APPROXIMATE MATCHING OF NETWORK EXPRESSIONS WITH SPACERS

□ Eugene W. Myers*
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Two algorithmic results are presented that are pertinent to the matching of patterns typically used by biologists to describe regions of macromolecular sequences that encode a given function. The first result is a threshold-sensitive algorithm for approximately matching both network and regular expressions. Network expressions are regular expressions that can be composed only from union and concatenation operators. Kleene closure (i.e., unbounded repetition) is not permitted. The algorithm is threshold-sensitive in that its performance depends on the threshold, *k*, of the number of differences allowed in an approximate match. This result generalizes the *O(kn)* expected-time algorithm of Ukkonen for approximately matching keywords. The second result concerns the problem of matching a pattern that is a network expression whose elements are approximate matches to network or regular expressions interspersed with specifiable distance ranges. For this class of patterns, it is shown how to determine a backtracking procedure whose order of evaluation is optimal in the sense that its expected time is minimal over all such procedures.

1. Introduction. Many patterns of interest to molecular biologists investigating the structure of proteins and their binding to nucleic acid sequences, take the form of a number of "domains" or "signals" distributed at various locations along the sequence in question, e.g., Miller *et al.* (1985) and Posfai *et al.* (1989). Both the spacing and the domains vary somewhat between sequences manifesting the behavior or function. This motivates a class of patterns in which one is searching for approximate matches to a consensus sequence for each domain, separated within certain distance ranges of each other. For example, in a software system called *ANREP* (Myers and Mehldau 1993), the specification for a cytosine methyltransferase pattern developed by Posfai *et al.* (1989) is shown in Figure 1. The pattern (or "net") `MTase` consists of ten domains (or "motifs"), `I` through `X`, separated by "spacers" of varying sizes. For example, the spacer `<-9,39>` between motifs `I` and `II` indicates that the left end of a match to motif `II` must be found between 9 symbols to the left and 39 symbols to the right of the right end of a match to motif `I`. The search requested in the last statement of the specification, scans the PIR database for a match to `MTase` with the parameter `t` set to `.8`, which requests that each motif match over

* This author’s work was supported in part by National Library of Medicine Grant R01-LM4960 and the Aspen Center for Physics.

at least 80% of its length. The motifs themselves are sequences of symbols, symbol classes (e.g., `[ILM]` which matches `I`, `L`, or `M`), and wild-cards (i.e., `.` which matches any symbol).

```
# 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.
```

Figure 1. An *ANREP* pattern specification.

A reader interested in the practical engineering issues and the complete range of capabilities of *ANREP* are referred to a companion paper (Myers and Mehldau 1993). The focus of this paper is on the algorithms for the formal discrete pattern matching problem that lies at the heart of the *ANREP* system. This introduction presents the essential background concepts and a formal definition of the two-tiered class of patterns for which we present algorithmic results. These algorithmic results have been implemented and form the core of *ANREP*'s implementation. The two-tiered pattern class considered here was formulated in direct response to the range of patterns posited in the current molecular biology literature as exemplified by Figure 1. However, it should be noted in passing that the pattern class could also be applied to other domains such as information retrieval and speech recognition.

To begin a formal treatment, one needs the concept of an approximate match to a pattern (say, a regular expression) R over alphabet Σ. But this requires introducing first the much studied concept of an alignment and its score. Given sequences $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$ over alphabet Σ, an alignment between them is a sequence of pairs $(i_1, j_1), (i_2, j_2), \cdots (i_{len}, j_{len})$ such that $i_k < i_{k+1}$ and $j_k < j_{k+1}$. This *trace* aligns a_{i_k}

with b_{j_k} for each k and if one imagines drawing lines between aligned symbols, then the condition on indices implies the lines do not cross. Observe that there are a tremendous number of distinct alignments between A and B. What is desired are those alignments that are optimal with respect to some criterion. To do so, introduce scoring scheme $\delta(a, b)$ which is a function giving a *non-negative* real-valued score for each pair of symbols a and b from $\Sigma \cup \{\varepsilon\}$. For $a, b \in \Sigma$, $\delta(a, b)$ gives the score of aligning a with b; $\delta(\varepsilon, b)$ is the score of leaving b unaligned in sequence B; and $\delta(a, \varepsilon)$ is the score of leaving a unaligned in sequence A. The score of an alignment is the sum of the scores assigned by δ to its aligned pairs and unaligned symbols. An optimal alignment is one of minimal score. Finding an optimal alignment and its cost, $\delta(A, B)$, is a much studied problem solvable with a dynamic programming algorithm in $O(mn)$ time (Levenshtein 1966, Needleman and Wunsch 1970, Wagner and Fischer 1974).

A pattern R, such as a regular expression, is formally a specification of a set (potentially infinite) of sequences, i.e., the language $L(R)$. From another perspective these are the sequences exactly matched by the pattern. With this view one can think of a sequence that aligns particularly well with a sequence exactly matched by R, as approximately matching R. Formally, the set of sequences approximately matching R within threshold T under scoring scheme δ is $L_\delta(R, T) = \{ A : \exists B \in L(R), \ \delta(A, B) \le T \}$. The problem of approximately matching patterns arises naturally in the context of pattern matching for biological sequences because evolutionary pressures mutate any given precursor over time. Myers and Miller (1989) presented an $O(np)$ algorithm for approximately matching sequence A to regular expression R where p is the length of R. A regular expression is any pattern that can be built up from symbols via concatenation, union, and/or unbounded repetition (formally known as Kleene closure). However, in most applications in molecular biology the Kleene closure operator is not useful, thus motivating the definition of a *network expression* as a regular expression not containing a Kleene closure. Thus in direct terms, a network expression is any pattern built up from concatenation and union operations. From here forward, this paper focuses on network expressions and only digresses in one paragraph of Section 3 to show that the result there is applicable to regular expressions. We do so primarily because the motivating application, *ANREP*, only requires networks, but also because restricting attention to networks simplifies much of what follows. Indeed, approximately matching networks is not difficult (as will be seen momentarily) and the first algorithm for doing so is attributable to Sankoff and Kruskal (1983, pp. 265-310).

Hereafter, a network expression when coupled with a threshold will be termed a *motif*. It should be noted, however, that the term "motif" has been used broadly in the computational molecular biology literature to denote a pattern of some type. For example, the Gribskov profiles (Gribskov *et al.* 1988) and Staden weight matrices (Staden 1988) have been called motifs along with several other pattern classes (e.g., Abarbanel *et al.* 1984, Saurin and Marliere 1987, Bairoch 1991). In the companion paper on ANREP (Myers and Mehldau 1993) we showed that a simple extension of the concept of a scoring scheme given above, allows the motifs of this paper to encompass and generalize many of these other pattern classes. In any event, for this paper we use the term motif to denote a network expression, threshold pair.

Now consider the composite problem of matching a pattern consisting of several motifs separated by specifiable distance ranges or *spacers*. Formally, let a *net N* be a network expression over the (infinite) alphabet of motifs, $\{R:T\}$, and spacers, $[l,\ r]$. The pair $\{R:T\}$ denotes an approximate match within threshold T of network expression R where an implied alphabet Σ and scoring scheme δ will be assumed to apply to all motifs for simplicity. The spacer $[l,\ r]$ matches any sequence of between l and r symbols. The integers may be negative in which case the spacer indicates that the left end of the item after it must begin so many characters to left of the right end of the preceding item. For example, the pattern $\{A:2\}\ (\ [0,\ 20]\ \{B:4\} \mid [-5,\ 5]\ \{C:1\}\ )$, matches an approximate match to network expression A, either followed zero to twenty symbols later by a fairly loose match to B, or followed within five symbols to the left or right by a more stringent match to C. The bar denotes union (alternation), juxtaposition denotes concatenation, and parentheses may be used to enforce an arbitrary order of precedence. Proceeding more formally, sequence $A = a_1 a_2 \cdots a_n$ over alphabet Σ is said to match net N, written $A \sim N$, if and only if there exists index sequence $i_{0,}\ i_1,\ \cdots\ i_p$ and sequence $W = w_1 w_2 \cdots w_p$ over the alphabet of motifs and spacers such that (1) $W \in L(N)$, (2) $i_0 = 0$ and $i_p = n$, and (3) if w_k is motif $\{R:T\}$ then $a_{i_{k-1}+1} a_{i_{k-1}+2} \cdots a_{i_k} \in L_\delta(R,\ T)$, and if w_k is spacer $[l,\ r]$ then $i_k - i_{k-1} \in [l,\ r]$. As noted earlier, this two-tiered problem of matching network expressions of motifs and spacers is a formal embodiment of the pattern matching capability built into the *ANREP* software system for the analysis of biosequences (Myers and Mehldau 1993).

In this paper, two results of algorithmic interest for the problem of matching nets are presented. The first is a threshold-sensitive algorithm for matching motifs that generalizes Ukkonen's $O(kn)$ algorithm for approximately matching keywords (Ukkonen 1985). This is presented in Section 3, after Section 2 which reviews the traditional solution to this problem cast in an automata-theoretic form required for our second result presented in Section 4. This second result is a backtracking algorithm for matching net patterns that picks a backtracking order that minimizes the expected time spent searching for a match.

2. A Review of Approximately Matching Network Expressions. A network expression over alphabet Σ is any expression built up from the symbols in $\Sigma \cup \{\varepsilon\}$ with the operations of concatenation (juxtaposition) and alternation (|). The symbol ε matches the empty string. For example, $a(bc \mid \varepsilon)d$ denotes the set $\{ad,\ abcd\}$. While an expression is a convenient textual representation of a network, a graph theoretic, finite automaton formulation is better suited to our purpose of approximately matching networks.

There are several different models of finite automata to choose from (Hopcroft and Ullman 1979, pp. 13-76). The non-deterministic, state-labeled, finite automaton model is used here and will be referred to as an ε-NFA. Formally, an *ε-NFA*, $F = \langle\ V,\ E,\ \lambda,\ \theta,\ \phi\ \rangle$, consists of: (1) a set, V, of vertices, called *states*; (2) a set, E, of directed edges between states; (3) a function, λ, assigning a "label" $\lambda_s \in \Sigma \cup \{\varepsilon\}$ to each state s; (4) a designated "source" state, θ; and (5) a designated "sink" state, ϕ. Intuitively, F is a vertex-labeled directed graph with distinguished source and sink vertices. We will use the notation $t \rightarrow s$ to denote that there is an edge in F from state t to state s. A directed path through F *spells* the sequence obtained by concatenating the non-ε state labels along the path. $L_F(s)$, the *language accepted at* $s \in V$, is the set of sequences spelled on paths from θ to s. The *language accepted by F* is $L_F(\phi)$.

Any network expression, R, can be converted into an equivalent finite automaton F with the inductive construction depicted in Figure 2. For example, the figure shows that F_{RS} is obtained by constructing F_R and F_S, adding an edge from ϕ_R to θ_S, and designating θ_R and ϕ_S as its source and sink states. After inductively constructing F_R, an ε-labeled start state is added as shown in the figure to arrive at F. This last step guarantees that the sequence spelled by a path is the sequence of symbols *at the head of each edge*, and together with the choice of finite automaton model is essential for the upcoming alignment graph construction.

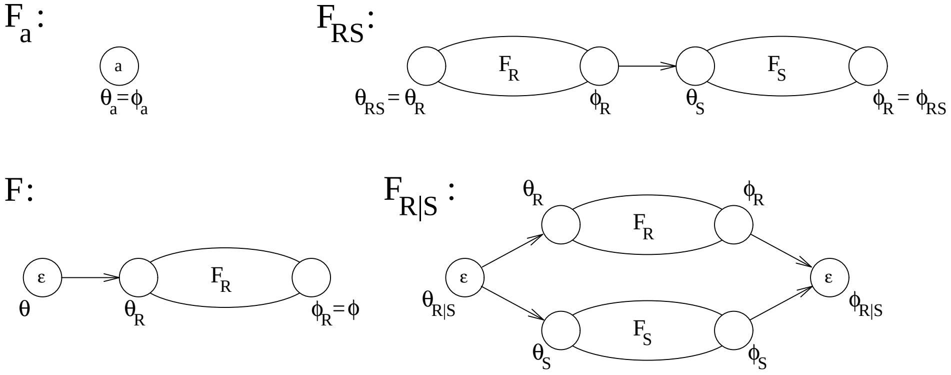

Figure 2. Constructing the ε-NFA for network expression R.

A straightforward induction shows that automata constructed for network expressions by the above process have the following properties: (1) every state has an in-degree and an out-degree of 2 or less; and (2) $|V| \leq 2|R|$, i.e., the number of states in F is less than twice R's length. That is, for any network expression, there is an equivalent ε-NFA whose size, measured in vertices or edges, is linear in the length of R. Another property of F's graph is that it is acyclic and so its states can be *topologically* ordered. An ordering of the states is said to be a topological order if and only if for every state, s, its predecessors in R come before s in the ordering. Finally, of essential importance to the threshold-sensitive algorithm of Section 3, is the fact that F is a *series/parallel* graph.

To arrive at the basic dynamic programming algorithm for approximately matching network expression R with sequence $A = a_1 a_2 \cdots a_n$, it is easiest to reduce the problem to one of finding a shortest source-to-sink path in a weighted and directed *alignment graph* constructed from R and A (Myers and Miller 1989). The vertices of the alignment graph consist of $n+1$ copies of F, the ε-NFA for R, arranged one on top of another as shown in Figure 3. Formally, the vertices are the pairs (i, s) where $i \in [0, n]$ and $s \in V$. For every vertex (i, s) there are up to five edges directed into it. (1) If $i > 0$, then there is a *deletion* edge from $(i-1, s)$ that models leaving a_i unaligned and its weight is $\delta(a_i, \varepsilon)$. (2) If $s \neq \theta$, then for each state t such that $t \rightarrow s$, there is an *insertion* edge from (i, t) that models leaving λ_s unaligned (in whatever sequence of $L(R)$ that is being spelled) and its weight is $\delta(\varepsilon, \lambda_s)$. (3) If $i > 0$ and $s \neq \theta$, then, for each state t such that $t \rightarrow s$, there is a *substitution* edge from $(i-1, t)$ that models aligning a_i with λ_s and its weight is $\delta(a_i, \lambda_s)$. An exercise in induction reveals that the construction is such that every path from (i, t) to (j, s) models an alignment between $a_{i+1} a_{i+2} \cdots a_j$ and the sequence spelled on the heads of the edges in the path from t to s in F that is the "projection" of the alignment graph path. The mapping of paths to alignments is not one-to-one since substitutions into ε-labeled states have the redundant effect of leaving a_i unaligned, and insertion edges into ε-states redundantly align ε with ε. However, as long as one defines $\delta(\varepsilon, \varepsilon) = 0$, then the cost of paths and their alignments coincide. Moreover, every alignment is modeled by at least one path. Thus the problem of approximately matching A to R is equivalent to finding a least cost path between source vertex $(0, \theta)$ and sink vertex (n, ϕ). It can be further shown that all substitution and deletion edges entering ε-labeled vertices except θ can be removed without destroying the property of there being a path corresponding to every alignment. These edges are removed in the example of Figure 3 to avoid cluttering the graph. In Figure 3, solid circles denote ε-labeled states and unlabeled edges have weight $\delta(\varepsilon, \varepsilon) = 0$.

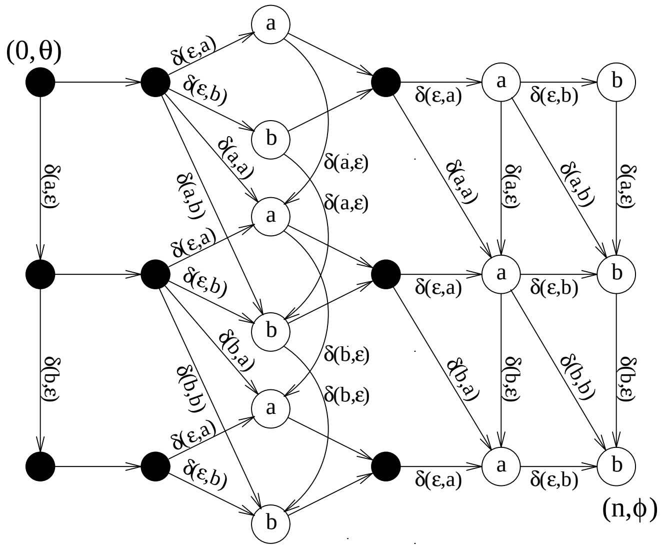

Figure 3. The alignment graph for $A = ab$ versus $R = (a|\, b)\, ab$.

Any alignment graph is easily seen to be acyclic since F is acyclic. Thus one can readily formulate the following central recurrence for the least path cost, $C(i, s)$, from $(0, \theta)$ to vertex (i, s):

$$C(i,s) = \min \{ \min_{t \rightarrow s} \{ C(i-1,t) + \delta(a_i, \lambda_s) \}, \min_{t \rightarrow s} \{ C(i,t) + \delta(\varepsilon, \lambda_s) \}, C(i-1,s) + \delta(a_i, \varepsilon) \} \qquad (2.1)$$

For the "boundary" vertices with $i = 0$ or $s = \theta$, the terms that are undefined should be omitted, and for the source vertex, $C(0, \theta) = 0$. By construction of the alignment graph, $C(i, s) = \min \{ \delta(A_i, B) : B \in L_F(s) \}$, the score of the best alignment between a sequence in $L_F(s)$ and the prefix $A_i = a_1 a_2 \cdots a_i$ of A. Assuming the length of R is p then there are $O(np)$ vertices in the alignment graph and the in-degree of each is less than 5. Thus by applying the dynamic programming paradigm one can compute $C(i, s)$ for every vertex in increasing order of i and any topological order of V. Computing the value of each vertex using the recurrence above takes $O(1)$ time given the value of its immediate predecessors in the graph. Thus $C(n, \phi) = \min \{ \delta(A, B) : B \in L(R) \}$, the score of the best alignment between A and a sequence in $L(R)$, can be computed in $O(np)$ time.

Thus far the problem under consideration has been that of determining the score of the best alignment between A and R. For a given threshold T, a solution to this basic problem suffices to allow us to determine if $A \in L_\delta(R, T)$ since the predicate is true if and only if $C(n, \phi) \le T$. But in general, one is faced with a very large text — n a million or more — and one is interested in finding those *substrings* of A that approximately match motif $M = \{R{:}T\}$. As Sellers (1980) noticed a decade ago, it suffices to simply modify the boundary of the central recurrence (2.1) so that $C(i, \theta) = 0$ for all i. This is tantamount to making every θ-vertex a source vertex as opposed to just $(0, \theta)$, and has the consequence that now $C(i, s) = \min \{ \delta(A_{j..i}, B) : j \le i \text{ and } B \in L_F(s) \}$ where $A_{j..i}$ denotes the substring $a_{j+1} a_{j+2} \cdots a_i$ of A (ε if $j = i$). It then follows that there exists a suffix of A_i matching R within threshold T if and only if $C(i, \phi) \le T$. In such an instance the index i is termed the right end of a match.

In applications where n is very large, it is prohibitive to use $O(np)$ space for the quantities $C(i, s)$. Let C_i denote the "row" of entries $\{C(i, s)\}_{s \in V}$ and observe from the central recurrence that row C_i can be computed given just C_{i-1} and a_i. Thus it is possible using only $O(p)$ space to scan A from left-to-right computing C_i at each index and asking if i is the right end of an approximate match. In direct analogy with the state-set simulation of an ε-NFA on a text string, one can think of the C_i's as modeling the states of a deterministic automaton where on symbol a_i the machine transits from state C_{i-1} to state C_i. It is impractical to actually build the automaton since it has an exponential number of states (infinite if δ is irrational). However, it is useful to think in terms of scanning A with a finite automaton recognizing motif M, for then the issues of how to report matches (e.g., left-most longest) are identical to those already studied for the matching of regular expressions. This will be explored further in Section 4.

The concepts expounded to this point are summarized in the pseudo-code of Figure 4. Within the code, all attributes of a motif M are prefixed by "$M.$", e.g., $M.V$ is the state set of M's ε-NFA, $M.\theta$ is its start state, and so on. Also associated with M are two arrays $M.C$ and $M.D$ indexed by the states in $M.V$. These $O(p)$ arrays are used to store "rows" of C-entries. The array $M.C$ contains the current "state" of M, and $M.D$ is used temporarily by the function *Advance* to compute a next state from the current one. The procedure *Start*(M) initializes M's state to C_0. Each call to *Advance*$(M, a, inject)$ with *inject* set to *true*, updates $M.C$ from its current state (row of C-entries) to the one obtained after scanning the symbol a. That is, if $M.C = C_{i-1}$ upon invocation, then $M.C = C_i$ after invoking *Advance*$(M, a_i, true)$. Also note that *Advance* returns *yes* if and only if $M.C[\phi] \le M.T$, i.e., one has just scanned to the right end of an approximate match. Until the next paragraph, ignore the distinction between the *no* and *never* values returned by *Advance*, and also what the function does when *inject* is *false*. The procedure *Scanall* uses *Start* and *Advance* to compute C_i for increasing values of i and reports the right end of matches as they are encountered. It thus realizes the algorithm developed in the previous paragraph.

Consider the following final problem: given a set of indices $J \subseteq [0, n]$ of potential left ends, determine the set $Scan(M, J) = \{ i : \exists j \in J \ s.t. \ A_{j..i} \in L_\delta(R, T) \}$ of right ends of approximate matches to M having a left end in J. Generalizing the previous discussion, what suffices is to modify the boundary conditions of the central recurrence (2.1) so that $C(j, \theta)$ is set to 0 if and only if $j \in J$. This has the effect that now $C(i, s) = \min \{ \delta(A_{j..i}, B) : j \in J \text{ and } B \in L_F(s) \}$, and so the problem is solved by finding all i such that $C(i, \phi) \le T$ under this modification of the recurrence's boundary. Suppose that $J = \{ j_1, j_2, \cdots j_c \}$ where $j_1 < j_2 < \cdots < j_c$, i.e., the elements are sorted. A simple approach is to modify *Scanall* so that *inject* is set to *true* only when $i \in J$, i.e., the value 0 is "injected" into the θ value of the next state only when an index in J is traversed during the scan of A. However, efficiency can be improved in two simple ways: (1) start scanning at j_1, and (2) stop the scan when j_c has been passed and a right end is no longer possible. If the value of every vertex in M's current state is greater than threshold, then *Advance* returns *never* instead of just *no*. Because δ is non-negative, it follows that every future state of M

```
procedure Start(M)
  {  $M.C[M.\theta] \leftarrow 0$
     for $s \in M.V - M.\theta$ in topological order do
          $M.C[s] \leftarrow \min_{t \to s} \{ M.C[t] + \delta(\varepsilon, \lambda_s) \}$
  }

function Advance(M, a, inject): (yes, no, never)
  {  $M.D[M.\theta] \leftarrow M.C[M.\theta] + \delta(a, \varepsilon)$
     if inject then
          $M.D[M.\theta] \leftarrow 0$
     for $s \in M.V - M.\theta$ in topological order do
          $M.D[s] \leftarrow \min \{ \min_{t \to s} \{ M.C[t] + \delta(a, \lambda_s) \}, \min_{t \to s} \{ M.D[t] + \delta(\varepsilon, \lambda_s) \}, M.C[s] + \delta(a, \varepsilon) \}$
     $M.C \leftarrow M.D$
     if $\forall s, M.C[s] > M.T$ then
          return never
     else if $M.C[M.\phi] > M.T$ then
          return no
     else
          return yes
  }
```

```
procedure Scanall(M)
  {  Start(M)
     for $i \leftarrow 1$ to $n$ do
          if Advance($M, a_i$, true) = yes then
               print $i$ · "is a match right end"
  }
```

```
function Scan(M, J): set of [1..n]
  {  $I \leftarrow \emptyset$
     $k \leftarrow 2$
     Start(M)
     for $i \leftarrow j_1 + 1$ to $n$ do
       {  $answer \leftarrow Advance(M, a_i, i = j_k)$
          if $k \le |J|$ and $i = j_k$ then $k \leftarrow k + 1$
          if $answer = yes$ then $I \leftarrow I \cup \{i\}$
          if $answer = never$ and $k > |J|$ then break
       }
     return I
  }
```

Figure 4. Scanning Routines for Matching Motifs.

will satisfy this condition unless a 0 is injected at some future point. Thus in the procedure *Scan* of Figure 4 that implements the reduced scan, condition (2) is detected when *never* is returned by *Advance* after the last possible injection has taken place. While *Scanall* scans $O(n)$ characters, *Scan* scans only $O(j_c - j_1 + p)$ provided the insertion and deletion costs of the scoring scheme δ are bounded away from 0. If the range of indices covered by J is small, as it generally is in the upcoming problem of Section 4, this refinement is particularly important.

3. A Threshold-Sensitive Motif Matching Algorithm. Several authors have observed that in the case of a thresholded problem such as matching a motif, one need not compute every value $C(i, s)$, but simply those that are within the threshold T in question. Generally a few more entries than just those desired must be computed in order to ensure that none are missed, but this is acceptable provided the "zone" examined is on the order of the number of vertices whose least path cost is not greater than T. Fickett (1984) presented such an algorithm for sequence versus sequence comparison under non-negative δ. Ukkonen (1985) presented an $O(kn)$ expected-time algorithm for approximate keyword matching where $T = k$ errors are allowed under the unit cost model: $\delta(x, y) \equiv$ **if** $x = y$ **then** 0 **else** 1. Note that approximate keyword matching is just a special case of network matching where F is a line graph or chain. The complexity of Ukkonen's algorithm depends on the fact that the expected number of vertices in each row C_i whose value is within threshold T is $O(T)$. Because performance depends on the parameter T, the algorithm is termed threshold-sensitive. The tighter (smaller) the threshold, the faster the algorithm performs. Ukkonen's algorithm easily generalizes to any non-negative δ, but it becomes difficult to characterize the expected size of the zone computed in each row. Nonetheless, this treatment will continue to adhere to the threshold-sensitive characterization since performance depends primarily on the stringency of the required match and not on the size of the pattern. In this section a zone or threshold-sensitive algorithm for approximate network expression matching under non-negative δ is developed.

Let $C_{i-1}(s)$ be the value of vertex $(i-1, s)$ in whatever row, C_{i-1}, a motif recognizer finds itself in after scanning the $i-1^{st}$ text symbol a_{i-1}. Let $T_{i-1} = \{ s : C_{i-1}(s) \leq T \}$ be the set of values in row C_{i-1} that are within the threshold T. Suppose at this point that one has somehow managed to arrive at a set $Z_{i-1} \supseteq T_{i-1}$ and values $C^*_{i-1}(s)$ for $s \in Z_{i-1}$ such that if $C_{i-1}(s) \leq T$ then $C^*_{i-1}(s) = C_{i-1}(s)$ and $C^*_{i-1}(s) > T$ otherwise. Such a set and its values, denoted Z_{i-1}/C^*_{i-1}, is called a *zone* and it correctly models all the values of C_{i-1} within T. The goal is to advance the motif recognizer over symbol a_i to its next state, C_i, but only computing enough of this new row to arrive at a new zone, Z_i/C^*_i, encompassing the values within threshold T. Of course the challenge is to keep each zone as small as possible (a trivial solution would be to let $Z_i = V$). It would be ideal if Z_i were T_i but it does not seem to be possible to compute these "perfect" zones at a cost of $O(1)$ time per zone vertex. We can however achieve this computational rate if Z_i is relaxed to a superset of T_i such that (1) the subgraph of F restricted to the vertices in Z_i is connected, and (2) the removal of any vertex from $Z_i - T_i$ disconnects this subgraph. As will be seen later, the connectedness property (1) is required to efficiently maintain a topological ordering of zone vertices and associated "dominator" information. The minimality condition of property (2) is weaker than requiring Z_i's cardinality to be minimal, but again seems to be the best compromise attainable subject to the constraint that the computation of Z_i take time $O(Z_i)$. We term a zone satisfying (1) and (2) as being a *minimally connected* zone.

Given a minimally connected zone Z_{i-1}/C^*_{i-1} and symbol a_i our problem is to compute a minimally connected zone Z_i/C^*_i modeling C_i. From the structure of the alignment graph it follows that the least cost path to a vertex in C_i within threshold T must consist of a deletion or substitution edge from a vertex $(i-1,s)$ where $s \in Z_{i-1}$ followed by a possibly empty series of insertion edges in row *i*. More formally, the zone $U_i = Z_{i-1} \cup Inserts(Z_{i-1})$ is guaranteed to be a superset of T_i where $Inserts(X) = \{\ s : \exists t \in X, t \rightarrow s$ or $\exists\ t \in Inserts(X), (t \rightarrow s$ and $C_i(t) \leq T)\ \}$. Certainly U_i is connected. If one computes C^*_i over the states in U_i by applying the standard recurrence (2.1) over just the edges of the alignment graph between vertices in $\bigcup_{s \in Z_{i-1}} (i-1,s) \cup \bigcup_{s \in U_i} (i,s)$, then C^*_i will properly model C_i over zone U_i. After computing U_i/C^*_i as the first step, the second step arrives at Z_i by arbitrarily selecting and removing states from $U_i - T_i$ until a set that is minimal with respect to connectedness is obtained. This two step process generalizes Ukkonen's algorithm for approximate keyword matching in that it maps into exactly his algorithm when the network expression is a single keyword. Note that his algorithm, like our generalization, does not compute "perfect" zones.

Before proceeding with a detailed development of the algorithm, we elaborate on the central difficulty faced in realizing the computation outlined in the preceding paragraph. The problem is in the first step that must simultaneously discover which vertices are in U_i and *correctly* evaluate C^*_i for each such vertex, for one cannot know one entity without the other. But the correct computation of the value of $C^*_i(u)$ for some $u \in U_i$ requires that all predecessors of *u* in U_i be discovered and have their C^*_i values computed before one proceeds to compute $C^*_i(u)$. This requires that the states of U_i *be discovered and evaluated in topological order*. Thus if (1) *s* and *t* are both immediate predecessors of *u* (i.e., $s \rightarrow u$ and $t \rightarrow u$), and (2) *s* is discovered to be in U_i and have value $C^*_i(s) \leq T$, then one cannot proceed to place *u* in U_i and evaluate it until it is known whether or not *t* is in U_i. This difficulty does not arise in the case that *R* is a simple keyword.

The threshold-sensitive algorithm takes greater advantage of the structure of the automata *F* for motif *M* than the simple algorithm of Section 2. Specifically, it requires that every state *s* have a type, *s.type*, and links *s.succ*, *s.other*, and *s.mate* to operationally important states as defined inductively in Figure 5. The algorithm will compute information at a state on the basis of its type, utilizing the links to efficiently access other states in the automata. The start state, *s*, of a subautomaton for $R|S$ is of type *SPLIT* and has links to its two successors (i.e., *s.succ* and *s.other*) and the final state (i.e., *s.mate*) of the subautomaton. Within $F_{R|S}$, the final states, *u* and *l*, of the subautomata for *R* and *S* are of type *UPPER* and *LOWER*, respectively. They have links to each other (i.e. *u.other* and *l.other*), their single successor (i.e., *u.succ* and *l.succ*), and the start state (i.e., *u.mate* and *l.mate*) of the subautomaton for which their partner is the final state. The final state of *F* is of type *FINAL* and requires no link information. All other states, *c*, are of type *CAT* and require only a link to their single successor in *F* (i.e., *c.succ*). A second requirement is that the states in F be numbered according to the unique topological ordering dictated by the constraint that the vertices of F_S precede those of F_R in an automaton for $F_{R|S}$. For example, if s is of type *SPLIT*, then $s.succ < s.other$ in this ordering. Observe that all these definitions make specific use of the fact that the graph of F is series/parallel and acyclic.

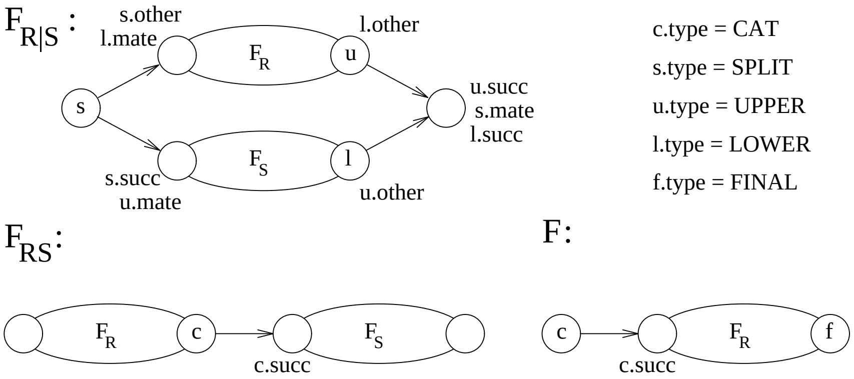

Figure 5. Type and Link Definitions for Series/Parallel *F*.

The goal then is an algorithm that given a zone $M.Z/M.C^*$ for a machine M and a character a will produce a zone for the state of the machine after scanning a. As alluded to earlier the difficulty is discovering the new zone in topological order. The key to doing so is to maintain "dominator" information for the states in $M.Z$. Formally, a state t is *dominated* by s if and only if every path from θ to t passes through s. The information needed for each state s is the largest state (with respect to the topological order) in $M.Z$ that is dominated by s. Formally, for any set $Z \subseteq M.V$, let $Dom_Z(s) = \max \{ t : t \in Z$ and s dominates $t \}$. This dominator information will be used to maintain the topological ordering property of a list of states exactly as follows. Suppose we have the states of subset Z in a topologically ordered list and we have a state $s \in Z$ of type *SPLIT* for which $x = s.succ$ is in Z but $y = s.other$ is not in Z. The topological ordering of the list is preserved if and only if y is inserted immediately after $DOM_Z(x)$ (which by definition is in the list). The same is true when we have a state $s \in Z$ of type *LOWER* for which $x = s.mate$ is in Z but $y = s.succ$ is not in Z. In all other cases, the issue of where to insert the successors of a state $s \in Z$ in the topologically ordered list are easy and do not require the dominator information. However, for the two cases above, this information is essential.

The procedure $ZAdvance(M, a)$ in Figure 6 gives a detailed specification of our threshold-sensitive algorithm for updating the current state of M to the one obtained by scanning the symbol a. Having now outlined the two step process for advancing from one zone to another, and given the key idea of using dominator information to maintain a topological ordering in a sorted linked list, we proceed directly to a proof of the algorithm's correctness.

```
procedure $ZAdvance(M, a)$
 { # Pre-Condition: $Z \equiv Z_{i-1}$ in topological order, $C[s] \equiv C^*_{i-1}(s)$, $In[s] \equiv s \in Z_{i-1}$, $Dom[s] \equiv Dom_{Z_{i-1}}(s)$ #

   $U \leftarrow \varnothing$
   while $Z \neq \varnothing$ do
     { $s \leftarrow$ pop $Z$
       $D[s] \leftarrow \min \{ \min_{t \to s} \{C[t] + \delta(a, \lambda_s)\}, \min_{t \to s} \{D[t] + \delta(\varepsilon, \lambda_s)\}, C[s] + \delta(a, \varepsilon) \}$
       if $In[s]$ or $D[s] \leq T$ then
         { if $s.type = SPLIT$ then
               if not $In[s.other]$ then
                   if $In[s.succ]$ then
                       insert $s.other$ after $Dom[s.succ]$
                   else
                       push $s.other$ onto $Z$
           if $s.type \in \{SPLIT, CAT, UPPER\}$ then
               if top$(Z) \neq s.succ$ then
                   push $s.succ$ onto $Z$
           else if $s.type = LOWER$ then
               if not $In[s.succ]$ then
                   if $In[s.mate]$ then
                       insert $s.succ$ after $Dom[s.mate]$
                   else
                       insert $s.succ$ after $s.mate$
         }
       $In[s] \leftarrow$ true
       push $s$ onto $U$
     }

   # Invariant: $Z \equiv \varnothing$, $U \equiv U_i$ in reverse topological order, $D[s] \equiv C^*_i(s)$, $In[s] \equiv s \in U_i$ #

   while $U \neq \varnothing$ do
     { $s \leftarrow$ pop $U$
       $C[s] \leftarrow D[s]$
       $In[s] \leftarrow C[s] \leq T$ or $s.type \in \{LOWER, UPPER\}$ and $In[s.succ]$ and not $In[s.other]$
                          or $s.type = CAT$ and $In[s.succ]$
                          or $s.type = SPLIT$ and ($In[s.succ]$ or $In[s.other]$)
       if $In[s]$ then
         { push $s$ onto $Z$
           if $s.type = SPLIT$ then
               if $In[s.mate]$ then          $Dom[s] \leftarrow Dom[s.mate]$
               else if $In[s.other]$ then $Dom[s] \leftarrow Dom[s.other]$
               else if $In[s.succ]$ then  $Dom[s] \leftarrow Dom[s.succ]$
               else                      $Dom[s] \leftarrow s$
           else if $s.type = CAT$ then
               if $In[s.succ]$ then $Dom[s] \leftarrow Dom[s.succ]$
               else                $Dom[s] \leftarrow s$
           else
               $Dom[s] \leftarrow s$
         }
     }

   # Post-Condition: $Z \equiv Z_i$ in topological order, $C[s] \equiv C^*_i(s)$, $In[s] \equiv s \in Z_i$, $Dom[s] \equiv Dom_{Z_i}(s)$ #

 }
```

Figure 6. A Threshold-Sensitive Algorithm for Scanning Motif *M* Over Symbol *a*.

Theorem: If $M.Z/M.C^* \equiv Z_{i-1}/C^*_{i-1}$ when $ZAdvance(M, a)$ is called, then $M.Z/M.C^* \equiv Z_i/C^*_i$ after it returns.

Proof: The simple algorithm of Section 2 for *Advance* only needed an $O(p)$ array $M.C$ to model the current row or "state" of motif *M*. The threshold-sensitive algorithm must additionally maintain (1) a linked list $M.Z$ of the states in the current zone in topological order, (2) a state-indexed array of boolean values $M.In$, indicating which states are in $M.Z$, and (3) a state-indexed array $M.Dom$ giving dominator information as described above. To alleviate the notational burden, we will omit the *M.* prefix for structures such as $M.Z$ for the remainder of this proof.

At the time *ZAdvance* is called, the precondition assumed is that the data structures *Z*, *C*, *In*, and *Dom* are correctly established for the minimally connected zone Z_{i-1}/C^*_{i-1} obtained after scanning $i-1$ symbols. The condition is formally stated in Figure 6 but note in particular that for all $s \in Z$, $Dom[s] = Dom_Z(s)$, i.e. the *Dom* array gives the dominator information for all states in the zone *Z*.

The first step of the algorithm (lines 1-22) discovers the states in U_i in topological order and simultaneously computes $D[s] = C^*_i(s)$ for every $s \in U_i$. Thinking of *Z* as a stack, this step pops elements from *Z* and pushes them onto an initially empty stack *U* (also realized as a simple linked list). Thus in outline form, this step "pours" states from the top of stack *Z* onto the top of stack *U* (lines 1, 2, 3, and 22) so that at the end of this first step the list *U* contains the elements of U_i in reverse topological order. As a state *s* transits from *Z* to *U* (lines 4-21): (a) the value $D[s]$ is computed (line 4), (b) *s*'s immediate successors in *F* are inserted into their appropriate place in the topologically ordered *Z* list/stack if they are discovered to be in $U_i - Z_{i-1}$ (lines 5-20), and (c) $In[s]$ becomes true (line 21). Note that the states discovered to be in $U_i - Z_{i-1}$ are placed into *Z* so that they will be poured into *U* in topological order. This insures that $D[s]$ is correctly evaluated as all *s*'s predecessors in U_i have already been poured and thus had their *D*-values computed. The setting of $In[s]$ only upon pouring also insures that one can distinguish between states on the list *Z* that are in Z_{i-1} and those in $U_i - Z_{i-1}$ according to whether their *In*-indicator is true or false, respectively.

To confirm the invariant claimed in Figure 6 at the completion of the first step, it only remains to verify that all the vertices in $U_i - Z_{i-1}$ are discovered and inserted into *Z* at their correct positions. When a state *s* is poured from *Z* to *U*, its immediate successors are in U_i if an only if *s* is in Z_{i-1} or $C_i(s) \le T$. But this is exactly equivalent to $In[s]$ being true or $D[s] \le T$ after executing lines 3-4 of the "pouring loop". Thus the predicate in line 5 correctly determines if *s*'s successors belong to U_i. As stated before the tricky part is to insert these successor states into the right place in *Z*. There are four cases depending on the type of *s*. We consider the most complex case which is when $s.type = SPLIT$ and leave the remaining cases to the reader. Let $x = s.succ$ and $y = s.other$ be the two successors of *s*. Note that because *s* is its only predecessor, *x* is already in *Z* if and only if *x* is in Z_{i-1}. The same is true for *y*. Thus *x*(*y*) is already in *Z* if and only if $In[x]$($In[y]$) is true. If *x* is not in *Z* then, it should be added immediately after *s* since it comes immediately after it in the topological ordering of *V*. But *s* was just popped from *Z*, so it is correct to push *x* onto the front of *Z* if it is not already there (lines 12-14). If *y* is not in

Z then where it goes depends on *x*: if *x* is also not in *Z* then it should be placed immediately after *x* because by connectivity none of the states dominated by *x* are on the list; otherwise it should be inserted immediately after $Dom_{Z_{i-1}}(x) = Dom[x]$ (as $x \in Z_{i-1}$). Note that in the first case the algorithm is correct to push *y* ($s.other$) onto the top of *Z* in line 11 as immediately thereafter it pushes *x* in front of it in line 14. Thus, the logic of lines 6-20 correctly inserts *s*'s successors in the event that its type is *SPLIT*.

After the first step of the algorithm has completed the invariant of Figure 6 is true. Namely, *Z* is empty, the list *U* contains the states in U_i in reverse topological order, $D[s]$ contains the value $C_i^*(s)$ for every $s \in U_i$, and the boolean indicator $In[s]$ is true iff $s \in U_i$. In the second step (lines 23-38), the algorithm pours states from *U* onto the now empty *Z* (lines 23, 24, and 28) with some states being dropped (i.e., not pushed onto *Z*) if they are not needed to maintain the connectedness property of Z_i. At the same time, *D*-values are transferred to *C* (line 25), *In* is established for Z_i (line 26), and the new dominator information for Z_i is computed in *Dom* (lines 29-38). When a state *s* is poured out of *U* it flows into *Z* only if (1) $D[s] \leq T$, (2) *s* is the sole predecessor of a state already placed in *Z*, or (3) *s* has a successor *t* already in *Z*, no other predecessor of *t* has been placed in *Z*, and *s* was the last predecessor of *t* in *U*. This strategy guarantees that the collection of states poured into *Z* is minimally connected. The predicate in line 26 realizes the criterion and is assigned to the *In*-indicator for the state. Dominators are easily computed as a function of the dominators of immediate successors also in Z_i as given in Figure 6. For example, consider *s* of type *CAT*. If $Dom_{Z_i}(s) \geq s.succ$ (in the topological order of states), then because $s.succ$ is the sole successor of *s* it follows that $Dom_{Z_i}(s) = Dom_{Z_i}(s.succ)$, and by connectivity it must be that $s.succ \in Z_i$. Thus $In[s.succ]$ is true and $Dom[s] = Dom[s.succ]$ (line 35). The other case is $Dom_{Z_i}(s) = s$ in which case $s.succ$ cannot be in Z_i (line 36). Verification of the logic for the other state types is left as a simple exercise to the reader. After executing the loop of lines 23-38, it thus follows that *Z*, *C*, *In*, and *Dom* are correctly established for the zone Z_i / C_i^* as asserted in the post-condition of Figure 6. ■

Next we show that our result is easily extended to regular expressions. We assume the reader is well acquainted with the result of Myers and Miller (1989) for approximately matching regular expressions. Note first that ignoring back-edges, the automaton for a regular expression is a series/parallel graph. So we can use the result of this paper as a subroutine for the two passes of the Myers/Miller algorithm. In brief, to advance a regular expression automaton over symbol *a*, start by calling $Zadvance(M',a)$ where M' is the automaton without back-edges. Then propagate values across back-edges $s \rightarrow t$ for all $s \in M.Z$ observing that *t* must also be in $M.Z$ as it dominates *s*. Finally, call $Zadvance(M',\varepsilon)$ to complete the second pass.

The final algorithmic consideration is to demonstrate that the procedure *ZAdvance* is embeddable in the context of Section 2 which required the concepts of injection, the status of the resulting zone, or how to get started. Figure 7 presents algorithms for the primitives *Advance* and *Start* of Section 2 that use *ZAdvance* as a subroutine. To achieve an injection of 0 into the *C*-value of the start state, *Advance* first adds θ to the current zone *Z* if necessary in lines 2 through 4, and then sets $C[\theta]$ to $-\delta(a, \varepsilon)$ in line 5. This guarantees that the subsequent call to

ZAdvance will correctly set $D[\theta]$ to 0 in line 4 on the first iteration of the loop of the first step. Upon return from the call to *ZAdvance* in line 6, *Advance* returns *yes*, *no*, or *never* depending on the zone. If Z is empty then every C-value must be greater than T and so *never* is the correct return value (lines 8). Otherwise, if $C^*(\phi) > T$ then *no* is returned (line 10), and *yes* otherwise (line 12). *Start* determines the initial zone Z_0 / C_0^* by simply establishing the empty zone (lines 1-3) and then calling *Advance* (line 4) with an arbitrary character (e.g., ε) and injection on.

```
function $Advance(M, a, inject)$: $(yes, no, never)$
1.   {  if $inject$ then
2.         {  if $M.Z = \emptyset$ then
3.               {  $M.Z \leftarrow \{ \theta \}$
4.                  $M.In[\theta] \leftarrow$ true
                 }
5.            $M.C[\theta] \leftarrow -\delta(a, \varepsilon)$
         }
6.      $ZAdvance(M, a)$
7.      if $M.Z = \emptyset$ then
8.            return $never$
9.      else if not $M.In[M.\phi]$ or $M.C[M.\phi] > M.T$ then
10.           return $no$
11.     else
12.           return $yes$
     }

procedure $Start(M)$
1.   {  for $s \in M.V$ do
2.            $M.C[s] \leftarrow M.D[s] \leftarrow M.T + 1$
3.      $M.Z \leftarrow \emptyset$
4.      $Advance(M, \varepsilon,$ true)
     }
```

Figure 7. Threshold-Sensitive Versions of *Start* and *Advance*.

Finally, we turn to the consideration of the efficiency of our algorithm. It is easy to see that the two major loops of *ZAdvance* are repeated exactly $|U_i|$ times when called with zone Z_{i-1}/C^*_{i-1}. Because the body of each loop is straight-line code and $|U_i|$ is $O(|Z_{i-1}|+|Z_i|)$ it follows that a call to *ZAdvance* takes $O(|Z_{i-1}|+|Z_i|)$ time. Thus the time to find all matches within threshold T with the procedure *Scanall* of Figure 4 takes $O(tn)$ expected time where t is the average size of the zones, $|Z_i|$, over the course of the scan. As for the threshold-sensitive algorithms presented by Ukkonen (1985) and Fickett (1984), we do not in this paper focus on the interesting and difficult question of characterizing t in terms of T, R and δ.[1] Certainly the two t's are positively correlated: the more stringent the choice of T, the smaller the computed zone, and the faster our threshold-sensitive algorithm performs compared to the simple dynamic programming algorithm of Figure 4. Moreover, we can make some preliminary statements about the correlation between t and T. Because our algorithm reduces to exactly Ukkonen's algorithm when the network expression is a single keyword, it must take $O(Tn)$ expected time when R is a keyword and δ is the unit cost scoring scheme. It then follows as a corollary that under unit cost

[1] Ukkonen's algorithm was not proven to be *O(kn)* by Ukkonen in his 1985 paper, but seven years later in a separate paper and analysis by Chang and Lampe (1992).

scores, the algorithm takes no more than $O(min\{|\Sigma|^T,\ Tw\}\ n)$ expected-time where Σ is the input alphabet and *w* is the width of the network expression, i.e., the maximum cardinality of a cut-set of *F*. The problems of a tighter characterization, or a characterization for the case where δ is arbitrary, are left open.

To give the reader an idea of the speed of the threshold algorithm in practice we present Table 1 below which compares its speed for various threshold values against an implementation of the basic algorithm presented in Figure 4 of Section 2. The implementations were run over one million characters of a typical protein database. The scoring scheme used was the simple unit cost model so that the threshold *T* is coincident with the number of differences in a match. The times are in seconds and were obtained on a Dec Alpha 4/233. Following each time in parenthesis is the average size of the zone during the scan. In the case of the entries for the basic algorithm these numbers are the number of states in the automaton for the pattern. The essential observation is that the more complex logic required for the threshold-sensitive algorithm makes it require roughly three times more time per state in the zone over the basic algorithm. Thus it is more efficient than the basic algorithm only when the threshold is such that, on average, the zone involves less than ⅓ of the states in the automaton. Within the ANREP system mentioned at the start of the paper (Myers and Mehldau 1993), we have implemented both the basic and threshold sensitive algorithms. Given a number of motifs in a net pattern, we run the threshold-sensitive algorithm over a random string of 1000 characters to determine if the average zone occupies more than ⅓ of the automaton. If so then we use the basic algorithm for the motif in question during the proper scan of the database; otherwise we use the threshold-sensitive algorithm. That is, ANREP adaptively chooses the better algorithm for the particular motif. It is clear from Table 1 that there are cases where the threshold algorithm provides superior performance.

Pattern	Threshold Sensitive Algorithm					Basic Algorithm
	T=0	T=1	T=2	T=3	T=4	
GCTCCGICTN	1.40(1.06)	1.68(2.19)	2.38 (3.37)	2.90 (4.53)	3.36 (5.58)	2.54(10)
(GCTCCGICTN |VEKGKKIFVQ |EETLMEYLEN)	3.41(3.20)	4.97(6.56)	6.85(10.08)	8.66(13.68)	10.68(17.02)	7.44(36)
GCTCC(GICTN |KIFVQ |EYLEN)	1.40(1.08)	1.72(2.22)	2.57 (3.43)	3.56 (4.77)	5.88 (7.84)	5.34(26)
[ILM][DS][FL]F[ACS]G. [GM][AG][FIL]..[AGS]....G	1.50(1.21)	1.99(2.54)	2.78 (3.87)	3.54 (5.40)	4.66 (7.45)	4.24(20)

Table 1: Empirical Results for a protein scan of length 1,000,000.

4. An Optimized Backtracking Net Matching Algorithm. Faced with the problem of matching a net pattern, one has at least two choices. First, when spacers are restricted to be positive it can be shown that the class of net patterns is a regular language and thus one could attack the problem as one of finding an approximate match to a single large regular expression where each motif matches within its threshold. But such an approach for large nets quickly becomes unwieldy and is potentially quite costly because the amount of time spent on a spacer is proportional to the value of its delimiting integers. Given the desire for negative spacers, which cannot be modeled by such an approach, we take instead a two-tiered approach of viewing the problem of finding net matches given subroutines, such as those of Figure 4, for matching motifs. One should note that the net matching problem reduces to "proximity search" when motifs are exact matches to keywords (Manber and Baeza-Yates 1991).

The performance of the threshold-sensitive versions of the routines *Start* and *Advance* can be estimated via Monte Carlo simulation over a random text whose stochastic properties model the text to be scanned up to some appropriate level, say a first- or second-order Markov model. By starting a motif, *M*, and then advancing it always with $inject$ set to true, one can estimate the average time, *t*, for each advance of the recognizer. That is, one can with some precision assert that the expected time to scan *A* with *M* will be *tn*. Via simulation one can also estimate the average amount of time, *x*, it will take for a recognizer to return *never* when advanced without injection. The recognizer is assumed to be started in each state according to the probability with which that state is occupied during an injecting scan. With these two parameters one may then *estimate* that a call to $Scan(M, \{j_1 \cdots j_k\})$ will consume $(j_k - j_1)t + x$ time.[2] During the simulation one can also get a rough estimate of *f*, the frequency with which *M* is found within a random text. If *n* is very large, then simulating each motif in a net over a random text of length, say, 1% or less of *n* to estimate the parameters above is not an unduly high overhead to pay for the ability to optimize the search for the net as shown below.

4.1. Optimizing the Backtrack Order. To illustrate the optimized backtracking idea, consider a "linear" net $M_0\, S_1 M_1 S_2 M_2 \cdots S_p M_p$ where the M_k are motifs and the S_k are spacers $[l_k, r_k]$. Let $\Delta_k = r_k - l_k$ be the variance of each spacer. Suppose that searching for a match to motif M_k on a substring of *A* of length *m* takes $t_k m + x_k$ where the parameters t_k and x_k have been determined as above. Further suppose that one finds a match to M_k on a random string with frequency f_k. Given these parameters, the time that a particular backtracking order will take can be accurately estimated. For example, consider the strategy of looking for M_0, whenever a match to it is found, search S_1 symbols downstream for M_1, if an instance of it is found, search S_2 symbols downstream for M_2, and so on, backtracking if one fails to find a match at any point. Assuming motif searches are independent of each other, the time to perform a search of *A* for this particular order is expected to be:

$$nt_0 \;+\; nf_0 \sum_{k=1}^{p} \Big(\prod_{c=1}^{k-1} \Delta_c f_c\Big)(\Delta_k t_k + x_k) \quad {}^{3}$$

[2] This is indeed just an estimate as the state occupancy probabilities of the automaton depend on the number of characters scanned with $inject$ set to true. So more a more accurate model of the average time taken would be $t(n) \cdot n + x(n)$ where $n = j_k - j_1$ and $t(n)$ and $x(n)$ are functions of *n*. We find in practice that such accuracy is not necessary especially since the cost of estimating $t(n)$ and $x(n)$ would outweigh any benefit gained.

[3] It takes nt_0 time to search *A* for instances of M_0 and a match will be found at nf_0 locations. For each of these, $\Delta_1 t_1 + x_1$ time is spent looking for matches to M_1 and of the Δ_1 potential left ends, $\Delta_1 f_1$ will be matches. Thus at $nf_0 \Delta_1 f_1$ locations, one will proceed to search for matches to M_2. Continuing in this fashion gives the formula.

However, this order is particularly bad if M_0 is very frequent and/or expensive to search for. A much better order would be to start by first searching for a motif or subset of motifs that are fairly rare and inexpensive to search for. All possible *consecutive orders* are considered as candidates for the order in which to perform a backtracking search for the linear net. An order is consecutive, if at each stage in the search, a consecutive range, say M_k through M_h of motifs has been matched and the next one searched for is M_{k-1} or M_{h+1}. For example, if $p = 4$ then the possible orders are 1-2-3-4, 2-1-3-4, 2-3-1-4, 2-3-4-1, 3-2-4-1, 3-2-1-4, 3-4-2-1, 4-3-2-1. Other orders are prohibited since in general one does not have a range estimate on the size of the substring matched by a motif (e.g. if M_2 were matched then in what range would one search for M_4 if M_3 has not yet been found?). Over this set of orders a simple dynamic programming calculation can determine an *optimal* order. As stated earlier, we assume that motif matches are independent of each other, an approximation satisfactory for our pragmatic purpose. Let $Best(k, h)$ be the minimum time to match all other motifs conditioned on motifs M_k through M_h already being matched. These $O(p^2)$ quantities can be computed using the recurrence:

$$Best(k+1,h-1) = \min \{ \Delta_{k+1} f_k Best(k,h-1) + t_k \Delta_{k+1} + x_k \, , \Delta_h f_h Best(k+1,h) + t_h \Delta_h + x_h \} [4]$$

It then follows that the best time for a consecutive backtracking order that starts with motif M_k, $Opt(k)$ is $nt_k + nf_k Best(k, k)$. Taking the best time over all choices of k gives us a backtracking order that is optimal in expectation. The first motif to be searched for is called the *seed* of the search.

The treatment above can be generalized to finding optimal backtracking orders over arbitrary nets as opposed to just linear nets. The extension is sketched here. Let F be the ε-NFA for the net in question; its states are labeled with spacers, motifs, and ε. The seed of a backtracking search now consists of any cut-set for F all of whose states are labeled with motifs. That is, one searches in parallel for a match to one of the motifs in this *seed cut-set*, and whenever one is found, one trys extending the match in an optimal consecutive order. However, the extension of the match is no longer along a linear network. A match involving motif M_s, where s is the seed state, can correspond to any path from θ to ϕ through s, and the subgraph of these vertices and edges can be a network. So extending a match in both directions from s can follow any of the paths in this network and there can be an exponential number of such paths. However, the probability of extension drops of exponentially as well, so that the expected time to explore/eliminate all possible extensions is generally proportional to the size of the net. The following formula computes $Best(u, v)$, the minimum time to complete searching for an extension to a match to M_u through M_v on a path through s under the assumption that the matching of motifs is an independent event (which it is not):

[4] If M_{k+1} to M_{h-1} are matched then the next step must be to either match M_k or M_h. The two terms in the minimum are the times to do the respective extensions.

$$Best(u, v) = \min \{ \sum_{w \in Pred(u)} (\Delta_{w \to u}\ f_w\ Best(w, v) + t_w \Delta_{w \to u} + x_w)\ ,\ \sum_{w \in Succ(v)} (\Delta_{v \to w}\ f_w\ Best(u, w) + t_w \Delta_{v \to w} + x_w) \}$$

In the formula above, $Pred(u)$ is the set of states immediately preceding u that are labeled with motifs, i.e., states w such that there is a path from w to u whose interior vertices (if any) are labeled with either ε or a space. $Succ(u)$ is analogously defined, and $\Delta_{w \to u}$ denotes the aggregate variance of the spacers on the interior of the path from w to u. Assuming that now p is the length of net N, one can compute $Best(u, v)$ for the $O(p^2)$ pairs of motif-labeled states that are on some source-to-sink path in N's automaton, and then estimate $Opt(s)$, the best time for a backtrack procedure that starts at s, as $nt_s + nf_s Best(s, s)$. Given these estimates, a minimal seed cut-set can be found in $O(p)$ time[5] over the series-parallel automaton of net N where the cost of a cut-set is the sum of $Opt(s)$ for s in the set. Thus in $O(p^2)$ time one can compute an optimal backtracking order that works well in practice as the interdependence of motif matches is generally a small effect in expectation.

4.2. Finding and Reporting Matches. Having now determined an order in which to search for the components of a net, consider the problem of finding and reporting a match to the linear net, $M_{-p} S_{-p} M_{-(p-1)} \cdots M_{-1} S_{-1} M_0\ S_1 M_1 \cdots M_{q-1} S_q M_q$, given an optimal backtracking order over $[-p, q]$ that begins with seed motif M_0. It will be left as a straightforward exercise for the interested reader to extend our treatment to arbitrary nets. From the preceding sections, assume a routine that determines $Scan(M, J)$ for a subset J of $[0, n]$. Further observe that by building an automaton for the reverse, R^r, of network expression R[6] and scanning A in reverse, one obtains an analogous routine that computes $Scan^r(M, J)$ = $\{\ i : \exists j \in J\ s.t.\ A_{i..j} \in L_\delta(R, T)\ \}$. That is, by scanning right-to-left with the reverse of R, one can find left ends as opposed to right ends of matches. This is essential since match extension must proceed right-to-left from M_0 to M_{-1} to M_{-2} and so on. Also recall from the introduction, that the notation $A \sim N$ denotes that string A matches net N so that, for example, one could have defined $Scan(M, J) = \{\ i : \exists j \in J\ s.t.\ A_{j..i} \sim M\ \}$. Finally assume the functions $Space_k(J)$ = $\cup_{j \in J}\ [j + l_k, j + r_k]$ and $Space^r_k(J) = \cup_{j \in J}\ [j - r_k, j - l_k]$ for spacer $S_k = [l_k, r_k]$. For sets J that are in sorted order, these simple functions are easily computed "on-the-fly" in time linear in the size of J.

With these primitives, finding a match to the linear net seems quite straightforward at first glance. Search for the right ends of matches to M_0 in a forward scan. Such endpoints tend to cluster in small intervals $R_0 = [a, b]$ when there is a tight match to the motif because extending the match a few characters in either direction yields approximate matches that are still within

[5] The cost of the best cut set for expression R, $Opt(R)$ is easily computed using the following recurrence. $Opt(a)$ is ∞ if a is ε or a spacer, and $Opt(s)$ of the state s modeling a otherwise. Inductively, $Opt(RS) = \min\{Opt(R), Opt(S)\}$ and $Opt(R|S) = Opt(R) + Opt(S)$. A cut-set delivering the optimal value is easily recovered.

[6] The reverse of a network expression R is the expression R^r that matches the reverse of every word matched by R. It is easily obtained by inductively applying the rules $(RS)^r = S^r R^r$ and $(R|S)^r = R^r|S^r$ top-down. For example, $(a(b|cd)e)^r = e(b|dc)a$.

threshold, albeit of greater score. For this *set* of right ends, determine the set of potential left ends, L_0 via a call to $Scan^r(M_0, R_0)$. Next in back track order from this seed set, begin extending the match in both directions until one either fails or completes the extension. The invariant for this process is that if at some point M_{-k} through M_h have been matched, then the set $L_k = \{ i : \exists j \in [a, b] \ s.t \ A_{i..j} \sim M_{-k} \cdots M_0 \}$ and the set $R_h = \{ i : \exists j \in [a, b] \ s.t \ A_{j..i} \sim S_1 \cdots M_h \}$. That is, L_k contains the left ends of matches to the subnet $M_{-k} \cdots M_0$ whose right ends are in $[a, b]$, and R_h contains the right ends of matches to the subnet $S_1 M_2 \cdots M_h$ whose left ends are in $[a, b]$. To extend the match to L_{k+1} it suffices to compute $Scan^r(M_{-(k+1)}, Space^r_{-(k+1)}(L_k))$ and, similarly, $R_{h+1} = Scan(M_{h+1}, Space_{h+1}(R_h))$. If at any point a set of ends is found to be empty, the process quits prematurely and the main scan for seed matches continues. This outward extension process is "Sweep 1" of the algorithm of Figure 8. Note that the extension of several endpoints is pursued in parallel whereas the formula driving the choice of back track order assumed the back tracking proceeded endpoint by endpoint. For approximate matching, where endpoints tend to cluster in the vicinity of matches, our approach is essential for efficiency in regions preconditioned to match the net.

The surprise is that the extension process can succeed even though there is no match to the net. If the sets L_p and R_q are non-empty, what this implies is that there are matches to the subnet $M_{-p} \cdots M_0$ and the subnet $S_1 \cdots M_q$ whose right and left ends are in $[a, b]$, respectively. However, by an extremely unlikely coincidence, it may arise that there is not a pair of these matches, one from each "half", that share the same end in $[a, b]$. This problem is a result of the choice to extend sets of match endpoints in parallel and would not have arisen if efficiency considerations hadn't precluded the application of the extension process to each index in $Scan([0, n], M_0)$ separately. This difficulty is rectified by proceeding with an additional inward sweep, "Sweep 2", that determines the set of right ends of matches to $M_{-p} \cdots M_0$ that are in $[a, b]$, and similarly, the set of left ends of matches to $S_1 \cdots M_q$ that are in $[a, b]$. Clearly, if these two sets intersect then there is a match to the net with a right-end match to M_0 in $[a, b]$. With *Scan* and *Space* the sweep finds, for progressively smaller values of *k*, the set of right ends of matches to $M_{-p} \cdots S_{-(k+1)}$ whose left end is in L_p, and then subtracts this set from L_k computed in the first sweep. Since L_p is the set of left ends of matches to $M_{-p} \cdots M_0$ whose right-end is in $[a, b]$, it follows by induction that at the end of the sweep, $L_k = \{ i : \exists j \in [a, b] \text{ and } l \leq i \ s.t \ A_{l..i} \sim M_{-k} \cdots S_{-(k+1)} \text{ and } A_{i..j} \sim M_{-k} \cdots M_0 \}$. Proceeding from the other end with $Scan^r$ and $Space^r$, the sweep refines the *R*-sets so that $R_h = \{ i : \exists j \in [a, b] \text{ and } r \geq i \ s.t \ A_{j..i} \sim S_1 \cdots M_h \text{ and } A_{i..r} \sim S_{h+1} \cdots M_q \}$ at the end of the sweep. The sweep concludes by determining if the set of desired left ends, R_0, intersects the desired set of right ends, $Scan(M_0, L_0)$.

```
for $[a, b]$, a maximal interval s.t. $[a, b] \subseteq Scan(M_0, [0,n])$, in left-to-right order do
  { $R_0 \leftarrow [a, b]$
    $X \leftarrow L_0 \leftarrow Scan^r(M_0, R_0)$                                   # Sweep 1 #
    for $k$ in backtrack order of $[-p,-1] \cup [1,q]$ do
      { if $X = \emptyset$ then break
        if $k < 0$ then
              $X \leftarrow L_{-k} \leftarrow Scan^r(M_k, Space^r_k(L_{-k-1}))$
        else
              $X \leftarrow R_k \leftarrow Scan(M_k, Space_k(R_{k-1}))$
      }
    if $X = \emptyset$ then continue

    for $k \leftarrow p-1$ downto $0$ do                                   # Sweep 2 #
        $L_k \leftarrow L_k \cap Space_{-(k+1)}(Scan(M_{-(k+1)}, L_{k+1}))$
    for $k \leftarrow q-1$ downto $0$ do
        $R_k \leftarrow R_k \cap Space^r_{k+1}(Scan^r(M_{k+1}, R_{k+1}))$
    if $Scan(M_0, L_0) \cap R_0 = \emptyset$ then continue

    $L_0 \leftarrow L_0 \cap Scan^r(M_0, R_0)$                                   # Sweep 3 #
    $R_0 \leftarrow R_0 \cap Scan(M_0, L_0)$
    for $k \leftarrow 1$ to $p$ do
        $L_k \leftarrow L_k \cap Scan^r(M_{-k}, Space^r_{-k}(L_{k-1}))$
    for $k \leftarrow 1$ to $q$ do
        $R_k \leftarrow R_k \cap Scan(M_k, Space_k(R_{k-1}))$

    for $k \leftarrow p$ downto $1$ do                                   # Sweep 4 #
        The range of $M_{-k}$ is min$\{ j : j \in L_k \}$ to max$\{ j : j \in Scan(M_{-k}, L_k) \cap Space^r_{-k}(L_{k-1}) \}$
    The range of $M_0$ is min$\{ j : j \in L_0 \}$ to max$\{ j : j \in R_0 \}$
    for $k \leftarrow 1$ to $q$ do
        The range of $M_k$ is min$\{ j : j \in Scan^r(M_k, R_k) \cap Space_k(R_{k-1}) \}$ to max$\{ j : j \in L_k \}$
  }
```

Figure 8. Finding and Reporting a Match to a Linear Net.

Having now found a match, the next question is what to report. In the case of approximate matching this is a non-trivial question. To illustrate, suppose that there is a match to motif *M* well-within its threshold. Then the substrings of *A* obtained by extending the stringent match a few characters at either end are also matches within threshold. Does one wish to see the longest match, the lowest scoring, or some indication of the range of possible matches? This reporting problem is further compounded in the case of a match to a net where each motif match is well within threshold and hence where there are a number of choices for each motif. Two algorithms are presented here: (1) a *range* algorithm that determines the range of left and right ends possible for each motif in some match to the entire net, and (2) an *optimum* algorithm that selects a net match for which the sum of the scores of its motif matches is minimal.

The range algorithm is presented as Sweeps 3 and 4 in the algorithm of Figure 8. The third sweep proceeds outward further refining the sets L_k and R_h computed in Sweeps 1 and 2. At the start of Sweep 3, R_0 contains the left ends of matches to $S_1 \cdots M_q$ that are in $[a, b]$. By extending this match set to the left with $Scan^r$ and $Space^r$, and intersecting the results with the sets L_k from the previous two sweeps, one arrives at the end of the sweep with $L_k = \{ i : \exists j \in [a, b], l \leq i,$ and $r \geq j$ $s.t$ $A_{l..i} \sim M_{-k} \cdots S_{-(k+1)}$, $A_{i..j} \sim M_{-k} \cdots M_0$, and $A_{j..r} \sim S_1 \cdots M_q \}$. Similarly, taking the match endpoint set L_0 and extending it to the right produces the sets $R_h = \{ i : \exists j \in [a, b], l \leq i,$ and $r \geq j$ $s.t$ $A_{l..j} \sim M_{-p} \cdots M_0$,

$A_{j..i} \sim S_1 \cdots M_h$ and $A_{i..r} \sim S_{h+1} \cdots M_q$ *}*. Thus at the end of Sweep 3, the set L_k gives the set of all possible left-end matches to motif M_k that are part of overall matches to the net *N* whose right-end match to M_0 is in $[a, b]$. Similarly, R_h gives the set of all possible right ends for motifs M_h for *h* from 0 to *q*. Thus after completing Sweep 3 the only missing information is the possible right ends of motifs M_{-k} for $k \in [1, p]$ and the possible left ends of motifs M_h for $h \in [1, q]$. But the former are readily computed "on-the-fly" by determining $Scan(M_{-k}, L_k) \cap Space^r_{-k}(L_{k-1})$ and the later by $Scan^r(M_h, R_h) \cap Space_k(R_{h-1})$. Thus, by computing these missing endpoint sets on the fly when needed, Sweep 4 can proceed left-to-right printing the minimum left end and maximum right end for each motif in the net. This treatment should suffice to convince the reader that one can effectively compute useful information about the degrees of freedom in "a" match to net *N*.

The second match-reporting algorithm attempts to select a specific match to the net that is in some sense best, and failing that, at least representative. The simple optimality criterion used is that the sum of the scores of the motif matches forming a match to the net is minimal. Notice that because of spacing constraints this is not equivalent to picking the lowest scoring match for each motif (as this may not form a match to the net). This optimal match algorithm, like the range algorithm, begins after Sweeps 1 and 2 above have determined that a match exists. It further decomposes into symmetric left and right halves, so it suffices to focus on the left half. The essence of the algorithm is a dynamic programming computation that for each L_k, in decreasing order of *k*, determines $Score_{L_k}[j]$, the minimum sum over matches to $M_{-p} \cdots S_{-(k+1)}$ whose left end *j* is in L_k. In order that a match achieving this score can be output, the algorithm simultaneously records the trace-back information, $Left_{L_k}[j]$ and $Right_{L_k}[j]$, the left and right ends of a match to $M_{-(k+1)}$ involved in a match achieving score $Score_{L_k}[j]$.

The basis for the induction of the algorithm is that $Score_{L_p}[j] = 0$ for all $j \in L_p$. $Left_{L_p}$ and $Right_{L_p}$ are superfluous. So the induction step, achieved by the code fragment in Figure 9, is given $Score_{L_{k+1}}[j]$ for all $j \in L_{k+1}$, determine $Score_{L_k}[j]$, $Left_{L_k}[j]$, and $Right_{L_k}[j]$ for $j \in L_k$. The first step of the code fragment computes for each $i \in Scan(M_{-(k+1)}, L_{k+1})$, $B[i] = \min \{ Score_{L_{k+1}}[j] + \delta(A_{j..i}, R_{-(k+1)}) : j \in L_{k+1} \}$ and $I[i]$, an index *j* giving the minimum value for $B[i]$. Recall that R_k is the net expression for motif M_k and that $\delta(A, R)$ is the score of the best alignment between *A* and a sequence in $L(R)$. By the induction hypothesis on $Score_{L_{k+1}}$ it follows that $B[i]$ is the best scoring match to $M_{-p} \cdots M_{-(k+1)}$ whose right end is *i*. This minimum is computed by running the scanner for $M_{-(k+1)}$ once for each $j \in L_{k+1}$, letting that *j* be the only potential left end for the scan. As the scan proceeds, the scanner not only reports whether *i* is the right end of a match to the motif, but also, $Value(i)$ the score of the match given by $C(i, \phi)$. If $Value(i) + Score_{L_{k+1}}[j]$ improves the current minimum recorded at $B[i]$, then it is updated and the *j* for the scan is recorded in $I[i]$. The second step completes the induction step by computing $Score_{L_k}[j] = \min \{ B[i] : i \in Space^r_{-(k+1)}(j) \}$ and for the *i* giving the minimum, recording $Left_{L_k}[j] = O[i]$ and $Right_{L_k}[j] = i$. To compute the minimum efficiently, $Score_{L_k}[j]$ is computed in increasing order of j, which implies that the minimum is needed over a series of intervals $[j - r_{-(k+1)}, j - l_{-(k+1)}]$ whose endpoints increase. The trick is to maintain a heap of the B-values in the current interval, and then incrementally update the heap for the next interval by deleting and adding positions as necessary. Given the heap, each minimum can be extracted in $O(\log \Delta_{-(k+1)})$ time.

```
$low \leftarrow \min \{ j : j \in Scan(M_{-(k+1)}, L_{k+1}) \}$
$hgh \leftarrow \max \{ j : j \in Scan(M_{-(k+1)}, L_{k+1}) \}$
for $i \leftarrow low$ to $hgh$ do
    $B[i] \leftarrow \infty$
for $j \in L_{k+1}$ do
    for $i \in Scan(M_{-(k+1)}, \{j\})$ do
        if $Score_{L_{k+1}}[j] + Value(i) < B[i]$ then
          { $B[i] \leftarrow Score_{L_{k+1}}[j] + Value(i)$
            $I[i] \leftarrow j$
          }
$Heap \leftarrow \emptyset$
$rgt \leftarrow low - 1$
$lft \leftarrow \infty$
for $j \in L_k$ in increasing order do
  { for $i \leftarrow \max \{lft, low\}$ to $j - l_{-(k+1)} - 1$ do
        if $B[i] < \infty$ then
            delete $i$ from $Heap$
    for $i \leftarrow rgt + 1$ to $\min \{j - r_{-(k+1)}, hgh\}$ do
        if $B[i] < \infty$ then
            add $i$ to $Heap$ with priority $B[i]$
    $lft \leftarrow j - l_{-(k+1)}$
    $rgt \leftarrow j - r_{-(k+1)}$
    $i \leftarrow$ extract min from $Heap$
    $Score_{L_k}[j] \leftarrow B[i]$
    $Left_{L_k}[j] \leftarrow I[i]$
    $Right_{L_k}[j] \leftarrow i$
  }
```

Figure 9. Determining an optimal match.

Given that the analogous computation for the R-sets has been performed, the score of the optimum match is easily found by computing the best of $Score_{L_0}[j] + \delta(A_{j..i}, R_0) + Score_{R_0}[i]$ over $j \in L_0$ and $i \in R_0$. The i and j giving the minimum delimit the match to M_0 in an optimum match. The left and right indices of the other matches are obtained by following the trace-back information in the $Left$ and $Right$ arrays in the obvious manner. For example, $Left_{L_2}[Left_{L_1}[Left_{L_0}[j]]]$ and $Right_{L_2}[Left_{L_1}[Left_{L_0}[j]]]$ give the left and right ends of the match to motif M_{-2} in an optimum scoring match (provided j is the index giving the minimum above). If it is desired, one can also deliver for each motif, an alignment between one of its sequences and the substring of A it matches that realizes the score of the optimal match using a linear space algorithm like the one presented in Myers and Miller (1989).

To conclude consider the worst-case time complexity for reporting matches in the case that a match does occur in a given region. In regions that are essentially random with respect to the pattern, the expected amount of time taken is described by the calculation of Section 4.1.

Observe that the algorithm of Figure 8, which includes the range reporting sub-algorithm, makes a number of scans that together span a range of symbols approximately as long as the span of the match to the net. More precisely, one can assert that a sweep of the algorithm scans no more than $\Sigma_k \ (set_k + mot_k + \Delta_k)$ symbols, where mot_k is the length of the longest word matched by M_k, and set_k = $\max\{ j : j \in X \} - \min\{ j : j \in X \}$ where X is $L_{| k |}$ or R_k depending on whether k is positive or negative. Scanning each symbol takes an amount of time depending on the motif involved in the scan, but letting p be the maximum length of a motif network expression in net N, the worst case complexity of the algorithm is certainly proportional to p times the sum above. In expectation, set_i is a small constant and mot_i is on the order of the size of its network expression. Thus more coarsely one may estimate the algorithm to take $O(mp(p + \Delta))$ time for a net with m motifs and average spacer variance Δ. Finally, Figure 9 takes $O(|L_{k+1}| mot_{-(k+1)} \ p \ + \ (set_k + \Delta_{-(k+1)}) \ \log \Delta_{-(k+1)})$ worst-case time. Using the approximations about set_i, etc., a "back-of-the-envelope" estimate for the additional overhead of the optimum match algorithm is $O(m(p^2 + \Delta \log \Delta))$.

LITERATURE

Abarbanel, R.M., Wieneke, P.R., Mansfield, E., Jaffe, D.A., and Brutlag, D.L. (1 984), ‘‘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

Chang, W. I., and J. Lampe. 1992. "Theoretical and Empirical Comparisons of Approximate String Matching Algorithms." *Proc. 3rd Symp. on Combinatorial Pattern Matching* Tucson, AZ (April 1992), 172-181.

Fickett, J.W. 1984. "Fast optimal alignment." *Nucleic Acids Research* **12**, 175-179.

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.

Hopcroft, J.E. and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley (Reading, MA), 13-76.

Levenshtein, V.I. 1966. "Binary codes capable of correcting deletions, insertions, and reversals." *Cybernetics and Control Theory* **10**, 707-710.

Manber, U. and R. Baeza-Yates. 1991. "An algorithm for string matching with a sequence of don’t cares." *Information Processing Letters* **37**, 133-136.

Miller, J., A.D. McLachlan and A. Klug. 1985. "Repetitive zinc-binding domains in the protein transcription factor IIIA from Xenopus oocytes." *EMBO Journal* **4**, 1609-1614.

Myers, E.W. and G. Mehldau. 1993. "A system for pattern matching applications on biosequences." *CABIOS* **9**, 299-314.

Myers, E.W. and W. Miller. 1989. "Approximate matching of regular expressions." *Bull. of Math. Biol.* **51**, 5-37.

Needleman, S.B. and C.D. Wunsch. 1970. "A general method applicable to the search for similarities in the amino-acid sequence of two proteins." *J. Molecular Biology* **48**, 443-453.

Posfai, J., A.S. Bhagwat, G. Posfai, and R.J. Roberts. 1989. "Predictive motifs derived from cytosine methyltransferases." *Nucleic Acids Research* **17**, 2421-2435.

Sankoff, D. and J. B. Kruskal. 1983. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley (Reading, MA), 265-310.

Sellers, P.H. 1980. "The theory and computation of evolutionary distances: pattern recognition." *J. Algorithms* **1**, 359-373.

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

Saurin, W. and P. Marliere, ‘‘Matching relational pattern in nucleic acid sequences’’, *CABIOS* **3**(2) (1987), 115-121.

Ukkonen, E. 1985. "Finding approximate patterns in strings." *J. of Algorithms* **6**, 132-137.

Wagner, R.A. and M.J. Fischer. 1974. "The String-to-String Correction Problem." *Journal of ACM* **21**, 168-173.