. For consistency the required default is that unweighted symbols are weighted 1. To illustrate these features, consider the examples:

```
motif Complex1 = "[> ag:1.5 ^]!Standard";
motif Complex2 = "[> -:0.8 +:0.5 ag:1.5 ^]!Standard";
```

that leads to the cost vectors shown in Figure 5. For `Complex1` substitution scores are the maximum reduction of all letters, where `a` and `g` are weighted 1.5 (`ag:1.5`) and other symbols 1.0 (^ included but not weighted). Insertions and deletions are not allowed. The consensus symbol of `Complex2` is the same, save that insertion and deletion have been "turned on" at weights 0.5 and 0.8, respectively. Note that these weights are applied to the result of the weighted reduction of the cost vectors implied by the other characters in the class. In this way, insertion and deletion costs may have their scores increased or decreased relative to substitution scores.

X	$\delta(\varepsilon, \Sigma)$				$\delta(X, \Sigma \cup \{\varepsilon\})$				
	a	c	g	t	ε	a	c	g	t
a	(–1	–1	–1	–1)*1.5	(–1	1.0	–.33	–.33	–.33)*1.5
g	(–1	–1	–1	–1)*1.5	(–1	–.33	–.33	1.0	–.33)*1.5
c	(–1	–1	–1	–1)*1.0	(–1	–.33	1.0	–.33	–.33)*1.0
t	(–1	–1	–1	–1)*1.0	(–1	–.33	–.33	–.33	1.0)*1.0
Complex1	–∞	–∞	–∞	–∞	–∞	1.5	1.0	1.5	1.0
Complex2	–.5	–.5	–.5	–.5	–.8	1.5	1.0	1.5	1.0

Figure 5: Computation of cost vectors for consensus symbol `Complex`

It should now be evident that consensus symbols are in fact a generalization of symbol classes. Specifically, `[act]` is equivalent to `[>+-act]` and is implemented exactly this way. This is significantly faster than `(a|c|t)` because a single pair of cost vectors is built once for the consensus symbol, rather than repeatedly having to take a three-way maximum during the matching of the network. With the weighting feature and the special features embodied in `+`, `-`, and `^`, *ANREP* provides a rich set of possible ways of summarizing the information content of a position in a motif.

The final atomic symbol to be considered, is the tailored consensus symbol that permits one to explicitly specify each element of the consensus' cost vectors. An example of such a ''tailored'' symbol is:

```
motif Tailored = "[= +:-1 ag:2 ^:1]";
```

The syntax is exactly the same as that of a reduction consensus symbol, but the interpretation is different. In this case, the weight associated with a symbol is the score for matching that symbol. For example, in the case of motif `Tailored`, substituting an `a` or `g` scores 2, substituting a `c` or `t` scores 1; each symbol inserted immediately after it scores –1, and the symbol may not be deleted. Any character that is not associated with a weight is assumed to be scored –∞, i.e., not permitted. For example, `[=a]`, must be matched to the symbol `a` at cost 1. It is the only symbol that the atom can be aligned with and it must be aligned with it because the atom cannot be deleted or inserted.

Tailored consensus elements permit one to specify arbitrary weight matrices (Staden, 1988; Stormo 1990). For example, Hawley and McClure (1983) compiled the Pribnow sequence of 112 *E. Coli* promotors. While the consensus is `tataat`, as used in the patterns of the first section, very few promotors exactly match this string. A better summary of the information present in the promotor region is a record of the percentage of times each character appears in a given position of the promotor. The motif below captures this information in the weight matrix of consensus symbols below. For example, the second consensus symbol asserts that an `a` appears in the second position 94% of the time. The length relative score of a match of `HandM` to the consensus `tataat` is 4.22/6 = .703.

```
motif HandM = "[= a:.02 c:.09 g:.10 t:.79]
               [= a:.94 c:.02 g:.01 t:.03]
               [= a:.26 c:.14 g:.16 t:.44]
               [= a:.59 c:.13 g:.15 t:.13]
               [= a:.50 c:.20 g:.13 t:.17]
               [= a:.01 c:.03 g:.00 t:.96]";
```

While the matrix above illustrates the concept of the position of a consensus pattern as being tendencies to be certain symbols, many investigators advocate that these tendencies be the log (to the base of the alphabet size) of the odds ratio of each symbol (Stormo, 1990). For example, the consensus symbol for the first position of the pattern above would be `[= a:-1.82 c:-.74 g:-.66 t:.83]`. We will not treat such information based measures in detail here, we simply wish to point out that *ANREP* provides the capability to model such patterns.

Note that consensus symbols, while inducing quite intricate cost vectors, are conceptually just ‘‘characters’’ in a motif. Thus their scores are still subject to multiplication by positional weighting factors, and the final score of a match to the motif containing them will be normalized with respect to length before being compared against the threshold.

Libraries, Environment Interface, and Search Output

To facilitate libraries of predefined components, *ANREP* provides an **`include`** statement that when encountered reads the contents of the specified file and processes the *ANREP* statements therein as if they had been typed in-line. **`Include`** statements may be nested. For example, setting up the Dayhoff scoring scheme from the earlier section requires typing 110 entries. This can be done once, put in a file (for this example `Dayhoff`), and then included in any desired pattern specification with the statement:

```
include Dayhoff;
```

One might, for example, develop a library of oft-used alphabets and scoring schemes and include it at the start of every *ANREP* specification. Another example would be to establish a parameterized pattern specification in a file and then experiment with it by invoking *ANREP* in interactive mode, including the pattern specification, and then making a series of enquiries into a database with **`search`** statements over the parameterized pattern. Another example would be to develop libraries of patterns from, say, the PROSITE repository.

In some cases, the specification *ANREP* is to be applied to may be the result of a calculation. For example, one may want to search for patterns that are formed by taking some kind of consensus of a multi-alignment (Gribskov, 1988; Barton and Sternberg, 1990). To facilitate such situations the **`execute`** statement can capture the output of any program and then process it as if it had been included directly into the specification. As a simple example, the *ANREP* package contains a program `pam` that produces a valid *A* score statement defining a Dayhoff substitution matrix at any desired number of PAMs. The **`execute`** statement below issues the command in

quotes as if it had been typed at the command-line level, and captures the output which defines a scoring scheme `PAM135` that is the Dayhoff matrix at 135 PAMs.

```
execute "pam -n135 -iPAM135";
```

The program *pam* also has an option to scale the matrix. Note that the **`execute`** statement is a generalization of the **`include`** statement as seen by the fact that, `execute "cat foo"`, is identical in effect to, `include foo`.

Just as it is useful for *ANREP* to be able to "call" other programs with the **`execute`** statement, so it is that other programs can call *ANREP* with parameters that control its execution. At the command-line level *ANREP* is invoked by the command, `anrep <arg1> <arg2> ...`, where the command-line arguments are optional and arbitrary. In keeping with UNIX convention for "pipes", *ANREP* expects to see the *A* specification on the standard input, and produces the echo and search results on the standard output. Any command-line arguments are passed separately to *ANREP* as character strings. Within the *A* specification, any occurrence of a string of the form `@i` where `i` is an integer, is interpreted to be a reference to the i^{th} argument on the command line. The argument string is substituted for the reference before *ANREP* processes the specification, so there are no restriction on the syntax of the argument other than that the specification after substitution be syntactically valid. For example, if a specification contains the statement:

```
execute "pam -n@1 -iDayhoff";
```

then the command `anrep 250` will interpret the specification as producing a Dayhoff matrix at 250 PAMs. One may pass thresholds, names, patterns, or any other pertinent specification feature as a command line argument.

The **`execute`** statement and command-line features allow *ANREP* to invoke other programs and to be controlled by programs that invoke it, respectively. These mechanisms for interacting with the environment are rich enough that one can use *ANREP* as the search engine for a pattern learning system. The system produces an initial instance of the variable part of the pattern to be learned as an appropriately parameterized *ANREP* specification. Then it invokes *ANREP* on a specification containing (1) the fixed pattern parts, (2) an **`execute`**- or **`include`** statement that fetches the generated part, and (3) a search of the pattern over a training set. The results are captured by the pattern learning system, the parameters refined, and *ANREP* re-invoked in a classic feedback loop, until the pattern is learned.

When invoked *ANREP* echos the pattern specification given to it as input (but not the contents of included files), and after each **`search`** statement outputs a display of each of the matches it finds for that search. What this display looks like depends in part on the the parenthesized modifier of the **`search`** statement that asserts the format of the data to be searched. Recall from the section on System and Methods that the formats supported depend on the coding of the I/O routines in `iopack.c`. Each match is prefaced by the "header" returned by the routine `Next_header`. For the `PLAIN` format this is null since there is only one

sequence. For the FASTA format the lines beginning with a > immediately before the sequence entry containing the match constitute the header. The substring of the entry or string against which a match is found is displayed, 50 characters to a line if more than one line of output is required. Each of these segments is enclosed in brackets of the form *i*> and <*j*, where *i* and *j* are the positions of the first and last symbols of the string matched, respectively. Underneath these segments is a line for each matched motif, where the lines are in left-to-right order of the motifs involved in the match. Each motif match is depicted by a line of the form: "*x*: . . ^ ^ . .", where *x* represents the length-relative score of the best match to that particular motif. The circumflex symbols are positioned under those database symbols giving the best match, and the period symbols indicate the total range of possible matches. For example, the search in the comprehensive example of the next section results in several matches, one of which is reported as follows:

```
>S00920 (PIR) Modification methylase, NgoPII - Neisseria

        13> KIISLFSGCGGLDLGFEKAGFEIPAANEYDKTIWATFKANHPKTHLIEGD <62
    1.0000: ..^^^^^^^^^^^^^^^^^^..
    0.8750:                    ^^^^^^^^^^^^^^^^^^
    1.0000:                                                ^->

        63> IRKIKEEDFPEEIDGIIGGPPCQSWSEAGALRGIDDARGQLFFDYIRILK <112
    1.0000: ^^
    1.0000:            ..^^^^^^^^^^^^^^^^^^^^^^..
    1.0000:                                ^^^^^^^

       113> SKQPKFFLAENVSGMLANRHNGAVQNLLKMFDGCGYDVTLTMANAKDYGV <162
    1.0000:   .^^^^^^^^^^^^^^^^^^.
    1.0000:                                     ^^^^        ....
    1.0000:                                             ..^^^^^^^->

       163> AQERKRVFYIGFRKDLEIKFSFPKGSTVEDKDKITLKDVIWDLQDTAVPS <212
    1.0000: ^^^^^^^^^^..

       213> APQNKTNPDAVNNNEYFTGSFSPIFMSRNRVKAWDEQGFTVQASGRQCQL <262

       263> HPQAPKMEKHGANDYRFAAGKETLYRRMTVREVARIQGFPDNFKFIYQNV <312
    1.0000:                          .^^^^^^^^^^^^^^^.
    0.9091:                                                .^^->

       313> NDAYKMIGNAVPVNLAYEIAA <333
    0.9091: ^^^^^^^^^^^^^^^^^^^.
```

Note, for example, that the small motif, VII, of the example matches in potentially two places on the line displaying symbols 115 to 164 of the sequence. The -> at the end of the display of symbols 115 to 164 indicates that the match to that motif continues on the next line.

A Comprehensive Example

To illustrate the use of *ANREP* on a real problem, we show how it might be applied to the pattern developed by Posfai *et al.* (1989) for the structure of the cytosine methyltransferases (m^5C

MTases). The example illustrates the flexibility of *ANREP* and the ease with which it can be used. Using a program that looks for strongly conserved positions and a data set of thirteen bacterial m^5C MTases, the investigators proposed that such MTases are characterized by a pattern consisting of a sequence of ten motifs in their amino acid sequences. In the syntax of *ANREP* the pattern is given by:

```
# Cytosine Methyltransferase pattern (Posfai et al. NAR 17, 7 (1989))

motif I = "[ILM][DS][FL]F[ACS]G.[GM][AG][FIL]..[AGS]....G";

motif II = "[ILV]..[INS][DE].[DFN]..[AI]..[STV][FIY]..[IN]";

motif III = "D[IV][RST]";

motif IV = "[DN].[ILV].[AGS]G[FPS]PC[PQ].[FW]S..G.....[EDS]";

motif V = "[EDP].[QR][GN].[LMV][FY]";

motif VI = "[PT].....ENV.[GN]......[GKN]";

motif VII = "[DG]Y.[FIV]";

motif VIII = "[DIN][ADS]..[FHY][FGN][ILV][AP]Q.R[EKQ]R...[EIV][ACG]";

motif IX = "R.[FLM][HTS]..E..[ARV][ILV][MQ].[FY][DEP]";

motif X = "[KRS]....Y[KQR][EMQ].GN[AS][IV].[IPV].[ALV]....[AFG]";

net MTase{t} =
       {I,t} <-9,39> {II,t}  <-5,20> {III,t} <-4,34> {IV,t} <-13,41> {V,t} <-1,19>
       {VI,t} <1,42> {VII,t} <-7,21> {VIII,t} <34,322> {IX,t} <-5,25> {X,t};

search(FASTA) PIR for MTase{.8};   # Search FASTA-formatted PIR for an 80% density match.
```

The individual motifs were arrived at by looking for positions that were perfectly conserved, and secondarily, those positions that varied among two or three amino acids. Positions containing more than three amino acids were considered non-specific and modeled as wild-cards in the pattern above. The stretches accepted as motifs, were those whose probability of matching at random in the database of interest was less than a certain level. For the spacing between the motifs the investigators used the variation that occurred between motif instances in the thirteen MTases analyzed.

The focus of the authors was to arrive at a pattern that exactly discriminated the MTases in that it matches only and every protein sequence that functions as a methyltransferase. *ANREP* doesn't help investigators arrive at potential patterns, but given a hypothesis, it is very easy to use *ANREP* as a search vehicle to see if the proposed pattern does indeed discriminate and to what degree. The example above illustrates how easy it is to set up a search. In fact, the pattern above generalizes the pattern given in the paper in several ways. First, matches may be approximate to varying levels of stringency by adjusting the single parameter `t` that controls the

stringency of all ten motif matches simultaneously. Under the default scoring scheme, searching with `t` set to `1.0` looks for exact matches. Second, the spacing between the motifs was loosened by expanding each spacer by 10 positions on each side (e.g. `<1,29>` was expanded to `<-9,39>`), except for the large interval between motifs `VIII` and `IX` which was expanded by 50 on each side. It is interesting to note that with `t` set to `.8`, this pattern when searched against the PIR database matched exactly those MTases in the database *and no others* with the one exception of the more recently sequenced NgoPII methylase that the authors also found by searching for less than "perfect" matches but in basically an ad hoc fashion.

The power of *ANREP* is that such experimentation with pattern variations is easy to realize along a number of dimensions. For example, one could establish different stringency parameters for different motifs, one could affect the matching process by establishing, say, the PAM250 scoring scheme before the definition, one could vary the spacing between motifs, one could positionally weight atoms dependent on their ambiguity, one could switch from character classes to averaged classes, and so on. More concretely consider the two examples of how one might "experiment" with motif `I`.

```
motif I = "[ILM] ([DS] [FL] :2) (F:3) [ACS] (G:3) .
          ([GM] [AG] :2) [FIL] .. [AGS] .... (G:3)" ;

motif I = "[~ I:7 L:3 M:3 +-] [~ D:6 S:7 +-] [~ F:1 L:12 +-] [~ F:13 +-]
          [~ A:5 C:1 S:7 +-] [~ G:13 +:.9 -] . [~ G:12 M:1 +-] [~ A:3 G:10 +-]
          [~ F:6 I:3 L:4 +:.8 -] .. [~ A:5 G:7 S:1 +:.6 -] .... [~ G:13 +-]";
```

Both definitions attempt to affect the value of each position in the pattern according to how unambiguous it is among the thirteen MTases. In the first case, positions that were absolutely conserved are weighted 3, those that varied among 2 amino acids are weighted by 2, and the rest by 1. The second cases attempts to take the actual composition of each position into more careful consideration by letting each position be the weighted average of the symbols that occurred in each position among the thirteen MTases that defined the pattern. In fact the second definition corresponds exactly with the manner in which many authors such as Gribskov (1988) and Barton and Sternberg (1990) have chosen to summarize information from a multi-alignment into a consensus. Note that the cost of insertions for the symbols just before wild-cards is positionally downgraded as might be done in a Gribskov profile. That is, insertion is (admittedly arbitrarily) weighted 90% before a single wild-card, 80% before two, and 60% before four. The idea is that there is greater variability in the pattern at such locations. Note that unlike Gribskov's work, *ANREP* cannot support affine gap costs.

As a final example, we demonstrate that *ANREP* is capable of modeling Barton and Sternberg's "flexible patterns". Already noted just above is the capability of summarizing a position in a multi-alignment by the weighted average of the cost vectors for the residues in a column of the alignment. We also not that the other four scoring methods mentioned in Barton and Sternberg (1990) can also be easily realized in *ANREP*. Between the strongly conserved motifs, they advocate allowing spacing according to the variation found in the multi-alignments. This could be done with spacers but in order to exactly mimic their work, one needs to use

wild-cards so that the pattern is still a single motif. The α-hemoglobin example from the paper would be modeled as follows:

```
# Globin pattern (Barton and Sternberg, J. Mol. Biol. 212 (1990))

motif I = ...;
motif II = ...;
motif III = ...;
motif IV = ...;
 . . .

motif S0_12 = ".?.?.?.?.?.?.?.?.?.?.?.?.?";
motif S0_4  = ".?.?.?.?";
motif S0_17 = ".?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?";
motif S3_14 = "....?.?.?.?.?.?.?.?.?.?.?";
 . . .

motif globin = I S0_12 II S0_4 III S0_17 IV S3_14 ...;
```

As noted earlier, the intra-motif spacers realized by the wild cards do not contribute to the score of a match to `globin`. It should further be noted that *ANREP* searches for matches to this pattern with the same efficiency as the specific algorithm described within Barton and Sternberg's paper. In other words, the approximate pattern matching algorithm at the heart of *ANREP* immediately implies their algorithmic result as a special instance.

Discussion

This paper introduces a new language, *A*, for specifying patterns ranging from very simple to very complex, and a prototype system, *ANREP*, for searching for approximate matches to these patterns. The most obvious difference to previous systems is the expressive power and versatility of the new system. While *ANREP* incorporates many features that are also found in older systems, it adds many new concepts not found elsewhere. Among these are (1) the capability of searching for approximate matches to entire networks, (2) the ability to concisely specify alphabets and scoring schemes and to attach them to different parts of a pattern, and (3) the flexibility to specify atoms via consensus symbols and positional weights.

In addition to the new language features, both the efficient approximate match search algorithm, and the optimized-backtracking search strategy have also proven to be very successful — the system is still quite fast given the complexity of the patterns it can search for. For example, interpreting the specification of the MTase pattern of the previous section and searching our 3MB version of the PIR took *ANREP* 174 seconds on a 33MHz workstation. Search speed was clocked at 54 seconds per megabyte and this speed does not vary much, even as patterns become more complex. As long as some motifs in the pattern are quite specific, the time taken doesn't even vary much with the specificity of a pattern. For example, the same search with `t = 1.0` took 168 seconds.

While most of our experience with *ANREP* points to a solid and useful system, we also found some areas where further research is necessary or desirable. First, an option to precompile

patterns and to use the resulting pattern-specific code for database searches (rather than interpreting a pattern using general-purpose code) may result in an order of magnitude speed improvement. This would be very useful in cases where large databases need to be searched for the same pattern. Search times could be reduced even further if the system could recognize and take advantage of the (frequent) special case of exact matches to motifs.

Another area of possible improvement concerns better support for frequently occurring types of motifs. While *A* allows nearly unlimited flexibility in the specification of symbols, more of this flexibility is desirable at the motif level. In particular, a quantified Kleene closure operator would be useful for internal repeats within motifs (i.e., the ability to specify a certain number or range of numbers of repeats). Other potential additions include a shorthand notation for the complement (in the sense of Watson-Crick base pairs) of a motif, patterns that match any permutation of a given set of motifs, and a pattern that matches any spacer not containing a match to a motif.

A final direction for future research, would be to couple *ANREP* with a facility which automatically infers patterns, perhaps by using the *ANREP* search engine to investigate changes in sensitivity and specificity as a potential pattern is varied among various dimensions.

References

Abarbanel, R.M., Wieneke, P.R., Mansfield, E., Jaffe, D.A., and Brutlag, D.L. (1984), ‘‘Rapid searches for complex patterns in biological molecules,’’ *Nucleic Acids Research*, Vol. 12, No. 1, pp. 263-280.

Bairoch, A., (1991) ‘‘PROSITE: a dictionary of sites and patterns in proteins,’’ *Nucleic Acids Research*, Vol. 19, Sequences Supplement, pp. 2241-2245

Barker, W.C., George, D.G., Hunt, L.T., and Garavelli, J.S. (1991) ‘‘The PIR protein sequence database,’’ *Nucleic Acids Research*, Vol. 19, Sequences Supplement, pp. 2231-2236

Barton, G.J. and Sternberg, J.E. (1990) ‘‘Flexible protein sequence patterns: A sensitive method to detect weak structural similarities,’’ *J. Mol. Biol.*, Vol. 212, pp. 389-402.

Burks, C., Cassidy, M., Cinkosky, M.J., Cumella, K.E., Gilna, P., Hayden, J.E.-D., Keen, G.M., Kelley, T.A., Kelly, M., Kristofferson, D., and Ryals, J. (1991), ‘‘GenBank,’’ *Nucleic Acids Research*, Vol. 19, Sequences Supplement, pp. 2221-2225

Dayhoff, M.O. (1978), *Atlas of Protein Sequence and Structure*, Vol. 5, Suppl. 3, National Biomedical Research Foundation, Washington, DC.

Gribskov, M., Homyak, M., Edenfield, J., and Eisenberg, D. (1988), ‘‘Profile scanning for three-dimensional structural patterns in protein sequences,’’ *CABIOS*, Vol. 4, No. 1, pp. 61-66.

Hawley, D.K., and McClure, W.R. (1983), ‘‘Compilation and analysis of Escherichia coli promoter DNA sequences,’’ *Nucleic Acids Research*, Vol. 11, No. 8, pp. 2237-2255.

Lipman, D.J. and Pearson, W.R. (1985), ‘‘Rapid and sensitive protein similarity searches,’’ *Science* Vol. 227, pp. 1435-1441.

Miller, J., McLachlan, A.D., and Klug, A. (1985), ‘‘Repetitive zinc-binding domains in the protein transcription factor IIIA from Xenopus oocytes,’’ *EMBO Journal*, Vol. 4, No. 6, pp. 1609-1614.

Myers, E.W. (1992), ‘‘Approximate Matching of Network Expressions with Spacers,’’ Technical Report TR92-5, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721.

Myers, E.W., and Miller, W. (1989), ‘‘Approximate matching of regular expressions,’’ *Bulletin of Mathematical Biology*, Vol. 51, No. 1, pp. 5-37.

Posfai, J., Bhagwat, A.S., Posfai, G., and Roberts, R.J. (1989), ‘‘Predictive motifs derived from cytosine methyltransferases,’’ *Nucleic Acids Research*, Vol. 17, No. 7, pp. 2421-2435.

Staden, R. (1988), ‘‘Methods to define and locate patterns of motifs in sequences,’’ *CABIOS*, Vol. 4, No. 1, pp. 53-60.

Stoehr, P.J. and Cameron, G.N., (1991) ‘‘The EMBL data library,’’ *Nucleic Acids Research*, Vol. 19, Sequences Supplement, pp. 2227-2230

Stormo, G.D. (1990), Consensus Patterns in DNA,’’ *Meth. in Enzymology*, Vol. 183, pp. 211-221.

Ukkonen, E. (1985), ‘‘Finding approximate patterns in strings,’’ *Journal of Algorithms*, Vol. 6, pp. 132-137.

Appendix A: Quick Reference Guide

A complete grammar for the language *A* is given in this appendix along with some terse comments and the formal definition of scoring for a consensus symbol. The aim is to give a snapshot of the entire language and to serve as a quick reminder to the experienced *ANREP* user. A complete description and an explanation of the syntactic notation used is given in Appendix B. A postscript file containing the following overview is provided with the software package.

STATEMENTS

spec → *stmt** # An *ANREP* spec is a sequence of statements

stmt	→	*comment*	*comment*	→	'`#.*\n`'
	|	*numdecl*	*numdecl*	→	**`number`** *id*{x} '`=`' *num*(x) '`;`' for x ∈ {int, float}
	|	*alpdecl*	*alpdecl*	→	**`alphabet`** *id*{alp} '`=`' *alpha* '`;`'
	|	*score*	*score*	→	**`score`** *id*{score} ['`=`' *alpha* '`{`' *matdef*+ '`}`'] '`;`'
	|	*motdecl*	*motdecl*	→	**`motif`** *id*{motif} ['`{`' *id* ('`,`' *id*)* '`}`'] '`=`' *mot* '`;`'
	|	*netdecl*	*netdecl*	→	**`net`** *id*{net} ['`{`' *id* ('`,`' *id*)* '`}`'] '`=`' *net* '`;`'
	|	*search*	*search*	→	**`search`** [*format*] *file_id* **`for`** *net* '`;`' *format* → '`\(([^)]|\\.)*\)`'
	|	*include*	*include*	→	**`include`** *file_id* '`;`'
	|	*execute*	*execute*	→	**`execute`** *command* '`;`' *command* → '`"([^"]|\\.)*"`'

IDENTIFIERS, NUMBERS, SYMBOLS, & ALPHABETS

id	→	'`[a-zA-Z_][a-zA-Z_0-9]*`'
file_id	→	'`[a-zA-Z0-9_./]+`'
symbol	→	'`[a-zA-Z]|\\.`'
alpha	→	*id*{alp} | '`[`' *symbol*+ '`]`'

Command line arguments are passed to *ANREP* by macro substitution. Every occurrence of the phrase '`@i`' is replaced by the `i`th command line argument, provided the `@`-sign is not preceded by a back-slash and `i` is in range.

num{int}	→	*id*(int) | *icon*	*num*{float}	→	*id*(float) | *fcon*
icon	→	*int* | '`[+-]`'	*fcon*	→	*int* '`[Ee]`' *int* | *rat* ['`[Ee]`' *int*]
int	→	'`[+-]?[0-9]+`'	*rat*	→	'`[+-]?([0-9]+\.[0-9]*|\.[0-9]+)`'

SCORE DECLARATIONS

matdef	→	'`<`' *set* ['`[.,]`' *set*] '`>`' '`#`' (*icon*|*fcon*) '`;`'	# enumerative spec
	|	*c_expr* '`#`' *c_expr* '`;`'	# functional spec

set	→	'`$`'	# epsilon / gap symbol
	|	*symbol*	# singleton set
	|	'`[`' (*symbol* |'`$`')+ '`]`'	# set of symbols including epsilon
c_expr	→	'`[^#;]*`'	# must be a legal C expression

MOTIFS & NETWORKS

net	→	*net* '|' *net*	# alternation (lowest precedence)
	|	*net net*	# concatenation
	|	*net* '`?`'	# zero or one (highest precedence)
	|	'`(`' *net* '`)`'	# parentheses
	|	'`{`' *mot* '`,`' *num* '`}`'	# approximate motif match
	|	'`<`' *num*(int) ['`,`' *num*(int)] '`>`'	# spacer
	|	*ref*(net)	# net reference

(continued)

mot	→	*mot* '|' *mot*	*pat*	→	*pat* '|' *pat*	# alternation (lowest precedence)
	|	*mot mot*		|	*pat pat*	# concatenation
	|	*mot* '!' *id*(score)		|	*pat* '!' *id*(score)	# local scoring scheme
	|	*mot* ':' *num*		|	*pat* ':' *num*	# positional weight
	|	*mot* '?'		|	*pat* '?'	# zero or one (highest precedence)
	|	'(' *mot* ')'		|	'(' *pat* ')'	# parentheses
	|	'"' *pat* '"'		|	*atom*	# quoted part / atomic symbols
	|	*ref*(motif)				# motif reference

ref{x} → *id*(x) ['{' *param* (',' *param*)* '}'] for x ∈ {motif, net}

param → *num* | *id*(score) | *mot* | *net*

ATOMS

atom	→	*symbol*	# ASCII symbol
	|	'.'	# wild card
	|	*consensus*	# consensus symbol

consensus → '[' ['[>~<=]'] (*literal*+ ':' *num*)* *literal*+ [':' *num*] ']'

literal	→	*symbol*	
	|	'+'	# insertion
	|	'-'	# deletion
	|	'^'	# all symbols not listed

CONSENSUS SYMBOL SEMANTICS:

Let A be the set of literals in a consensus symbol α, let $C = A - \{+, -, \hat{}\}$ be the set of symbols in α, and let w_x be the weight associated with literal x. Then the cost vectors for α are defined as follows where f is *max* for `[>...]` symbols, *min* for `[<...]`, and *average* for `[~...]`. For tailored symbols, `[=...]`, the costs are given by the second set of definitions.

Reduction Consensus Symbols:

$$\delta(\alpha, c) = \begin{cases} f_{x \in \Sigma}(\delta(x, c)(\text{if } x \in C \text{ then } w_x \text{ else } w_{\hat{}})) & \text{if } \hat{} \in A \\ f_{x \in C}(\delta(x, c) w_x) & \text{otherwise} \end{cases}$$

$$\delta(\alpha, \varepsilon) = \begin{cases} f_{x \in \Sigma}(\delta(x, \varepsilon)(\text{if } x \in C \text{ then } w_x \text{ else } w_{\hat{}})) w_- & \text{if } -, \hat{} \in A \\ f_{x \in C}(\delta(x, \varepsilon) w_x) w_- & \text{if } - \in A \text{ and } \hat{} \notin A \\ -\infty & \text{otherwise} \end{cases}$$

$$\delta(\varepsilon, c) = \begin{cases} f_{x \in \Sigma}(\delta(\varepsilon, c)(\text{if } x \in C \text{ then } w_x \text{ else } w_{\hat{}})) w_+ & \text{if } -, \hat{} \in A \\ f_{x \in C}(\delta(\varepsilon, c) w_x) w_+ & \text{if } - \in A \text{ and } \hat{} \notin A \\ -\infty & \text{otherwise} \end{cases}$$

Tailored Consensus Symbols:

$$\delta(\alpha, c) = \begin{cases} w_c & \text{if } c \in C \\ w_{\hat{}} & \text{if } c \notin C \text{ and } \hat{} \in A \\ -\infty & \text{otherwise} \end{cases}$$

$$\delta(\alpha, \varepsilon) = \begin{cases} w_- & \text{if } - \in A \\ -\infty & \text{otherwise} \end{cases}$$

$$\delta(\varepsilon, c) = \begin{cases} w_+ & \text{if } + \in A \\ -\infty & \text{otherwise} \end{cases}$$

Appendix B: Language Specification

The *ANREP* system described in the paper proper has been implemented in the form of a free-format, declarative computer language, *A*, for specifying patterns and initiating database searches. The language is strongly typed and allows for hierarchical and parameterized definition of variables of type number, alphabet, scoring scheme, motif, and network. *ANREP* provides an interactive as well as a batch-type environment for *A*, and it supports libraries of predefined subcomponents.

This appendix serves as the defining document for the language *A*. It describes the syntax using a modified BNF grammar and gives a precise statement of the semantic meaning of each construct.

The following notation is used to describe the grammar of *A*. The syntactic element being defined is listed, followed by a ''→'' and its definition. Syntactic elements listed on one line need to appear in the same order in the program; alternatives are separated by ''|''. All keywords (identifiers with reserved meaning) appear in **`Bold`** type. Individual terminal symbols are bracketed by single quotes. When a sequence of symbols appears between single quotes it is to be interpreted as a regular expression of terminal symbols. The notation for these regular expression is precisely that of the *egrep* UNIX™ tool. Non-terminals are printed in *italics*. A ''+'' after a syntactic element indicates one or more occurrences of this element, and a ''*'' means zero or more occurrences. Syntactic elements enclosed in square brackets are optional, i.e. they can appear zero or one times. Numbers and identifiers in the grammar are followed by a type in curly braces or in parentheses. Curly braces have a declarative meaning, i.e. the type is being defined, whereas parentheses have an assertive meaning, i.e. the type is required.

A specification (or program) in *A* consists of a sequence of statements. White spaces (blanks, tabs, and newlines) may appear anywhere except within keywords, identifiers, and numeric constants. Newlines do terminate only comments; all other statements are terminated by a semi-colon (''`;`'') and may be spread over several lines as per the aesthetic whim of the user. Comments may appear between any two statements in order to annotate the specification. A comment starts with a pound sign (''#''), and terminates at the end of the current line.

spec → *stmt**

stmt → *comment*
| *numdecl*
| *alpdecl*
| *score*
| *motdecl*
| *netdecl*
| *search*
| *include*
| *execute*

comment → '`#.*\n`'

Variables & Numbers

Variable names must begin with a letter or an underscore, and may be followed by any number of letters, digits, and underscores. Variable names are case-sensitive.

id → '`[a-zA-Z_][a-zA-Z_0-9]*`'

Numbers are either integer or floating point quantities. Integers are numbers without a decimal point or exponent. As a special case, ''+'' and ''-'' are interpreted as shorthand for +1 and −1, respectively. Floating point numbers contain a decimal point or an exponent (or both). Numbers can be assigned to variables via the number declaration statement.

numdecl → **`number`** *id*{x} '`=`' *num*(x) '`;`' where x ∈ {int, float}

num{int} → *id*(int)
| *icon*

num{float} → *id*(float)
| *fcon*

icon → *int*
| '[+-]'

fcon → *int* '[Ee]' *int*
| *rat* ['[Ee]' *int*]

int → '[+-]?[0-9]+'

rat → '[+-]?([0-9]+\.[0-9]*|\.[0-9]+)'

Alphabets

Alphabets are sets of symbols, where symbols can be upper or lower case letters (represented by themselves), or any other ASCII character (represented by a back slash followed by the character itself). The alphabet declaration statement is used to assign alphabets to variables.

alpdecl → **alphabet** *id*{alp} '=' *alpha* ';'

alpha → *id*{alp}
| '[' *symbol*+ ']'

symbol → '[a-zA-Z]|\\.'

Scoring Schemes

ANREP contains the concept of a current scoring scheme, which applies to all motifs that do not have a scoring scheme explicitly associated with them (see the section on *Motifs*). Initially, the current scoring scheme is the unitary scoring scheme over the alphabet of ASCII characters. Scoring schemes are defined and/or made current with the score definition statement.

score → **score** *id*{score} '=' *alpha* '{' *matdef*+ '}' ';'
| **score** *id*(score) ';'

With the first form of the score statement, an identifier of type scoring scheme is declared, initialized according to the specification to the right of the equal sign, and made the current scoring scheme. The second form of the score statement (without the declaration part), simply causes the current scoring scheme to be set to the one specified by the identifier.

matdef → '<' *set* ['[.,]' *set*] '>' '#' (*icon*|*fcon*) ';'
| *c_expr* '#' *c_expr* ';'

set → '$'
| *symbol*
| '[' (*symbol* |'$')+ ']'

c_expr → '[^#;]*'

The first (or enumerative) format explicitly specifies which entries are to be set, and their values. A declaration of this form is interpreted as follows: Let set_1 be the first set of symbols listed inside the angle brackets, and set_2 the second set of symbols (the dollar sign as a special symbol stands for ε). If set_2 is not present in the declaration, it is assumed to be Σ. Let *op* be either ',' or '.' (assumed to be ',' if not present), and *value* the value of the numeric constant to the right of the pound sign. Then, if $op = ','$, for all $a \in set_1$ and all $b \in set_2$, $\delta(a,b) = value$. If $op = '.'$, $\delta(a,b) = \delta(b,a) = value$, for all $a \in set_1$ and all $b \in set_2$.

The second (or functional) format uses Boolean conditions to specify the entries to be set. The C expression to the left of the pound sign is interpreted as a Boolean expression in variables x and y, where x represents the row (first) index and y the column (second) index into matrix δ. This Boolean expression is then applied to each element of the matrix δ. Each entry for which the expression evaluates to true (i.e., is not equal to zero), is set to the value of the second C expression (to the right of the pound sign), which may also be a function of x and y.

Atoms

An atom can be a regular ASCII symbol, the special wild card symbol ‘‘`.`’’, or a consensus symbol. The wild card symbol matches any symbol from the current alphabet and does not contribute to the score or length of a motif.

atom	→	*symbol*
	|	‘`.`’
	|	*consensus*

A consensus symbol is a user-defined atomic element, whose cost vectors are computed from the literals involved in its definition. Consensus symbols are specified in a syntactic form similar to alphabets:

consensus	→	‘`[`’ [‘`[>~<=]`’] (*literal*+ ‘`:`’ *num*)* *literal*+ [‘`:`’ *num*] ‘`]`’

literal	→	*symbol*
	|	‘`+`’
	|	‘`-`’
	|	‘`^`’

The (optional) special symbol following the left square bracket indicates how the cost vectors are to be computed from the literals and their weights. A ‘‘`>`’’ stands for a maximum reduction, a ‘‘`~`’’ for an average reduction, a ‘‘`<`’’ for a minimum reduction, and an ‘‘`=`’’ means that the specification is to be taken literally — no transformation is applied. A missing special symbol denotes a symbol class and is equivalent to an explicitly specified maximum reduction. Following the special symbol, all literals involved in the definition of the consensus symbol are listed. A literal can be either a regular symbol (which must be defined in the current scoring scheme), or it can be one of the three special symbols ‘‘`+`’’, ‘‘`-`’’, and ‘‘`^`’’, which stand for insertion, deletion, and substitution of all symbols not specified explicitly, respectively. Individual literals as well as groups of literals can be given a weight by appending a colon and the desired weight.

To describe the computation of the edit costs (cost vectors) for a consensus symbol α, let A be the set of literals between the square brackets, and let w_x be the weight associated with literal x. For example, for `[ag:3 t:-1 +]`, $A = \{a, g, t, +\}$ and $w_a = w_g = 3$, $w_t = -1$, and $w_+ = 1$. In addition, let $C = A - \{+, -, \hat{}\}$, i.e., the set of all regular symbols in the specification. Substitution, deletion, and insertion costs are then defined for the case of a reduction function f as follows:

$$\delta(\alpha, c) = \begin{cases} f_{x \in \Sigma}(\delta(x, c)(if\ x \in C\ then\ w_x\ else\ w_{\hat{}})) & \text{if } \hat{} \in A \\ f_{x \in C}(\delta(x, c) w_x) & \text{otherwise} \end{cases}$$

$$\delta(\alpha, \varepsilon) = \begin{cases} f_{x \in \Sigma}(\delta(x, \varepsilon)(if\ x \in C\ then\ w_x\ else\ w_{\hat{}}))\, w_- & \text{if } -, \hat{} \in A \\ f_{x \in C}(\delta(x, \varepsilon) w_x) w_- & \text{if } - \in A \text{ and } \hat{} \notin A \\ -\infty & \text{otherwise} \end{cases}$$

$$\delta(\varepsilon, c) = \begin{cases} f_{x \in \Sigma}(\delta(\varepsilon, c)(if\ x \in C\ then\ w_x\ else\ w_{\hat{}}))\, w_+ & \text{if } -, \hat{} \in A \\ f_{x \in C}(\delta(\varepsilon, c) w_x)\, w_+ & \text{if } - \in A \text{ and } \hat{} \notin A \\ -\infty & \text{otherwise} \end{cases}$$

In the case of a tailored consensus symbol, these costs are defined as:

$$\delta(\alpha, c) = \begin{cases} w_c & \text{if } c \in C \\ w_{\hat{}} & \text{if } c \notin C \text{ and } \hat{} \in A \\ -\infty & \text{otherwise} \end{cases}$$

$$\delta(\alpha, \varepsilon) = \begin{cases} w_- & \text{if } - \in A \\ -\infty & \text{otherwise} \end{cases}$$

$$\delta(\varepsilon, c) = \begin{cases} w_+ & \text{if} + \in A \\ -\infty & \text{otherwise} \end{cases}$$

Motifs

Motifs are regular expressions built from atoms and variables of type motif. Motifs are assigned to variables via the motif declaration statement. A motif declaration may be parameterized. Parameters serve as placeholders for actual values that are filled in when the motif is ''called'', or referenced.

motdecl → **`motif`** *id*{motif} ['`{`' *id* ('`,`' *id*)* '`}`'] '`=`' *mot* '`;`'

Motif regular expressions are built from atoms and references to other variables of type motif, using the three operators alternation (''`|`''), concatenation (juxtaposition), and optionality (''`?`''). A particular scoring scheme can be applied to a specific motif by appending an exclamation mark and an identifier of type scoring scheme to the motif. To express the importance of a particular motif relative to other motifs in the same pattern, a motif may be followed by a colon and a weight. Nested weights accumulate multiplicatively. The order of precedence is ''`|`'' (lowest), concatenation, and ''`!`'', ''`:`'', and ''`?`'' (highest). Parentheses may be used for grouping and to force a different order of precedence. In order to resolve the possible syntactic confusion between an identifier and a sequence of regular symbols, atoms must be enclosed in double quotes (''`"`''). Rather than suffer the burden of having to quote each atom, one can quote an entire expression, as long as it does not involve a motif reference. Often, this means that the entire motif may be placed in a single pair of quotes. In order to describe this formally, a *mot* below is a regular expression of motif references and *pat*s enclosed in quotes. A *pat*, in turn is a regular expression of atoms. The parallel rules a given side-by-side to help see the differences.

mot	→	*mot* '`|`' *mot*	*pat*	→	*pat* '`|`' *pat*
	|	*mot mot*		|	*pat pat*
	|	*mot* '`!`' *id*(score)		|	*pat* '`!`' *id*(score)
	|	*mot* '`:`' *num*		|	*pat* '`:`' *num*
	|	*mot* '`?`'		|	*pat* '`?`'
	|	'`(`' *mot* '`)`'		|	'`(`' *pat* '`)`'
	|	'`"`' *pat* '`"`'		|	*atom*
	|	*ref*(motif)			

The actual parameters used in a motif reference can be numbers, motifs, networks, or scoring scheme identifiers. The type of the actual parameter must match the type of the formal parameter in the motif declaration. The type of the formal is inferred from its use in the relevant declaration and must be consistent throughout. Note that it then follows that in the case of motifs, it is actually not possible for a parameter to be of type network. However, we include the more general definition so that it need not be repeated for the treatment of networks below.

ref{x} → *id*(x) ['`{`' *param* ('`,`' *param*)* '`}`'] where x ∈ {motif, net}

param → *num* | *id*(score) | *mot* | *net*

Networks

A variable of type network is declared and assigned to with the network declaration statement. Like a motif declaration, a network declaration may be parameterized.

netdecl → **`net`** *id*{net} ['`{`' *id* ('`,`' *id*)* '`}`'] '`=`' *net* '`;`'

Network regular expressions are built from approximate matches to motifs, spacers, and references to other networks, using the three operators (in order of increasing precedence) alternation (''`|`''), concatenation, and optionality (''`?`''). Parentheses may be used for grouping and to force a different order of precedence. An approximate match to a motif is a motif preceded by an opening curly brace, followed by a comma, a threshold, and a closing curly brace. A spacer is an integer or a pair of integers enclosed in angle-brackets.

net	→	*net* '|' *net*
	|	*net net*
	|	*net* '?'
	|	'(' *net* ')'
	|	*apmat*
	|	*spacer*
	|	*ref*(net)
apmat	→	'{' *mot* ',' *num* '}'
spacer	|	'<' *num*(int) [',' *num*(int)] '>'

A motif in a network specification must match with a score greater than or equal to the specified threshold during the search of a database, in order for the whole network to match. The score of matching a motif to a target sequence is the score of the highest scoring alignment between some sequence denoted by the motif and the target sequence, divided by the length of the motif sequence used to make the alignment. A spacer $< i , j >$ between two networks M and N serves as a separator. Intuitively, it matches any sequence of symbols whose length is between i and j. Assuming both i and j are positive, it asserts that the right end of a match to network M must be at least i and at most j symbols to the right of a match to the left end of network N. However, spacers may also contain negative numbers and these are interpreted as distances to the left. The notation $< i >$ is accepted as shorthand for $< i , i >$. Spacers at the beginning or the end of a network are ignored.

Searching

A search for a network over a database is initiated with the search statement. The file identifier is the name of the file containing the database. The optional format is a string between parentheses that is passed to the I/O-interface routine `Start_scan` just prior to the start of the search. See the section "System and Methods" for a precise description of the I/O library and supported formats. *ANREP* currently understands the two formats, `PLAIN` and `FASTA`, where the former is the default when a format is not given.

search	→	**`search`** [*format*] *file_id* **`for`** *net* ';'
format	→	'\(([^)]|\\.)* \)'
file_id	→	'[a-zA-Z0-9_./]+'

The database is searched according to the algorithms described previously. All matches to the network specification contained in the database are reported, subject to two restrictions: (1) The maximum length of a match is limited by the size of an internal buffer, and (2), if the overlap between two adjacent matches exceeds some percentage of the total span of the union of both matches, only the first match found is reported. However, both the maximum length of a match as well as the overlap percentage are compilation constants. They can easily be adjusted to suit any particular need.

Libraries and Command-Line Arguments

To facilitate libraries of predefined components, the include statement is provided. When an include statement is encountered, the contents of the file specified by the file identifier are read in and processed as if they were typed on-line — with one exception: if an error is encountered, processing skips everything following the offending statement until the end of the included file. Include statements may be nested.

include	→	**`include`** *file_id* ';'

A generalization of the include statement is the execute statement, which passes the command string between double-quotes to the operating system as if it had been typed at a terminal. The standard output produced by executing the command is captured and processed by *ANREP* exactly as if it had been included. For example, the statement, `execute "cat foo"`, is equivalent to, `include foo`.

execute	→	**`execute`** *command* ';'
command	→	'"([^"]|\\.)* "'

At the command-line level of the operating system one invokes *ANREP* by typing `anrep` followed by a number of command-line arguments in the usual UNIX convention. These arguments are passed to *ANREP* as strings. Any occurrence within the *A* specification read from the standard input of a phrase of the form `@i` where `i` is an integer, is assumed to be a reference to the i^{th} command-line argument and that string replaces the phrase `@i`. Such an argument reference may occur anywhere in a specification, provided the `@`-sign is not preceded by a backslash and there is an i^{th} command-line argument.