Approximate Matching of Network Expressions with Spacers*

Eugene W. Myers

TR 92-5

ABSTRACT

Two algorithmic results are presented that are pertinent to the matching of patterns of interest in macromolecular sequences. The first result is an output sensitive algorithm for approximately matching network expressions, i.e., regular expressions without Kleene closure. This result generalizes the $O(kn)$ expected-time algorithm of Ukkonen for approximately matching keywords [Ukk85]. The second result concerns the problem of matching a pattern that is a network expression whose elements are approximate matches to network 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.

Key words: Approximate Match, Backtracking, Network Expression, Proximity Search

January 16, 1992

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported in part by the National Institutes of Health under Grant R01 LM04960-01 and the Aspen Center for Physics.

Approximate Matching of Network Expressions with Spacers

0. 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., [MMK85, PBP89]). In general, the spacing between the signals varies from one instance to the next, as do the signals themselves. This motivates a class of patterns in which one is searching for approximate matches to a consensus pattern for each signal, separated within certain distance ranges of each other. A software system called *ANREP* for recognizing such patterns [MeM91] has been built. This paper focuses on the formal discrete pattern matching problems underpinning this system. This introduction presents the necessary background concepts and introduces an idealized version of the problem actually faced in practice.

To begin, one needs the concept of an approximate match to a pattern (say, a regular expression) R over alphabet Σ. But this first requires introducing 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. Obviously, there are a tremendous number of distinct alignments between A and B. What is desired are those 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 [Lev66, NeW70, WaF74].

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 presented an $O(np)$ algorithm for approximately matching sequence A to regular expression R where p is the length of R [MyM89]. 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. Directly, a network expression is any pattern built up from concatenation and union operations. Approximately matching networks is particularly easy (as will be seen momentarily) and the first algorithm is attributable to Sankoff and Kruskal [SaK83]. Hereafter, a network expression when coupled with a threshold will be termed a *motif*.

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 the 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 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 $i_k - i_{k-1} \in [l, r]$. This two-tiered problem of matching network expressions of motifs and spacers is a formal embodiment of the pattern matching capability built into our *ANREP* software system for the analysis of biosequences [MeM91].

In this paper, two results of algorithmic interest for the problem of matching nets are presented. The first is an output sensitive algorithm for matching motifs that generalizes Ukkonen's $O(kn)$ algorithm for approximately matching keywords [Ukk85]. This is presented in Section 2, after Section 1 which reviews the traditional solution to this problem cast in an automatic-theoretic form relevant to the second result presented in Section 3. There, a backtracking algorithm for matching net patterns is presented that picks a backtracking order that minimizes the expected time spent finding a match.

1. 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 [HoU79]. 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. 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 $L_F(\phi)$.

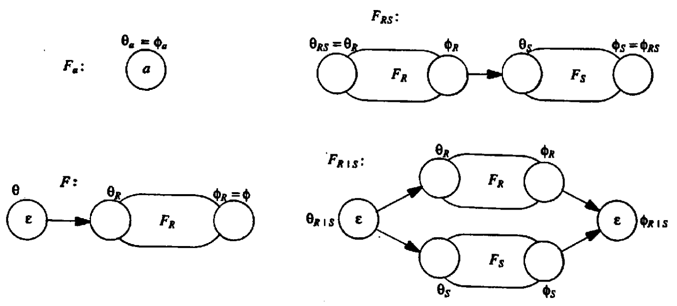

Figure 1: Constructing the ε-NFA for network expression R.

Any network expression, R, can be converted into an equivalent finite automaton F with the inductive construction depicted in Figure 1. 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.

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) $|V| \le 2|R|$, 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. Finally, of essential importance to the output-sensitive algorithm of Section 2, 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, 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 [MyM89]. 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 2. 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 $s \ne \theta$, then for each state t such that $t \to s$, there is a *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 $s \ne \theta$, then, for each state t such that $t \to s$, 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 2 to avoid cluttering the graph.

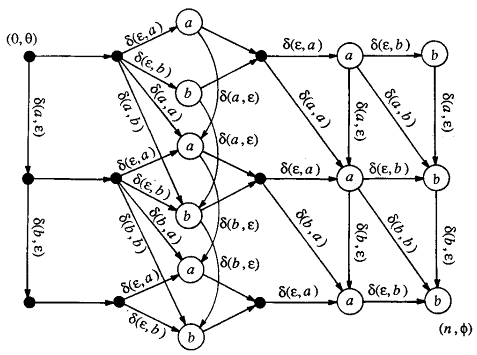

Figure 2: The alignment graph for $A = ab$ versus $R = (a \mid b)\, ab$.

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

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

When $i=0$ 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 s. 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. Certainly this is sufficient to determine if $A \in L_\delta(R,T)$ since it simply suffices to check if $C(m,\phi) \le T$. 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 noticed a decade ago [Sel80], it suffices to simply modify the boundary of the recurrence 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) is identical to that already studied for the matching of regular expressions. This issue will be explored further in Section 3.

Consider the following final problem: given a set of indices $J \subseteq [0,n]$ of potential left ends, determine the set $Scan(M,J) = \{ i : \exists j \in J \text{ s.t. } A_{j..i} \in L_\delta(R,T) \}$ of right ends of approximate matches to M having a left end in J. Figure 3 gives pseudo-code for solving this problem and within it all attributes for motif M are prefixed by "$M.$", e.g., $M.V$ is the set of states of M's ε-NFA, $M.\theta$ is its start state, etc. Generalizing from the previous paragraph, what needs to be done is to set $C(j,\theta)$ to be 0 for exactly those $j \in J$ and none others. Assuming the indices in J are sorted, it suffices to place the "machine" in the start state (by calling $Start(M)$) and then scanning A from left-to-right starting with symbol $a_{J[1]+1}$. The state of the machine, $M.C$, is advanced over each character with a call to $Advance(M,a,inject)$ where a 0 is "injected" into the θ-state whenever an index in J is traversed. The routine *Advance* returns a scalar value *yes*, *no*, or *never* according to whether the ϕ-vertex of the new state is within the threshold $M.T$, above it, or will always be above it unless another inject occurs. This last condition is true iff every vertex in the current state has value above the threshold because δ is non-negative. Knowing the *never* status is useful because once the largest index in J has been passed and *never* is returned, one can safely stop the search.

```
Start(M)
  {  M.C[M.θ] ← 0
     for s ∈ M.V−M.θ in topological order do
           M.C[s] ← min_{t→s} {M.C[t] + δ(ε, λ_s)}
  }

Advance(M, a, inject): (yes, no, never)
  {  M.D[M.θ] ← M.C[M.θ] + δ(a, ε)
     if inject then
           M.D[M.θ] ← 0
     for s ∈ M.V−M.θ in topological order do
           M.D[s] ← min { min_{t→s} {M.C[t] + δ(a, λ_s)}, min_{t→s} {M.D[t] + δ(ε, λ_s)}, M.C[s] + δ(a, ε) }
     M.C ← M.D
     if ∀s, M.C[s] > M.T then
           return never
     else if M.C[M.φ] > M.T then
           return no
     else
           return yes
  }

Scan(M, J): set of [1..n]
  {  I ← ∅
     k ← 2
     Start(M)
     for i ← J[1]+1 to n do
        {  answer ← Advance(M, a_i, i = J[k])
           if k ≤ |J| and i = J[k] then k ← k+1
           if answer = yes then I ← I ∪ {i}
           if answer = never and k > |J| then break
        }
  }
```

Figure 3: Scanning Routines for Matching Motifs.

2. An Output 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 presented such an algorithm for sequence versus sequence comparison under non-negative δ [Fic84]. Ukkonen [Ukk85] 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 can be considered to be output 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. Nontheless, this treatment will continue to adhere to the output 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 output sensitive algorithm for approximate network expression matching under non-negative δ is developed.

Let $C_i(s)$ be the value of vertex (i, s) in whatever row, C_i, a motif recognizer finds itself in after scanning a_i. Let $T_i = \{ s : C_i(s) \le T \}$ be the set of values in row C_i that are within the threshold T. Suppose at this point that one has somehow managed to arrive at a set $Z_i \supseteq T_i$ and values $C_i^*(s)$ for $s \in Z_i$ such that if $C_i(s) \le T$ then $C_i^*(s) = C_i(s)$ and $C_i^*(s) > T$ otherwise. Such a set and its values is called a *zone* and it correctly models all the values of C_i within T. The goal is to advance the motif recognizer over symbol a_{i+1} to its next state, C_{i+1}, but only computing enough of this new row to arrive at a new zone, Z_{i+1}/C^*_{i+1}, encompassing the values within threshold T. Of course the challenge is to keep the zone as small as possible (a trivial solution would be to let $Z = V$). Our algorithm requires that the subgraph of F induced by Z_i be connected. It would be desirable for the zone to be as small as possible with respect to this criterion, but this is not possible. However, Z_i is guaranteed to be minimal in that the removal of any vertex in $Z_i - T_i$ destroys connectedness.

Given a connected zone Z_{i-1}/C^*_{i-1} and symbol a_i our problem is to compute a 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 $Inserts(X) = \{ s : \exists t \in X, t \rightarrow s \text{ or } \exists t \in Inserts(X), (t \rightarrow s \text{ and } C_i(t) \leq T) \}$. Certainly U_i is connected and if one computes C^*_i over the edges in the subgraph of the alignment graph induced by $Z_{i-1} \cup U_i$, then it will properly model C_i over this zone. After computing U_i all that remains to arrive at Z_i is to remove states in $U_i - T_i$ until a set that is minimal with respect to connectedness is reached. This two step process generalizes Ukkonen's algorithm for approximate keyword matching in that it maps into exactly that algorithm when the network expression is a single keyword.

Algorithmically, the first step involves computing C^*_i for the vertices in row i whose states are in Z_{i-1}. If any vertex is discovered to have a value not greater than T, then recursively its successors in row i are evaluated. The difficulty is that this closure computation of $Inserts(Z_{i-1})$ must be interwoven with the correct computation of C^*_i at each vertex and this requires that the closure be discovered in topological order of F's states. For example, suppose s and t are in Z_{i-1} and that there are paths from both states to another $u \notin Z_{i-1}$. If all three states are in U_i then the correct value at (i, u) may come from either (i, s) or (i, t). Thus in order to guarantee the correct value all predecessors of u on both paths must have their values computed first. This difficulty does not arise in the case of a keyword.

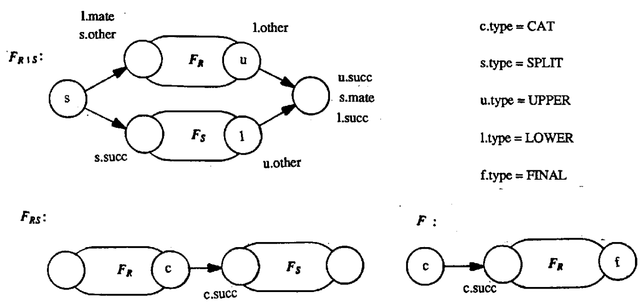

Figure 4: Type and Link Definitions for Series/Parallel F.

Figure 5 gives the pseudo code for our output sensitive algorithm for $Advance(M, a, inject)$. For simplicity, M is implied and the *inject* option is assumed to be **false**. The algorithm assumes that every state s has a type, *s.type*, and links *s.succ*, *s.other*, and *s.mate* to immediate successors and other operationally important states as defined inductively in Figure 4. Our method for handling the ordering problem depends critically on the fact that F is a series/parallel digraph as reflected in the definition of *mate*, and the categorization of state types into the classes *CAT*, *SPLIT*, *LOWER*, *UPPER*, and *FINAL*. The algorithm further assumes that the vertices in F are numbered in the topological ordering uniquely mandated by the requirement that the states in the automaton F_R have higher number than those in F_S in the machine for $F_{R|S}$. For example, this order implies *s.succ* comes before *s.other* for

s of type $SPLIT$. The key to visiting the states in U_i in topological order will be to have available for each state s, the largest (with respect to the topological order) state in Z_{i-1} that is "dominated" by s. A state is dominated by s if and only if every path from θ to it passes through s. Formally, for a set $Z \subseteq V$, let $Dom_Z(s) = \max\{\, t : t \in Z$ and s dominates $t\,\}$. As an example of how this dominator information will be used, consider states $x = s.succ$ and $y = s.other$ where s is a state of type $SPLIT$. Suppose that x is in a list Z of states of F in topological order, y is not in the list, and y needs to be inserted into its correct place in the list. This place is exactly after the element $Dom_Z(x)$.

```
# Z ≡ Z_{i−1} in topological order, C[s] ≡ C*_{i−1}(s), In[s] ≡ s ∈ Z_{i−1}, Dom[s] ≡ Dom_{Z_i}(s) #

U ← ∅
while Z ≠ ∅ do
  {  s ← pop Z
     D[s] ← min { min_{t→s} {C[t] + δ(a, λ_s)}, min_{t→s} {D[t] + δ(ε, λ_s)}, C[s] + δ(a, ε) }
     if In[s] or D[s] ≤ 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 ∈ {SPLIT, CAT, UPPER} then
               if top(Z) ≠ 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] ← true
     push s onto U
  }

# U ≡ U_i in reverse topological order, D[s] ≡ C*_i(s), In[s] ≡ s ∈ U_i #

while U ≠ ∅ do
  {  s ← pop U
     C[s] ← D[s]
     In[s] ← C[s] ≤ T or s.type ∈ {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] ← Dom[s.mate]
               else if In[s.other] then  Dom[s] ← Dom[s.other]
               else if In[s.succ] then   Dom[s] ← Dom[s.succ]
               else                      Dom[s] ← s
          else if s.type = CAT then
               if In[s.succ] then Dom[s] ← Dom[s.succ]
               else               Dom[s] ← s
          else
               Dom[s] ← s
       }
  }

# Z ≡ Z_i in topological order, C[s] ≡ C*_i(s), In[s] ≡ s ∈ Z_i, Dom[s] ≡ Dom_{Z_i}(s) #
```

Figure 5: An Output Sensitive Algorithm for $Advance(M, a_i, \mathbf{false})$.

At the start of the algorithm of Figure 5, Z is a linked list of the states in Z_{i-1} in topological order, $C[s]$ contains the value $C^*_{i-1}(s)$ for every $s \in Z_{i-1}$, the boolean indicator $In[s]$ is true iff $s \in Z_{i-1}$, and $Dom[s] = Dom_{Z_{i-1}}(s)$. The first half of the algorithm builds a list U of U_i in reverse topological order and simultaneously computes $D[s] = C^*_i(s)$ for every $s \in U_i$. This is accomplished by "spilling" states, s, from the top of list/stack Z into the top of stack U, and at that instant evaluating $D[s]$ with the central recurrence, and adding s's immediate successors to their appropriate place in list Z if they are in $U_i - Z_{i-1}$. Such a state's successors belong in U_i if either the state is in Z_i or $C_i(s) \le T$ (the predicate '$In[s]$ **or** $D[s] \le T$' in Figure 5). The tricky part is putting the successor states into the right place in Z. The case where $s.type = SPLIT$ is treated as an example; the remaining cases are left to the reader. Let $x = s.succ$ and $y = s.other$. If x and y are both in Z_{i-1} then neither needs to added since they are already in Z. If x is not in Z_{i-1} then regardless of y, it should be added immediately after s since it is s's immediate successor 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 y is not in Z_{i-1} then where it goes depends on x: if x is not in Z_{i-1} then it should be placed immediately after x because none of the states dominated by x are on the list; otherwise it should be inserted immediately after $Dom_{Z_{i-1}}(x) = Dom[x]$. The reader should observe that the pseudo-code of Figure 5 has exactly the effect above, but the logic is rearranged to minimize the length of the algorithm.

After the first half of the algorithm of Figure 5 has completed, 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 half of the algorithm, states are spilled from the stack U onto the now empty stack Z with some elements being dropped if they are not needed to maintain the connectedness property of Z_i. At the same time, the new dominator information for Z_i is computed, D-values are transferred to C, and In is established for Z_i. A state s in U is transferred to Z only if (1) $D[s] \le T$, (2) s has at least one successor already placed in Z and that successor has s as its sole predecessor, or (3) s has a successor in Z and all other predecessors of that successor were not transferred to Z. The minimal connectedness of the Z_i that results follows. Dominators are easily computed as a function of the dominators of immediate successors also in Z_i as given in Figure 5.

The time complexity of our algorithm is hard to characterize in terms of simple parameters. Certainly, the time spent in expectation is $O(tn)$ where t is the average number of entries in a minimally connected zone with respect to threshold T. Certainly the two t's are correlated. In the event that T is stringent, our output sensitive algorithm is significantly faster than the straightforward dynamic programming algorithm. Specifically, in our computing environment if $t \le \frac{1}{2}V$ then our algorithm is faster in practice. Moreover, our algorithm reduces to exactly Ukkonen's algorithm when the network expression is a single keyword. That is, our algorithm is $O(kn)$ for approximate keyword matching under the unit cost model with k errors. Under this cost model, our algorithm can also be shown to take no more than $O(min\{|\Sigma|^k, kw\}n)$ 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.

3. An Optimized Backtracking Net Matching Algorithm

Faced with the problem of matching a net pattern, one has several choices. For one, 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 in a "monolithic" fashion. But such an approach for large nets seems artificial and is potentially quite costly because the amount of time spent on a spacer is proportional to the value of its integers. Given the desire for negative spacers and the fact that scanner-based routines such as those in Figure 3 are available for recognizing motifs, our approach considers the problem as a composite one of finding net matches given subroutines for matching motifs. It is worth noting that this problem reduces to "proximity search" when motifs are exact matches to keywords [MaB91].

It is a fairly simple exercise to extend the development of the algorithm in Section 2, so that the routines *Start* and *Advance* of Section 1 may be replaced with output sensitive versions. The performance of these algorithms 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 expected 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 expected amount of time, x, it will take for a recognizer in an "average" row configuration to be advanced without injection until it returns *never*. 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. 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.

3.1. 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. For this particular order, the time to perform a search of A is expected to be:

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

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 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. 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:

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

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. This non-trivial 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*

† 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.

‡ 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.

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. Nonetheless, in expectation the extension process quickly fails on many of the possible extension pathways so that the time for the backtracking process is still reasonable. 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):

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

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† over the series-parallel automaton of net N where the cost 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.

3.2. Finding and Reporting Matches

To finish the treatment of nets, consider the problem of finding and reporting a match to the linear net, $M_{-p}S_{-p}M_{-(p-1)}\cdots M_{-1}S_{-1}M_0S_1M_1\cdots M_{q-1}S_qM_q$, 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 preceeding 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‡ and scanning A in reverse, one obtains an analogous routine that computes $Scan^r(M,J) = \{\, i : \exists j \in J \text{ s.t. } A_{i..j} \in L_\delta(R,T) \,\}$. 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 $Scan(M,J) = \{\, i : \exists j \in J \text{ s.t. } A_{i..j} \sim M \,\}$. Finally assume the functions $Space_k(J) = \cup_{j\in J}\,[j+l_k, j+r_k]$ and $Space^r_k(J) = \cup_{j\in J}\,[j-l_k, j-r_k]$ 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 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 $L_k = \{\, i : \exists j \in [a,b] \text{ s.t } A_{i..j} \sim M_{-k}\cdots M_0 \,\}$ and the set $R_h = \{\, i : \exists j \in [a,b] \text{ s.t } A_{j..i} \sim S_1 \cdots M_h \,\}$. That is, L_k contains the left ends of matches to the subnet $M_{-k}\cdots M_0$

---

† 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.

‡ 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, $(a(b\,|\,cd)e)^r = e(b\,|\,dc)a$.

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 6. 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.

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 6: Finding and Reporting a Match to a Linear 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, $L_k = \{ i : \exists j \in [a,b]$ and $l \le i$ $s.t$ $A_{l..i} \sim M_{-k} \cdots S_{-(k+1)}$ 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 $R_h = \{ i : \exists j \in [a,b]$ and $r \ge i$ $s.t$ $A_{j..i} \sim S_1 \cdots M_h$ 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, $Scan(M_0, L_0)$.

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 6. 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 $L_k = \{ i : \exists j \in [a,b], l \le i,$ and $r \ge j$ $s.t$ $A_{l..i} \sim M_{-k} \cdots S_{-(k+1)}$, $A_{i..j} \sim M_{-k} \cdots M_0$, 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 $R_h = \{ i : \exists j \in [a,b], l \le i,$ and $r \ge j$ $s.t$ $A_{l..j} \sim M_{-p} \cdots M_0$, $A_{j..i} \sim S_1 \cdots M_h$ 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 outputing 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 mimimal. Notice that because of spacing contraints 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 essense 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 is j is in L_k. In order that a match acheiving this score can be output, the algorithm simultaneously records the traceback 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 acheiving 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, acheived by the code fragment in Figure 7, 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 $[j - l_{-(k+1)}, j - r_{-(k+1)}]$ 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 ← min{ j : j ∈ Scan(M_-(k+1), L_k+1) }
hgh ← max{ j : j ∈ Scan(M_-(k+1), L_k+1) }
for i ← low to hgh do
      B[i] ← ∞
for j ∈ L_k+1 do
      for i ∈ Scan(M_-(k+1), {j}) do
            if Score_L_k+1[j] + Value(i) < B[i] then
              {   B[i] ← Score_L_k+1[j] + Value(i)
                  I[i] ← j
              }
Heap ← ∅
rgt ← low−1
lft ← ∞
for j ∈ L_k in increasing order do
  {   for i ← max{lft, low} to j − l_-(k+1)−1 do
            if B[i] < ∞ then
                  delete i from Heap
      for i ← rgt+1 to min{j − r_-(k+1), hgh} do
            if B[i] < ∞ then
                  add i to Heap with priority B[i]
      lft ← j − l_-(k+1)
      rgt ← j − r_-(k+1)
      i ← extract min from Heap
      Score_L_k[j] ← B[i]
      Left_L_k[j] ← I[i]
      Right_L_k[j] ← i
  }
```

Figure 7: 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 traceback 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 [MyM89].

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 3.1. Observe that the algorithm of Figure 6, which includes the range reporting sub-algorithm, makes a number of scans that together span a range of symbols approximately as long as 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 7 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 $O(m(p^2 + \Delta \log \Delta))$.

References

[Fic84] Fickett, J.W., "Fast optimal alignment," *Nucleic Acids Research* **12** (1984), 175-179.

[HoU79] Hopcroft, J.E. and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, 1979), 13-76.

[Lev66] Levenshtein, V. I., "Binary codes capable of correcting deletions, insertions, and reversals," *Cybernetics and Control Theory* **10** (1966), 707-710.

[MaB91] U. Manber and R. Baeza-Yates, "An algorithm for string matching with a sequence of don't cares," *Information Processing Letters* **37** (1991), 133-136.

[MeM91] Mehldau, G. and E.W. Myers, "A system for pattern matching applications on biosequences," Technical Report TR91-31, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721.

[MMK85] Miller, J., A.D. McLachlan and A. Klug, "Repetitive zinc-binding domains in the protein transcription factor IIIA from Xenopus oocytes," *EMBO Journal* **4** (1985), 1609-1614.

[MyM89] Myers, E.W. and W. Miller, "Approximate matching of regular expressions," *Bull. of Math. Biol.* **51** (1989), 5-37.

[NeW70] Needleman, S.B. and C.D. Wunsch, "A general method applicable to the search for similarities in the amino-acid sequence of two proteins," *J. Molecular Biology* **48** (1970), 443-453.

[PBP89] Posfai, J., A.S. Bhagwat, G. Posfai, and R.J. Roberts, "Predictive motifs derived from cytosine methyltransferases," *Nucleic Acids Research* **17** (1989), 2421-2435.

[SaK83] Sankoff, D. and J. B. Kruskal, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison* (Addison-Wesley, 1983), 265-310.

[Sel80] Sellers, P.H., "The theory and computation of evolutionary distances: pattern recognition," *J. Algorithms* **1** (1980), 359-373.

[Ukk85] Ukkonen, E., "Finding approximate patterns in strings," *J. of Algorithms* **6** (1985), 132-137.

[WaF74] Wagner, R.A. and Fischer, M.J., "The String-to-String Correction Problem," *Journal of ACM 21* (1974), 168-173.