APPROXIMATE MATCHING OF REGULAR EXPRESSIONS

□ EUGENE W. MYERS†
Department of Computer Science
University of Arizona
Tucson, AZ 85721

□ WEBB MILLER
Department of Computer Science
The Pennsylvania State University
University Park, PA 16802

Given a sequence A and a regular expression R, the *approximate regular expression matching* problem is to find a sequence matching R whose optimal alignment with A is the highest scoring of all such sequences. This paper develops an algorithm to solve the problem in time $O(MN)$, where M and N are the lengths of A and R. Thus, the time requirement is asymptotically no worse than for the simpler problem of aligning two fixed sequences. Our method is superior to an earlier algorithm by Wagner and Seiferas in several ways. First, it treats real-valued costs, in addition to integer costs, with no loss of asymptotic efficiency. Second, it requires only $O(N)$ space to deliver just the score of the best alignment. Finally, its structure permits implementation techniques that make it extremely fast in practice. We extend the method to accommodate gap penalties, as required for typical applications in molecular biology, and further refine it to search for substrings of A that strongly align with a sequence in R, as required for typical data base searches. We also show how to deliver an optimal alignment between A and R in only $O(N + \log M)$ space using $O(MN \log M)$ time. Finally, an $O(MN(M+N) + N^2 \log N)$ time algorithm is presented for alignment scoring schemes where the cost of a gap is an arbitrary increasing function of its length.

1. Introduction. Databases of known DNA and protein sequences have expanded steadily for a number of years and are now poised for a period of explosive growth. Proper utilization of biosequence data requires efficient methods to search for sequences that match a given pattern or template. The patterns known as *regular expressions* have proven useful in this context (Abarbanel, *el. al.*, 1984; Cohen, *el. al.*, 1986) and efficient search algorithms for them are available (Aho, 1980; Miller, 1987). Often, geneticists are interested in *approximate*, as well as exact, matches. For example, given a regular expression describing a set of sequences known to form a given activation site, one may find new sequences by searching for ''close'' matches to the regular expression.

The problem of comparing sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$, which is a special case of approximate regular expression matching, has been studied extensively (Sankoff and Kruskal, 1983; Waterman, 1984). Two formulations of the problem are common. First, one can ask for a highest-scoring alignment of A and B. The alternative is to seek a minimum-cost set of deletion, insertion, and substitution operations that converts A to B. The formulations are ''dual'' in the sense that a method for solving one can be adapted to give a method for solving the other (Smith, *et. al.*, 1981). This paper is couched in terms of alignments because it is the more popular framework for applications in molecular biology.

Section 2 reviews the well-known dynamic programming algorithms for optimally aligning a pair of sequences under two common scoring models. In the *column-sum cost model*, an alignment's score is merely the sum of the scores of its columns (Sankoff and Kruskal, 1983). In the more general model (Gotoh, 1982), an additional penalty is levied for each ''gap'' of the alignment. This *gap-penalty cost model* is appropriate for certain applications in

† This work was supported in part by NSF Grant DCR-8511455.

biology (Fitch and Smith, 1983) and computer science (Miller and Myers, 1988a; Myers and Miller, 1988a). In preparation for the paper's main results, Section 2 describes the alignment algorithms as computing maximum-scoring source-to-sink paths in certain weighted and directed *edit graphs*.

The theory of regular expressions (Hopcroft and Ullman, 1979) is another traditional area that this paper builds upon. Section 3 reviews regular expressions and their conversion into equivalent non-deterministic finite state machines. The section also gives the special properties of the constructed finite state machines that are exploited in this paper.

The basic problem of comparing two sequences has been generalized in a number of directions. One extension allows one or both of the sequences to be replaced by a ''directed network'' (Sankoff and Kruskal, 1983, pp. 265-274). This problem then extends further to give the approximate regular expression matching problem (Wagner, 1974; Wagner and Seiferas, 1978). The crucial difference between the two extensions is that a directed network is an acyclic graph, whereas the graph of a finite state machine can have cycles. This implies that the underlying edit graph models for directed network comparison are acyclic, while those for regular expression comparison are not necessarily so. Thus, a directed network comparison can still be accomplished in a single $O(MN)$ time sweep of its edit graph. On other hand, the presence of cycles in a regular expression edit graph lead Wagner and Seiferas (1978) to compute optimal paths using a discrete version of Dijkstra's shortest path algorithm. Their approach takes time $O(\max(MN, D))$ and space $O(MN)$, where M is the length of the sequence, N is the length of the regular expression, and D is the minimum distance. However, their result requires that the costs of the basic operation are integers, and the straightforward extension to real-valued costs results in an $O(MN \log MN)$ time bound. Moreover, they consider only the column-sum cost model for alignments.

Our approach to approximate regular expression matching is based on the observation that our regular expression edit graphs posses a special property that allows optimal paths to be computed in a single ''listing'' of the graph's vertices. Although a vertex may need to be treated more than once, a small number of ''visits'' are sufficient. In essence, this technique is borrowed from the field of data flow analysis (Hecht, 1977), which is commonly used in the generation of efficient machine instruction sequences by a compiler. Section 4 develops the approach for the column-sum cost model and verifies that it handles arbitrary costs in time $O(MN)$ and, if only the optimal score is desired, space $O(N)$. Section 5 extends the technique to the gap-penalty cost model.

Section 6 discusses the *database query* version of the problem, which requires finding *substrings* of A, the database, that strongly align with a sequence R, the pattern. This variation raises a number of distinctive algorithmic and implementation issues. With regard to implementation, we advocate ''compiling'' the regular expression into a one-of-a-kind program (Thompson, 1968; Pennello, 1986), rather than following the traditional interpretive approach (Miller, 1987) of generating data structures that control the actions of a general-purpose program. Such an approach is possible with our method, and not with that of Wagner and Seiferas, because the order in which vertices are visited is known *a priori*.

Extensions are outlined in Section 7. Based on the algorithm for computing the optimal score in time $O(MN)$ and space $O(N)$, we give an $O(MN \log M)$-time, $O(N + \log M)$-space algorithm that delivers a sequence in R and an alignment between it and A that yields the optimal score. When the regular expression is a directed network, the time reduces to $O(MN)$, thus generalizing the linear space result of Myers and Miller (1988b) and improving the space complexity of Sankoff and Kruskal's (1983, pp. 265-274) result. Finally, we further extend our method to give an $O(MN(M+N) + N^2 \log N)$-time, $O(N(M+N))$-space algorithm for the approximate regular expression matching problem under scoring schemes where gaps are penalized by an arbitrary increasing function of their length.

2. Sequence-vs-sequence comparison.

Suppose the sequences under consideration consist of zero or more symbols chosen from an alphabet Σ. Let ε, a unique symbol not in Σ, denote the sequence of zero symbols. The empty sequence, ε, is the identity element with

respect to the concatenation of sequences, i.e., if v and w are sequences, then $v\varepsilon\, w = vw$.

An *aligned pair* has the form $\begin{bmatrix} a \\ b \end{bmatrix}$, where $a, b \in \Sigma \cup \{\varepsilon\}$. An *alignment* is a sequence of aligned pairs. An alignment *S aligns A and B* if A is the concatenation of upper elements of S and B is the concatenation of lower elements. For example, the sequence of four aligned pairs $\begin{bmatrix} p \\ x \end{bmatrix}\begin{bmatrix} \varepsilon \\ y \end{bmatrix}\begin{bmatrix} q \\ \varepsilon \end{bmatrix}\begin{bmatrix} r \\ z \end{bmatrix}$ aligns *pqr* and *xyz*. Thus, the *null pair*, $\eta = \begin{bmatrix} \varepsilon \\ \varepsilon \end{bmatrix}$, acts as the identity element for the concatenation of alignments, i.e., if α and β are alignments, then $\alpha\eta\beta$ and $\alpha\beta$ are considered equal. Without loss of generality, assume that alignments consist of non-null aligned pairs, and let η denote the zero-length empty alignment.

Underlying both the column-sum and gap-penalty cost models is a user-specified scoring function σ that assigns a real-valued cost to each possible non-null aligned pair. In the column-sum cost model the score of an alignment S is simply the sum of the costs of each aligned pair in it, i.e., $Score_{sum}(S) = \Sigma\{\sigma(\pi) : \pi \text{ is an aligned pair of } S\}$. Term an aligned pair whose upper entry is ε an *insertion pair*, and one whose lower entry is ε a *deletion pair*. An *insertion gap* is a contiguous subsequence of insertion pairs delimited by non-insertion pairs or an end of the alignment. A *deletion gap* is similarly defined, and collectively such blocks are called *gaps*. For example, the alignment $\begin{bmatrix} \varepsilon \\ x \end{bmatrix}\begin{bmatrix} \varepsilon \\ y \end{bmatrix}\begin{bmatrix} q \\ \varepsilon \end{bmatrix}\begin{bmatrix} r \\ z \end{bmatrix}$ has an insertion gap of length two and a deletion gap of length one.† In the gap-penalty cost model, the user specifies a fixed *gap penalty* $g > 0$ in addition to σ. The score of an alignment S is its column-sum score save that each gap is penalized g, i.e., $Score_{gap}(S) = Score_{sum}(S) - g\times$(number of gaps in S).

For sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$, the *column-sum edit graph*, $H_{A,B}$, is an edge-labeled directed graph. The vertices of $H_{A,B}$ are the pairs (i, j) where $i \in [0, M]$ and $j \in [0, N]$. Distinguish $\Theta = (0, 0)$ as $H_{A,B}$'s source vertex, and $\Phi = (M, N)$ as its sink vertex. The following edges, and only these edges, are in $H_{A,B}$.

1. If $i \in [1, M]$ and $j \in [0, N]$, then there is a *deletion* edge $(i-1, j) \to (i, j)$ labeled $\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}$.
2. If $i \in [0, M]$ and $j \in [1, N]$, then there is an *insertion* edge $(i, j-1) \to (i, j)$ labeled $\begin{bmatrix} \varepsilon \\ b_j \end{bmatrix}$.
3. If $i \in [1, M]$ and $j \in [1, N]$, then there is a *substitution* edge $(i-1, j-1) \to (i, j)$ labeled $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$.

Figure 1 provides an example of the construction.

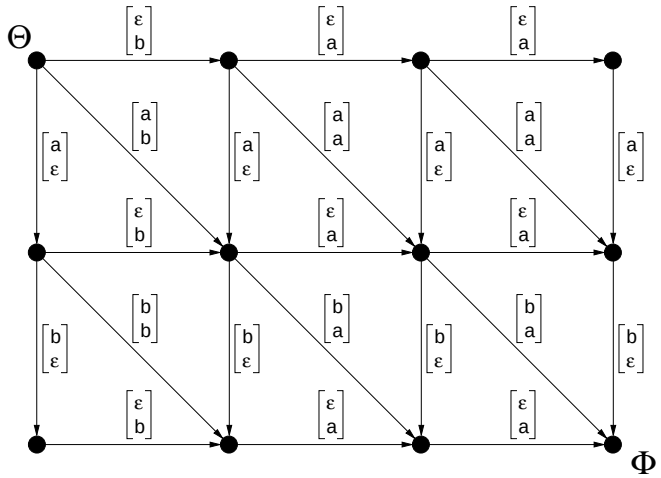


Figure 1: $H_{A,B}$ for $A = ab$ and $B = baa$.

---

† Some authors define a gap as a largest contiguous subsequence of insertion or deletion pairs, so that in the example there would be only one gap of length three. While not treated in this paper, edit graphs and algorithms for this variation can be obtained in a fashion that parallels our treatment.

Let A_i denote the i-symbol prefix of A, i.e., $A_0 = \varepsilon$ and $A_i = a_1 a_2 \cdots a_i$ for $i \in [1, M]$. Similarly, B_j is the j-symbol prefix of B. A path in $H_{A,B}$ is said to *spell* the alignment obtained by concatenating its edge labels. A path containing zero edges is assumed to spell η. A basic exercise reveals that each path from Θ to (i, j) spells an alignment of A_i and B_j, and a different path spells a different alignment. Thus, there is a one-to-one correspondence between paths in $H_{A,B}$ from Θ to Φ and alignments of A and B.

Our goal is an algorithm that computes an optimal alignment of A and B under the column-sum cost model. Weight $H_{A,B}$ by assigning cost $\sigma(\pi)$ to each edge labeled π. Note that none of the edges are labeled η in column-sum edit graphs, so the weighting is well-defined. Moreover, the sum of the weights of a path's edges is exactly the sum of the scores of the aligned pairs in its corresponding alignment. Thus the problem reduces to computing a maximum-weight path from Θ to Φ in $H_{A,B}$ as weighted by σ.

Because $H_{A,B}$ is acyclic, a maximum-weight path can be determined in a single pass over its vertices, so long as they are taken in a *topological order*, i.e., an ordering of the vertices with the property that every edge is directed from a vertex to a successor in the ordering. One topological order for $H_{A,B}$'s vertices is to treat the rows in order, sweeping left to right within a row. Using this vertex ordering, the following procedure computes the score, $C(i, j)$, of a maximum-weight path from Θ to each vertex (i, j) where $i \in [0, M]$ and $j \in [0, N]$.

$C(0, 0) \leftarrow 0$
for $j \leftarrow 1$ **to** N **do**
    $C(0, j) \leftarrow C(0, j-1) + \sigma(\left[\begin{smallmatrix}\varepsilon\\ b_j\end{smallmatrix}\right])$
for $i \leftarrow 1$ **to** M **do**
  { $C(i, 0) \leftarrow C(i-1, 0) + \sigma(\left[\begin{smallmatrix}a_i\\ \varepsilon\end{smallmatrix}\right])$
    for $j \leftarrow 1$ **to** N **do**
        $C(i, j) \leftarrow \max\{C(i-1, j) + \sigma(\left[\begin{smallmatrix}a_i\\ \varepsilon\end{smallmatrix}\right]),\ C(i, j-1) + \sigma(\left[\begin{smallmatrix}\varepsilon\\ b_j\end{smallmatrix}\right]),\ C(i-1, j-1) + \sigma(\left[\begin{smallmatrix}a_i\\ b_j\end{smallmatrix}\right])\}$
  }
write "Similarity score is" $C(M, N)$

Figure 2: A sequence alignment algorithm for the column-sum cost model.

Now attention is turned to the second algorithm, which accommodates gap penalties. The corresponding *gap-penalty edit graph*, $H^+_{A,B}$, is more complex than before. For each integer pair (i, j) there are three vertices: one of type C(ommon), one of type D(eletion), and one of type I(nsertion). Specifically, the vertices of $H^+_{A,B}$ are the pairs: $(i, j)_C$, $(i, j)_D$, and $(i, j)_I$ for $i \in [0, M]$ and $j \in [0, N]$. Distinguish $\Theta = (0, 0)_C$ as $H^+_{A,B}$'s source vertex, and $\Phi = (M, N)_C$ as its sink vertex. The following edges, and only these edges, are in $H^+_{A,B}$.

1. If $i \in [1, M]$ and $j \in [0, N]$, then there is a *deletion* edge $(i-1, j)_D \rightarrow (i, j)_D$ labeled $\left[\begin{smallmatrix}a_i\\ \varepsilon\end{smallmatrix}\right]$.
2. If $i \in [1, M]$ and $j \in [0, N]$, then there is a *deletion initiation* edge $(i-1, j)_C \rightarrow (i, j)_D$ labeled $\left[\begin{smallmatrix}a_i\\ \varepsilon\end{smallmatrix}\right]$.
3. If $i \in [0, M]$ and $j \in [1, N]$, then there is an *insertion* edge $(i, j-1)_I \rightarrow (i, j)_I$ labeled $\left[\begin{smallmatrix}\varepsilon\\ b_j\end{smallmatrix}\right]$.
4. If $i \in [0, M]$ and $j \in [1, N]$, then there is an *insertion initiation* edge $(i, j-1)_C \rightarrow (i, j)_I$ labeled $\left[\begin{smallmatrix}\varepsilon\\ b_j\end{smallmatrix}\right]$.
5. If $i \in [1, M]$ and $j \in [1, N]$, then there is a *substitution* edge $(i-1, j-1)_C \rightarrow (i, j)_C$ labeled $\left[\begin{smallmatrix}a_i\\ b_j\end{smallmatrix}\right]$.
6. If $i \in [0, M]$ and $j \in [0, N]$, then there is a *null* edge $(i, j)_D \rightarrow (i, j)_C$ labeled η.
7. If $i \in [0, M]$ and $j \in [0, N]$, then there is a *null* edge $(i, j)_I \rightarrow (i, j)_C$ labeled η.

Note that the *D*-vertices in row 0 and the *I*-vertices in column 0 are not reachable from Θ. Figure 3 illustrates $H^{+}_{ab,\,baa}$. All edge not annotated are null edges labeled η.

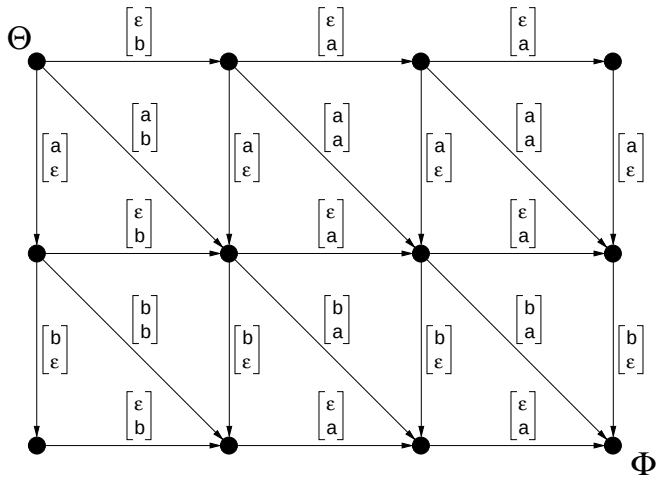


Figure 3: $H^{+}_{A,\,B}$ for $A = ab$ and $B = baa$.

Since η is the identity for alignment concatenation, a path in $H^{+}_{A,\,B}$ may be thought of as spelling the alignment obtained by concatenating its non-null labels. As with $H_{A,\,B}$, each path from Θ to $(i,\,j)_C$ spells an alignment between A_i and B_j, and every such alignment is spelled by some path. This follows easily since for every edge $v \rightarrow w$ labeled π in $H_{A,\,B}$, there is a path in $H^{+}_{A,\,B}$ from v_C to w_C spelling π. However, there may be many paths to $(i,\,j)_C$ spelling the same alignment. So the correspondence is not one-to-one in this framework unless one restricts attention to a canonical subset of the paths. A path is *normal* if and only if it does not contain subpaths of the form $(i-1,\,j)_D \rightarrow (i-1,\,j)_C \rightarrow (i,\,j)_D$ or $(i,\,j-1)_I \rightarrow (i,\,j-1)_C \rightarrow (i,\,j)_I$. An exercise, not proven here, shows that alignments between A_i and B_j are in one-to-one correspondence with normal paths from Θ to $(i,\,j)$.

$H^{+}_{A,\,B}$ is weighted as follows. Deletion initiation and insertion initiation edges are weighted $\sigma(\pi) - g$ where π is the aligned pair labeling the edge. Null edges are weighted 0 and all other edges are weighted σ(π). The weight of a path under this scheme is the sum of the scores of its non-null labels minus *g* times the number of initiation edges. For normal paths this is exactly the gap-penalty score of its corresponding alignment since each initiation edge corresponds to the leftmost aligned pair in a gap. Hence the problem is to compute a maximum-weight normal path from Θ to Φ. However, for every non-normal path there is a normal path of greater weight spelling the same alignment. This is simply because a subpath such as $(i-1,\,j)_D \rightarrow (i-1,\,j)_C \rightarrow (i,\,j)_D$ can be replaced by $(i-1,\,j)_D \rightarrow (i,\,j)_D$ for a net weight gain of $g > 0$. Thus the problem is to compute a maximum-weight path (it must be normal) from Θ to Φ in $H^{+}_{A,\,B}$ as weighted by σ and *g*.

Because $H^{+}_{A,\,B}$ is acyclic, a maximum-weight path can be determined in a single topological sweep of its vertices. In the algorithm of Figure 4, the values $D(i,\,j)$, $I(i,\,j)$, and $C(i,\,j)$ are the maximum path weights from Θ to $(i,\,j)_D$, to $(i,\,j)_I$, and to $(i,\,j)_C$, respectively. To simplify matters, the unreachable vertices, $D(0,\,j)$ and $I(i,\,0)$, are set to $-\infty$, a suitably large negative constant.

$D(0, 0) \leftarrow I(0, 0) \leftarrow -\infty$
$C(0, 0) \leftarrow 0$
for $j \leftarrow 1$ **to** N **do**
{ $D(0, j) \leftarrow -\infty$
$C(0, j) \leftarrow I(0, j) \leftarrow \max\{I(0, j-1),\ C(0, j-1) - g\} + \sigma(\begin{bmatrix} \varepsilon \\ b_j \end{bmatrix})$
}

for $i \leftarrow 1$ **to** M **do**
{ $I(i, 0) \leftarrow -\infty$
$C(i, 0) \leftarrow D(i, 0) \leftarrow \max\{D(i-1, 0),\ C(i-1, 0) - g\} + \sigma(\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix})$
for $j \leftarrow 1$ **to** N **do**
{ $D(i, j) \leftarrow \max\{D(i-1, j),\ C(i-1, j) - g\} + \sigma(\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix})$
$I(i, j) \leftarrow \max\{I(i, j-1),\ C(i, j-1) - g\} + \sigma(\begin{bmatrix} \varepsilon \\ b_j \end{bmatrix})$
$C(i, j) \leftarrow \max\{D(i, j),\ I(i, j),\ C(i-1, j-1) + \sigma(\begin{bmatrix} a_i \\ b_j \end{bmatrix})\}$
}
}
write "Similarity score is" $C(M, N)$

Figure 4: A sequence alignment algorithm for the gap-penalty cost model.

3. Regular expressions and finite automata. Many pattern matching problems can be phrased in terms of regular expressions, which are built up from individual symbols via union, concatenation, and concatenation-closure operations. Formally, the set of *regular expressions* over a given alphabet, Σ, is defined recursively as follows:

1. If $a \in \Sigma \cup \{\varepsilon\}$, then a is a regular expression.
2. If R and S are regular expressions, then so are $R|\ S$, RS, R^*, and (R).

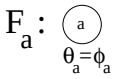
A regular expression determines a language, i.e., a set of sequences of symbols from Σ. In fact, a regular expression is frequently referred to as if it were the language it denotes, as in ''c is a sequence in b^*c^*''. The precise language-defining rules are as follows. If $a \in \Sigma \cup \{\varepsilon\}$, then a denotes the set containing the single sequence a. $R|\ S$ is the union of the languages denoted by R and S. RS denotes the set of sequences obtained by concatenating a sequence in R and a sequence in S. R^* consists of all sequences that can be obtained by concatenating zero or more sequences from R. Finally, (R) denotes the same language as R. Parentheses are used to override the ''natural'' precedence of the operators, which places * highest, concatenation next, and | last. For example, $ab|\ cb^*$ denotes the set, $\{ ab, c, cb, cbb, \cdots \}$.

While regular expressions permit the convenient textual specification of *regular languages*, their finite state machine counterparts are better suited for the task of recognizing the sequences in a language. Several different models of finite automata have appeared, but all are equivalent in power and recognize exactly the class of regular languages. The complexity results of this paper require the use of a non-deterministic, ε-labeled model, and labeling states instead of edges yields simpler software. Formally, a *finite automaton*, $F = < V, E, \lambda, \theta, \phi >$, consists of:

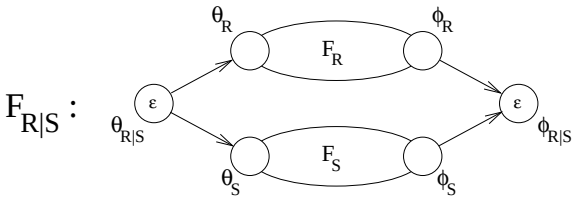
1. A set, *V*, of vertices, called *states*.
2. A set, *E*, of directed edges between states.
3. A function, λ, assigning a ''label'' $\lambda(s) \in \Sigma \cup \{\varepsilon\}$ to each state *s*.
4. A designated ''source'' state, θ, and a designated ''sink'' state, ϕ.

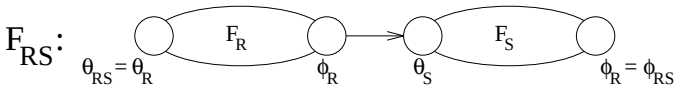
Intuitively, *F* is a vertex-labeled directed graph with distinguished source and sink vertices. A directed path through *F spells* the sequence obtained by concatenating the state labels along the path. Recall that ε acts as the identity element for concatenation, i.e., $v\varepsilon w = vw$. So, one may think of spelling just the non-ε labels on 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 $L_F(\phi)$.

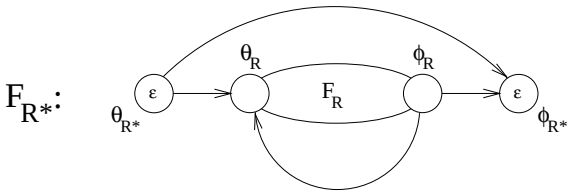
For any regular expression, *R*, the following recursive method constructs a finite automaton, F_R, that accepts exactly the language denoted by *R*. For $a \in \Sigma \cup \{\varepsilon\}$, construct the automaton:



Suppose that *R* is a regular expression and automaton F_R with source θ_R and sink ϕ_R has been constructed. Further suppose that automaton F_S with source θ_S and sink ϕ_S has been constructed for regular expression *S*. Then automata for *R*| *S*, *RS*, and *R** are constructed as follows:





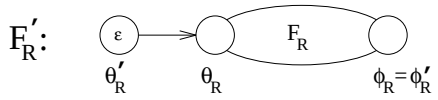


Each diagram precisely specifies how to build the composite automaton from F_R and F_S. For example, F_{RS} is obtained by adding an edge from ϕ_R to θ_S, designating θ_R as its source state, and ϕ_S as its sink.

A straightforward induction shows that automata constructed for regular expressions by the above process have the following properties:

1. The in-degree of θ is 0.
2. The out-degree of ϕ is 0.
3. Every state has an in-degree and an out-degree of 2 or less.
4. $|V| < 2|R|$, i.e., the number of states in F_R is less than twice R's length.

It is both algorithmically and formally simpler to let the sequence spelled by a path be the concatenation of all state labels on the path *except for the first vertex*. For then the sequence is the concatenation of the labels at the head of each *edge* in the path. This alteration is simply accommodated by modifying the automaton F_R produced for R above as follows:



Since the start state of F'_R is labeled with ε, $L_{F'}(s)$ is the same, regardless of which notion of spelling is used. Moreover, this altered automaton satisfies the properties above, save that now $|V| \leq 2|R|$. The key observation is that for any regular expression R there is a non-deterministic, ε-labeled automaton whose size, measured in vertices or edges, is linear in the length of R.

The algorithms to be developed depend on another subtle property of the constructed automata. Call the constructed edges that run left-to-right in the above pictures *DAG edges*. The remaining *back edges* consist of just those from ϕ_R to θ_R in the diagram of F_{R*}. For those familiar with the data flow analysis literature, Part 2 of Lemma 1 asserts that the *loop connectedness parameter* of F's graph is one (Hecht and Ullman, 1975).

Lemma 1. Consider a finite automaton, F, constructed by the above process. (1) If a back edge occurs on a path that begins at θ, then the edge ends at a state that precedes the edge on the path, i.e., the edge closes a loop in the path. (2) Any loop-free path in F has at most one back edge.

Proof. Suppose that F is the automaton for $R*$ and the back edge is the one added when constructing F. Then part 1 clearly holds. But the constructions embedding F as a sub-automaton in a larger automaton, L, guarantee that a path from L's start state that includes the back edge must first pass through F's start state, proving part 1 in general.

Let p be a loop-free path in F and suppose that, contrary to part 2, p contains two back edges. The first back edge in p connects the final state of some sub-automaton, S, to its start state. Consider the suffix, q, of p that follows that back edge. If q leaves S, then a loop would be formed when it exits via S's final state. But since p is loop-free, q must stay within F. However, part 1, when applied to S, implies that q contains no back edges, contradicting the assumption that p contains two back edges. □

4. The column-sum edit graph and algorithm for regular expressions. The approximate regular expression matching problem entails comparing a sequence $A = a_1 a_2 \cdots a_M$ against the sequences in a regular expression R in search of the sequence that is most similar to A under a scoring function σ. In analogy with the edit graph model for comparing two sequences, a graph theoretic model is developed in this section for comparing a sequence and a regular expression. A *regular expression edit graph*, $G_{A,R}$, is an edge-labeled directed graph. It is designed so that paths between two designated vertices correspond to alignments between A and sequences in R, and the maximum-weight path models the solution. Let $F = < V, E, \lambda, \theta, \phi >$ be the finite automaton F'_R for R constructed as in Section 3. The vertices of $G_{A,R}$ are the pairs (i, s) where $i \in [0, M]$ and $s \in V$. Informally, the graph contains $M + 1$ copies of F, where (i, s) is ''in row i.'' Distinguish $\Theta = (0, \theta)$ as the source vertex of $G_{A,R}$, and $\Phi = (M, \phi)$ as its sink vertex. The following edges, and only these edges, are in $G_{A,R}$.

1. If $i \in [1, M]$ and either $s = \theta$ or $\lambda(s) \neq \varepsilon$, then there is a *deletion edge* $(i-1, s) \rightarrow (i, s)$ labeled $\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}$.
2. If $i \in [0, M]$ and $t \rightarrow s \in E$, then there is an *insertion edge* $(i, t) \rightarrow (i, s)$ labeled $\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix}$.
3. If $i \in [1, M]$, $t \rightarrow s \in E$, and $\lambda(s) \neq \varepsilon$, then there is a *substitution edge* $(i-1, t) \rightarrow (i, s)$ labeled $\begin{bmatrix} a_i \\ \lambda(s) \end{bmatrix}$.

Let N be the length of R. Then $|V| \leq 2N$ and $G_{A,R}$ has no more than $2(M+1)N$ vertices. Since F has at most N non-ε states, it follows that $G_{A,R}$ has no more than $M(N+1)$ deletion edges, $4(M+1)N$ insertion edges, and $2MN$ substitution edges, for a total of $7MN + 4N + M$ edges. Thus the ''size'' of the graph is $O(MN)$. Figure 5 gives an example. All edges not annotated are labeled with η and the solid vertices are ε-labeled in F.

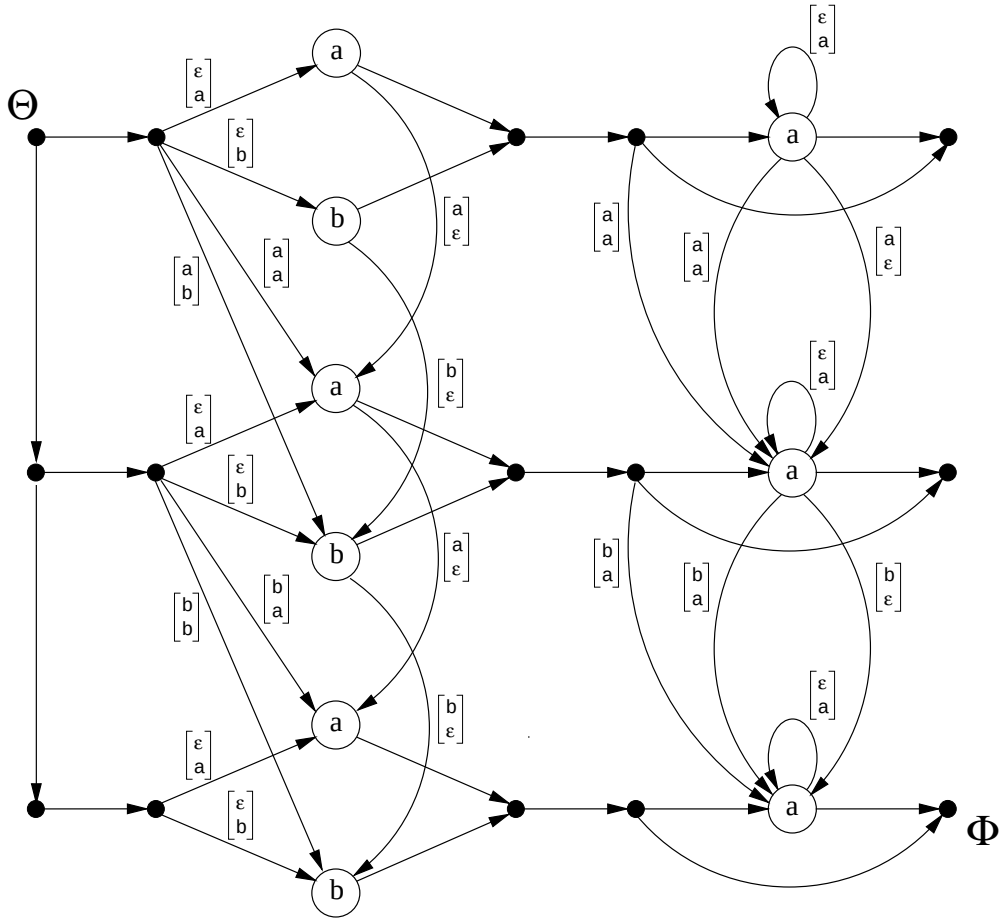


Figure 5: $G_{A,R}$ for $A = ab$ and $R = (a|\ b)\ a^*$.

A path in $G_{A,R}$ *spells* the alignment obtained by concatenating its edge labels. Recall that η acts as the identity for alignment concatenation and so one may think of spelling just the non-null labels on the path. Lemma 2 shows that paths from Θ to a vertex (i, s) spell alignments between A_i and sequences in $L_F(s)$, and, conversely, all such alignments are modeled by paths from Θ to (i, s). An immediate corollary is that the set of paths from Θ to Φ spell exactly the set of alignments between A and sequences in R. Essentially, $G_{A,R}$ is an edge-labeled finite automaton accepting the language of all alignments between A and sequences in R. A construction similar to $G_{A,R}$ was first presented by Wagner and Seiferas (1978).

Lemma 2. There is a path in $G_{A,R}$ from Θ to (i, s) spelling S if and only if S aligns A_i and $B \in L_F(s)$.

Proof. (=>) Let P be a path from Θ to (i, s) spelling S. The upper entry of the label on an edge $(j, t)\rightarrow(j, u)$ is ε, whereas it is a_j for an edge $(j-1, t)\rightarrow(j, u)$. Thus, concatenating the upper entries of S gives A_i. Let p be the sequence, $<t\rightarrow u$: $(j, t)\rightarrow(k, u)$ is an insertion or substitution edge of $P>$, as ordered by P. The construction then guarantees that p is a path from θ to s in F and that concatenating the lower entries of S gives the string $B \in L_F(s)$ spelled by p. Hence, S aligns A_i and B.

(<=) The claim is first proved subject to the constraint that $s = \theta$ or $\lambda(s) \neq \varepsilon$. This restriction will be relaxed in the last paragraph. The argument proceeds by induction on the length of S, i.e., on the number of non-null aligned pairs. For the basis case, i.e., $|S| = 0$, $S = \eta$ and so aligns A_0 and $B = \varepsilon$. Moreover, B can only be in $L_F(\theta)$ because $\lambda(s) \neq \varepsilon$ implies ε is not in $L_F(s)$. But then the path of zero edges from Θ to $(0, \theta)$ spells S. Suppose the result holds for alignments of length $k-1$ and let $|S| = k$, where S aligns A_i and $B \in L_F(s)$ and either $s = \theta$ or $\lambda(s) \neq \varepsilon$.

Case 1: $S = S'\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]$. Then S' aligns A_{i-1} and $B \in L_F(s)$ and, by induction, there is a path from Θ to $(i-1, s)$ spelling S'. Extending this path by the edge $(i-1, s)\rightarrow(i, s)$, which is labeled $\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]$, produces the desired path.

Case 2: $S = S'\left[\begin{smallmatrix} \varepsilon \\ b \end{smallmatrix}\right]$. Then $B \neq \varepsilon$ and so $B \notin L_F(\theta)$. Thus $B \in L_F(s)$ where $\lambda(s) \neq \varepsilon$, and, consequently, where $\lambda(s) = b$. Let $B = Cb$, i.e., C is the bottom row of S'. Then S' aligns A_i and $C \in L_F(t)$, where (i) either $t = \theta$ or $\lambda(t) \neq \varepsilon$ and (ii) there is a path from $t\rightarrow t_1\rightarrow t_2\rightarrow \cdots \rightarrow t_n\rightarrow s$ in F with $\lambda(t_j) = \varepsilon$ for all $j\in[1, n]$. By induction there is a path from Θ to (i, t) spelling S'. Extending this path by the edges $(i, t)\rightarrow(i, t_1)\rightarrow(i, t_2)\rightarrow \cdots \rightarrow(i, t_n)\rightarrow(i, s)$ produces the desired path since the final edge has label $\left[\begin{smallmatrix} \varepsilon \\ b \end{smallmatrix}\right]$ and the other added edges are labeled η.

Case 3: $S = S'\left[\begin{smallmatrix} a_i \\ b \end{smallmatrix}\right]$. Then S' aligns A_{i-1} and $C \in L_F(t)$, where $t, t_1, t_2, \cdots, t_n$ are as in Case 2. By induction there is a path from Θ to $(i-1, s)$ spelling S'. Extending this path by the edges $(i-1, t)\rightarrow(i-1, t_1)\rightarrow(i-1, t_2)\rightarrow \cdots \rightarrow(i-1, t_n)\rightarrow(i, s)$ produces the desired path since the final edge has label $\left[\begin{smallmatrix} a_i \\ b \end{smallmatrix}\right]$ and the other added edges are labeled η.

Finally consider the case where $s \neq \theta$, $\lambda(s) = \varepsilon$, and S aligns A_i and $B \in L_F(s)$. Then there exists a t such that (i) $t = \theta$ or $\lambda(t) \neq \varepsilon$, (ii) $B \in L_F(t)$, and (iii) there is a path $t\rightarrow t_1\rightarrow t_2\rightarrow \cdots \rightarrow t_n\rightarrow s$ in F with $\lambda(t_j) = \varepsilon$ for all $j\in[1, n]$. Because of (i), the induction above gives a path from Θ to (i, t) spelling S. Extending this path by the edges $(i, t)\rightarrow(i, t_1)\rightarrow(i, t_2)\rightarrow \cdots \rightarrow(i, t_n)\rightarrow(i, s)$ gives the desired path since the added edges are all labeled η. □

Under the column-sum cost model, the score of an alignment is the sum of the σ-scores of its aligned pairs. Weight $G_{A,R}$ by assigning weight $\sigma(\pi)$ to each edge labeled $\pi \neq \eta$, and weight zero to edges labeled η, i.e., choose $\sigma(\eta) = 0$. With this weighting scheme, the sum of the weights of a path's edges is exactly the sum of the scores of the (non-null) aligned pairs in its corresponding alignment. So the approximate regular expression matching problem under the column-sum cost model reduces to computing a maximum-weight path from Θ to Φ in $G_{A,R}$ as weighted by σ.

Any cycle in $G_{A,R}$ is contained within a row, so it consists solely of insertion edges. If such a cycle has a net positive weight, then choosing source-to-sink paths that employ this cycle an arbitrary number of times give alignments with A that have arbitrarily large scores. Since each copy of F in $G_{A,R}$ is identically weighted, one can determine the existence of such cycles in $O(N)$ time during the construction of F.† If such a cycle is found then it is reported and the computation terminates. Otherwise, all cycles have non-positive weight, and it follows that removing a cycle from a path can only increase its weight. Thus, without loss of generality, a maximum-weight path from Θ to Φ may be found by considering only those paths that are cycle-free.

The *trace* of a path $P = w_0\rightarrow w_1\rightarrow w_2\rightarrow \cdots \rightarrow w_q$ is the sequence of vertices encountered at the heads of P's edges, i.e., $trace(P) = w_1 w_2 \cdots w_q$. The vertex at which P starts is not part of its trace. A *node listing* (Kennedy,

† It suffices to determine if there is a positive *simple* cycle in F where an edge $t\rightarrow s$ is weighted $w(s) = \sigma(\left[\begin{smallmatrix} \varepsilon \\ \lambda(s) \end{smallmatrix}\right])$. As each sub-

1975) for a set Π of paths is a sequence $L = v_1 v_2 \cdots v_p$ of vertices such that for every path $P \in \Pi$, P's trace is a subsequence of L. That is, there exist $t(1) < t(2) < \cdots < t(q)$ such that $trace(P) = v_{t(1)} v_{t(2)} \cdots v_{t(q)}$. Vertices may occur more than once in L, and indeed must for many instances of Π. Moreover, there can be paths whose trace is a subsequence of L that are not in Π, i.e., L may list a superset of the paths in Π.

Let χ be the set of cycle-free paths from Θ to Φ, and suppose L is a node listing for χ. Without loss of generality one may assume that Θ does not occur in L and that the last vertex in L is Φ. The following algorithm uses L to compute the weight of a maximum-weight path from Θ to Φ. A special vertex, v_0, not in $G_{A,R}$ conveniently handles the boundary case where v_k is first encountered.

$C(v_0) \leftarrow -\infty$
$C(\Theta) \leftarrow 0$
for $k \leftarrow 1$ **to** p **do**
  { $j \leftarrow \max\{i\colon\ i = 0$ or $i \in [1, k-1]$ and $v_i = v_k\}$
    $C(v_k) \leftarrow \max\{\ C(v_j),\ \max\{C(v_i) + \sigma(v_i \rightarrow v_k)\colon\ i \in [j+1, k-1]$ and $v_i \rightarrow v_k$ is an edge$\}$ $\}$
  }
write "Maximum score is" $C(\Phi)$

After the k^{th} iteration of the **for** loop, $C(v_i)$, for all $i \le k$, is the maximum weight of a path from Θ to v_i whose trace is a subsequence of the listing $v_1 v_2 \cdots v_k$. If not zero, j is the last position at which v_k was encountered in L, and it is only necessary to update $C(v_k)$ with respect to those C-values updated between the j^{th} and k^{th} iterations of the loop. These two observations suffice to show that upon completion, $C(\Phi)$ is the maximum weight of a path whose trace is a subsequence of L. Since L lists a superset of χ and it suffices to consider only cycle-free paths, $C(\Phi)$ is the maximum weight of a source-to-sink path in $G_{A,R}$.

Lemma 3 gives a node listing for χ, the set of all cycle-free paths from Θ to Φ in $G_{A,R}$. Let $\theta, s_1, s_2, \cdots, s_n = \phi$ be a topological ordering of F's states relative to the DAG edges. Such an ordering exists because removing the back edges from F makes its graph acyclic. For $i \in [0, M]$, let τ_i be the vertex sequence, $(i, s_1)(i, s_2) \cdots (i, s_n)$.

Lemma 3. The sequence $\tau_0\ (1, \theta)\ \tau_1\ \tau_1\ (2, \theta)\ \tau_2\ \tau_2\ \cdots\ (M, \theta)\ \tau_M\ \tau_M$ is a node listing for χ.

Proof. In the automaton F, any cycle-free path from θ consists solely of DAG edges by Lemma 1.1. Thus: (1) $\tau = s_1 s_2 \cdots s_n$ is a node listing for the set of cycle-free paths from θ. Any cycle-free path from $s_k \ne \theta$ contains at most one back edge by Lemma 1.2. Such a path must consist of a sequence of DAG edges, possibly followed by a back edge to a vertex other than θ and another sequence of DAG edges. Thus: (2) $s_{k+1} s_{k+2} \cdots s_n \tau$ is a node listing for the set of cycle-free paths from $s_k \ne \theta$.

By the construction of $G_{A,R}$, a path from Θ to Φ has a trace that consists of a sequence of vertices in row 0, followed by a sequence of vertices in row 1, and so on, to a final sequence of vertices in row M. Thus a cycle-free path P from Θ to Φ may be partitioned into an initial cycle-free subpath P_0 from Θ to P's last vertex in row 0, followed by a cycle-free subpath P_1 from P's last vertex in row 0 to P's last vertex in row 1, and so on, to a final cycle-free subpath P_M from P's last vertex in row $M-1$ to Φ. To complete the proof it suffices to show that P_0's trace is a subsequence of τ_0, and that P_i's trace is a subsequence of $(i, \theta)\ \tau_i \tau_i$ for $i \in [1, M]$. P_0 is a cycle-free path from θ in the 0^{th} copy of F and so its trace is a subsequence of τ_0 by (1). If the second vertex of P_i is (i, θ) then the remainder is a cycle-free path from θ in the i^{th} copy of F and so P_i's trace is a subsequence of $(i, \theta)\ \tau_i$ by (1). Otherwise, suppose the second vertex of P_i is (i, s_k) where $s_k \ne \theta$. Then the remainder is a cycle-free path from s_k

---

automata S is constructed, the weight $W(S)$ of the maximum weight cycle-free path from θ_S to ϕ_S can be determined with the recurrences: $W(a) = 0$, $W(RS) = W(R) + w(\theta_S) + W(S)$, $W(R|S) = \max\{w(\theta_R) + W(R),\ w(\theta_S) + W(S)\}$, and $W(R^*) = \max\{w(\theta_R) + W(R), 0\}$. A simple cycle consists of a back edge in the automaton for S^* and a cycle-free path from θ_S and ϕ_S. Thus a positive cycle exists if and only if $W(S^*) > 0$ for some sub-expression S^* of R.

in the i^{th} copy of F and so P_i's trace is a subsequence of $(i, s_k)(i, s_{k+1}) \cdots (i, s_n)\ \tau_i$ by (2). In both cases, $(i, \theta)\ \tau_i \tau_i$ is a node listing for P_i. □

Specializing the node listing algorithm above for the specific node list for χ given in Lemma 3 leads to the algorithm in Figure 6. Lines 2 and 3 process the node listing τ_0, where D is the set of DAG edges in F. During the i^{th} iteration of the **for** loop of lines 4 to 13, line 5 processes (i, θ), lines 6 through 10 process the first occurrence of τ_i, and lines 11 through 12 process the second occurrence. In each case the maximum computation of the general node listing algorithm is customized according to the structure of $G_{A,R}$ and the σ term is moved outside the maximum computation wherever possible.

1. $C(0, \theta) \leftarrow 0$
2. **for** $s \in V - \{\theta\}$ in topological order **do**
3. $\quad C(0, s) \leftarrow \max_{t \to s \in D} \{C(0, t)\} + \sigma(\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix})$
4. **for** $i \leftarrow 1$ **to** M **do**
5. $\quad \{\ C(i, \theta) \leftarrow C(i-1, \theta) + \sigma(\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix})$
6. $\quad$ **for** $s \in V - \{\theta\}$ in topological order **do**
7. $\quad\quad \{\ C(i, s) \leftarrow \max_{t \to s \in D} \{C(i, t)\} + \sigma(\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix})$
8. $\quad\quad$ **if** $\lambda(s) \neq \varepsilon$ **then**
9. $\quad\quad\quad C(i, s) \leftarrow \max\{C(i, s), C(i-1, s) + \sigma(\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}), \max_{t \to s \in E} \{C(i-1, t)\} + \sigma(\begin{bmatrix} a_i \\ \lambda(s) \end{bmatrix})\}$
10. $\quad\quad \}$
11. $\quad$ **for** $s \in V - \{\theta\}$ in topological order **do**
12. $\quad\quad C(i, s) \leftarrow \max\{C(i, s), \max_{\substack{t \to s \in E \\ t \neq \theta}} \{C(i, t)\} + \sigma(\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix})\}$
13. $\quad \}$
14. **write** "Similarity score is" $C(M, \phi)$

Figure 6: The approximate match algorithm with column-sum costs.

Since the in-degree of every vertex in F is two or less, the in-degree of every vertex in $G_{A,B}$ is five or less. Thus only $O(1)$ time is spent updating the C-values for each entry in χ. Recalling that F contains $n+1$ vertices it follows that there are $n + M(2n+1)$ entries in χ. Moreover, $n+1 \leq 2N$ by the construction of F. Thus, the total time spent in the column-sum algorithm is $O(MN)$. A maximum-weight path may be found by retaining, for each entry $C(i, s)$, the edge along which its maximum value is attained. This trace-back information requires $O(MN)$ bits. With it one can deliver the sequence in R whose optimal alignment with A is highest scoring, and its optimal alignment with A. However, if just the score of the best alignment between A and a sequence in R is desired, then only $O(N)$ words of memory are required. χ lists vertices row-by-row and the C-values in row i depend only on the C-values in rows i and $i-1$. Thus an $(n+1)$-element vector can hold the C-values for row $i-1$ while the C-values for row i are computed in a second $(n+1)$-element vector. In Section 6, an algorithm variation exploits this observation to actually deliver an approximate match and its alignment with A in linear space.

5. The gap-penalty edit graph and algorithm for regular expressions. Attention is now turned to developing a *gap-penalty edit graph*, $G^{+}_{A,R}$, for comparing sequence $A = a_1 a_2 \cdots a_M$ and regular expression R under the gap-penalty cost model. As before, this edge-labeled graph is designed so that source-to-sink paths model alignments between A and sequences in R, and the maximum-weight path gives the solution. Let $F = < V, E, \lambda, \theta, \phi >$ be the finite automaton F_R for R constructed as in Section 3. As with $H^{+}_{A,B}$ there are three vertices for each integer-state pair (i, s): one of type C(ommon), one of type D(eletion), and one of type I(nsertion). Specifically, the vertices of

$G^+_{A,R}$ are the pairs: $(i, s)_C$ and $(i, s)_I$ for $i \in [0, M]$ and $s \in V$, and $(i, j)_D$ for $i \in [0, M]$ and $s \in V - \{t: \lambda(t) = \varepsilon$ and $t \neq \theta\}$. Distinguish $\Theta = (0, \theta)_C$ as $G^+_{A,R}$'s source vertex, and $\Phi = (M, \phi)_C$ as its sink vertex. The following edges, and only these edges, are in $G^+_{A,R}$.

1. If $i \in [1, M]$ and either $s = \theta$ or $\lambda(s) \neq \varepsilon$, then there is a *deletion* edge $(i-1, s)_D \rightarrow (i, s)_D$ labeled $\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}$.
2. If $i \in [1, M]$ and either $s = \theta$ or $\lambda(s) \neq \varepsilon$, then there is a *deletion initiation* edge $(i-1, s)_C \rightarrow (i, s)_D$ labeled $\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}$.
3. If $i \in [0, M]$ and $t \rightarrow s \in E$, then there is an *insertion* edge $(i, t)_I \rightarrow (i, s)_I$ labeled $\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix}$.
4. If $i \in [0, M]$, $t \rightarrow s \in E$, and $\lambda(s) \neq \varepsilon$, then there is an *insertion initiation* edge $(i, t)_C \rightarrow (i, s)_I$ labeled $\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix}$.
5. If $i \in [0, M]$, $t \rightarrow s \in E$, and $\lambda(s) = \varepsilon$, then there is a *free* edge $(i, t)_C \rightarrow (i, s)_C$ labeled η.
6. If $i \in [1, M]$, $t \rightarrow s \in E$, and $\lambda(s) \neq \varepsilon$, then there is a *substitution* edge $(i-1, t)_C \rightarrow (i, s)_C$ labeled $\begin{bmatrix} a_i \\ \lambda(s) \end{bmatrix}$.
7. If $i \in [0, M]$ and either $s = \theta$ or $\lambda(s) \neq \varepsilon$, then there is a *null* edge $(i, s)_D \rightarrow (i, s)_C$ labeled η.
8. If $i \in [0, M]$ and $s \in V$, then there is a *null* edge $(i, s)_I \rightarrow (i, s)_C$ labeled η.

There are a number of vertices unreachable from Θ in the construction as given, namely, $(i, s)_I$ for all s such that $L_F(s) = \{\varepsilon\}$ and $(i, s)_D$ for $i = 0$. One may remove such vertices and the edges directed out of them if desired.

Let N be the length of R. In a given row of $G^+_{A,R}$ there are no more than $2N$ C-vertices, $2N$ I-vertices, and $N+1$ D-vertices. Thus there are at most $(5N+1)(M+1)$ vertices in $G^+_{A,R}$. With respect to edges, there are at most $4(M+1)N$ of type 3, $4(M+1)N$ of either type 4 or 5, $2(M+1)N$ of type 8, $2MN$ of type 6, and $M(N+1)$ each of types 1, 2, and 7. So $G^+_{A,R}$ contains at most $15MN + 10N + 3M$ edges and its ''size'' is $O(MN)$ regardless of whether one counts vertices or edges.

As with $G_{A,R}$, a path in $G^+_{A,R}$ from Θ to $(i, s)_C$ spells an alignment between A_i and a sequence in $L_F(s)$, and every such alignment is spelled by some path. Lemma 4 formally proves this claim by giving label-preserving homomorphisms, $\vec{\mu}$ and $\overleftarrow{\mu}$, from paths in one graph to paths in the other, and then appealing to Lemma 2.

Lemma 4. There is a path in $G^+_{A,R}$ from Θ to $(i, s)_C$ spelling S if and only if S aligns A_i and $B \in L_F(s)$.

Proof. (=>) For every non-null edge, $e = (i, t)_X \rightarrow (j, s)_Y$, in $G^+_{A,R}$ there is an edge $\vec{\mu}(e) = (i, t) \rightarrow (j, s)$ in $G_{A,R}$ with the same label. Moreover, null edges aren't spelled and occur between vertices with the same (row, state) pair. Thus mapping each non-null edge e of a path in $G^+_{A,R}$ to the edge $\vec{\mu}(e)$ gives a path in $G_{A,R}$ spelling the same alignment. Hence, for any path from Θ to $(i, s)_X$ in $G^+_{A,R}$, there is a synonymous path from Θ to (i, s) in $G_{A,R}$ that by Lemma 2 aligns A_i and $B \in L_F(s)$.

(<=) By Lemma 2 there is a path from Θ to (i, s) in $G_{A,R}$ spelling S, so it suffices to map each edge $e = v \rightarrow w$ labeled π to a path, $\overleftarrow{\mu}(e)$, from v_C to w_C in $G^+_{A,R}$ that also spells π. First, for substitution edges $e = (i-1, t) \rightarrow (i, s)$, $\overleftarrow{\mu}(e) = (i-1, t)_C \rightarrow (i, s)_C$ is identically labeled and exists for exactly the same choices of i, t, and s. Next, there are deletion edges $e = (i-1, s) \rightarrow (i, s)$ labeled $\pi = \begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}$ in $G_{A,R}$ for $i \in [1, M]$ and either $s = \theta$ or $\lambda(s) \neq \varepsilon$. For the same choices of i and s the path $\overleftarrow{\mu}(e) = (i-1, s)_C \rightarrow (i, s)_D \rightarrow (i, s)_C$ exists in $G^+_{A,R}$ and it spells $\pi\,\eta = \pi$. Finally, there are insertion edges $e = (i, t) \rightarrow (i, s)$ labeled $\pi = \begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix}$ in $G_{A,R}$ for $i \in [0, M]$ and $t \rightarrow s \in E$. If $\lambda(s) \neq \varepsilon$, then for the same choices of i, t, and s, the path $\overleftarrow{\mu}(e) = (i, t)_C \rightarrow (i, s)_I \rightarrow (i, s)_C$ exists in $G^+_{A,R}$ and it spells $\pi\,\eta = \pi$. Otherwise, $\lambda(s) = \varepsilon$ and so $\pi = \eta$. In this case, there is a free edge $\overleftarrow{\mu}(e) = (i, t)_C \rightarrow (i, s)_C$ labeled η in $G^+_{A,R}$. □

Weight $G^+_{A,R}$ as follows. Deletion initiation and insertion initiation edges are weighted $\sigma(\pi) - g$ where π is the aligned pair labeling the edge. Weight the remaining edges $\sigma(\pi)$ if $\pi \neq \eta$, and zero otherwise, i.e., choose $\sigma(\eta) = 0$. Under this scheme, the cumulative weight of a path P's edges, *Weight*(P), is the sum of the scores of its

non-null labels minus g times the number of initiation edges. Unfortunately, the number of initiation edges in P may exceed the number of gaps in the alignment S that it spells. In such a case, the weight of P is less than S's gap-penalty score. However, for a select subset of the paths in $G^+_{A,R}$, path weights and alignment scores correspond, as shown in Lemma 5. For a path P, a *deletion anomaly* in P is a subpath of the form $(i-1, s)_D \rightarrow (i-1, s)_C \rightarrow (i, s)_D$, an *insertion anomaly* in P is a subpath of the form $(i, t)_I \rightarrow (i, t)_C \rightarrow (i, s)_I$, and a *free-edge anomaly* in P is a subpath of the form $(i, t)_I \rightarrow (i, t)_C \rightarrow (i, s)_C$. P is *normal* if it has no deletion, insertion, or free-edge anomalies.

Lemma 5. Let P be a normal path from Θ to Φ in $G^+_{A,R}$ spelling S. Then $Weight(P) = Score_{gap}(S)$.

Proof. It suffices to show that the number of initiation edges in P equals the number of gaps in S. To spell an insertion gap of S, P must past though a sequence of I-vertices which are reached by an insertion initiation edge. Thus, each insertion gap of S corresponds to a least one insertion initiation edge of P. A similar statement is true for deletion pairs. Thus the number of gaps in S is not greater than the number of initiation edges of P. This proves that $Weight(P) \leq Score_{gap}(S)$ for *all* paths in $G^+_{A,R}$. The proof is completed by showing that an initiation edge spells the leftmost pair of a gap in S.

First consider a deletion initiation edge, $e = (i-1, s)_C \rightarrow (i, s)_D$, of P. If $(i-1, s)_C = \Theta$ then e's label is certainly the leftmost pair of a gap in S. Otherwise, there is an edge f preceding e and it cannot come from $(i-1, s)_D$ because P is normal. Hence f is either $(i-2, t)_C \rightarrow (i-1, s)_C$ or $(i-1, s)_I \rightarrow (i-1, s)_C$. In the first case, the lower element of f's label is not ε, and so e's label is the leftmost pair in a gap of S. In the second case, f is labeled η and the construction of $G^+_{A,R}$ insures that a trace back along P from $(i-1, s)_I$ encounters a possible empty sequence of insertion edges until an insertion initiation edge is reached. These preceding edges are either labeled with η or an insertion pair, and at least one, the insertion initiation edge, has a non-null label. Thus e's label is the leftmost pair in a gap of S in this case too.

Now, consider an insertion initiation edge, $e = (i, t)_C \rightarrow (i, s)_I$, of P. We may assume that there is an earlier edge of P with non-null label, since otherwise e's label is the first pair of S. By the construction of $G^+_{A,R}$, the edge e may be preceded in P by a sequence of zero or more η-labeled free edges which are then preceded by either (a) a null edge from an I-vertex, (b) a substitution edge, or (c) a null edge from a D-vertex. Case (a) is not possible since P is normal. In case (b), e's label must be the leftmost pair in a gap of S because the substitution edge's upper element is not ε. In case (c), the edge preceding the η-labeled null edge must be a deletion or deletion initiation edge whose label is a deletion pair. Thus, e's label is the leftmost pair in a gap in this case too. □

By Lemma 4 every normal source-to-sink path spells an alignment between A and a sequence in R. Lemma 6 shows that any path may be transformed into a normal path that spells the same alignment. Combined with Lemma 4 this assures that for every alignment between A and a sequence in R there is a normal path spelling the alignment. Thus the approximate regular expression matching problem under the gap-penalty model reduces to computing a maximum-weight normal path from Θ to Φ in $G^+_{A,R}$ as weighted by σ and g. Moreover, Lemma 6 implies that a maximum-weight path must either be normal or transformable into a normal path of equal weight. Thus, it suffices to find a maximum-weight path of $G^+_{A,R}$ because its weight must be that of its alignment's gap-penalty score.

Lemma 6. For every path P from Θ to Φ in $G^+_{A,R}$ spelling S, there is a normal path Q from Θ to Φ spelling S of greater or equal score.

Proof. Suppose P contains an anomaly. We will show how to remove the anomaly (or at least shift it to later in the path) without changing the alignment spelled by P. First, a deletion anomaly $(i-1, s)_D \rightarrow (i-1, s)_C \rightarrow (i, s)_D$ can be replaced by the single edge $(i-1, s)_D \rightarrow (i, s)_D$, thereby removing the anomaly. Similarly, an insertion anomaly $(i, t)_I \rightarrow (i, t)_C \rightarrow (i, s)_I$ can be replaced by the single edge $(i, t)_I \rightarrow (i, s)_I$. Finally, a free-edge anomaly $(i, t)_I \rightarrow (i, t)_C \rightarrow (i, s)_C$ can be replaced by the synonymous subpath $(i, t)_I \rightarrow (i, s)_I \rightarrow (i, s)_C$, thereby either removing the anomaly or creating a new anomaly one edge later in the path. In the later case, the anomaly may be similarly removed, possibly causing a cascade of replacements, until either the end of P is reached or a replacement

is performed that does not create a new anomaly. With regard to weights, simply observe that the first two transforms increase a path's weight by g and the third replacement leaves it's weight unchanged. □

As for the column-sum problem, paths with cycles can be removed from consideration by first checking that there are no cycles of positive weight. It is not hard to show that a cycle of $G^{+}_{A,R}$ consists solely of insertion edges. Thus it suffices to check for positive cycles in a copy of F where each edge $t \rightarrow s$ is weighted $\sigma(\left[\begin{smallmatrix}\varepsilon\\ \lambda(s)\end{smallmatrix}\right])$. This can be done in $O(N)$ time as described in the footnote of Section 4. If there are no positive cycles, then a maximum-weight path from Θ to Φ may be found by examining only those paths that are normal and cycle-free.

To compute the maximum-weight source-to-sink path in $G^{+}_{A,R}$, it suffices to give a node-listing for χ, the set of cycle-free normal paths from Θ to Φ, and then customize the general node-listing algorithm of Section 4. Lemma 7 gives an $O(MN)$ length node-listing for χ in terms of the following definitions. Let $\tau = s_1 s_2 \cdots s_n$ where $s_n = \phi$, be a topological ordering of $V - \{\theta\}$ relative to F's DAG edges. Let $t_1 t_2 \cdots t_m$ be the sequence obtained by deleting from τ all states s for which $\lambda(s) = \varepsilon$. Let δ_i be the node-listing $(i, t_1)_D (i, t_2)_D \cdots (i, t_m)_D$; let τ_i be the node-listing $(i, s_1)_C (i, s_2)_C \cdots (i, s_n)_C$; and let ι_i be the node-listing $(i, s_1)_I (i, s_2)_I \cdots (i, s_n)_I$. Finally, let $\rho_0 = \tau_0\ \iota_0\ \iota_0\ \tau_0$ and let $\rho_i = (i, \theta)_D\ \delta_i\ (i, \theta)_C\ \tau_i\ \tau_i\ \iota_i\ \iota_i\ \tau_i$ for $i \in [1, M]$.

Lemma 7: The sequence $\rho_0\ \rho_1\ \rho_2\ \cdots\ \rho_M$ is a node-listing for χ.

Proof: As in Lemma 3, a cycle-free path P from Θ to Φ may be partitioned into an initial cycle-free subpath P_0 from Θ to P's last vertex in row 0, followed by a cycle-free subpath P_1 from P's last vertex in row 0 to P's last vertex in row 1, and so on, to a final cycle-free subpath P_M from P's last vertex in row $M-1$ to Φ. It suffices to show that P_0's trace is a subsequence of ρ_0, and that P_i's trace is a subsequence of ρ_i for $i \in [1, M]$.

P_0 may begin with a possibly empty, cycle-free sequence of free edges from Θ. By Lemma 1.1 this portion is traced by τ_0. Then P either leaves row 0 or it continues along an insertion initiation edge, followed by a possibly empty, cycle-free sequence of insertion edges, and a final null edge back to a C-vertex. This portion of P_0's trace is a subsequence of $\iota_0\ \iota_0\ \tau_0$ by Lemma 1.2. Since P is normal, it must leave row 0 at this final C-vertex. Thus $trace(P_0)$ is a subsequence of ρ_0.

Now consider P_i for $i \in [1, M]$. P may enter $(i, s)_D$ along a deletion initiation or deletion edge and then exit immediately along a deletion edge. In this case, P_i's trace is a subsequence of $(i, \theta)_D \delta_i$. Otherwise, P must either enter $(i, s)_D$ and then proceed to $(i, s)_C$ along a null edge, or enter $(i, s)_C$ directly along a substitution edge. From there P may then follow a possibly empty, cycle-free sequence of free edges through other C-vertices. This portion of P_i's trace is a subsequence of $(i, \theta)_D\ \delta_i (i, \theta)_C\ \tau_i \tau_i$ by Lemma 1.2. At this point, P may either leave row i or it may continue along an insertion initiation edge, followed by a possibly empty, cycle-free sequence of insertion edges, and a final null edge back to a C-vertex. This portion of P_i's trace is a subsequence of $\iota_i \iota_i \tau_i$, again by Lemma 1.2. Since P is normal it must leave row i at this final C-vertex. Thus $trace(P_i)$ is a subsequence of ρ_i.

□

The gap-penalty approximate match algorithm of Figure 7 is obtained by specializing the general node listing algorithm of Section 4 for the node list of Lemma 7. The values $D(i, s)$, $I(i, s)$, and $C(i, s)$ give the maximum path weights from $(0, \theta)$ to $(i, s)_D$, to $(i, s)_I$, and to $(i, s)_C$, respectively. Each minor loop of the algorithm is annotated on the right with the portion of the node listing to which it corresponds. To simplify matters, unreachable variables such as $D(0, s)$ and $I(i, \theta)$ are set to $-\infty$, a suitably large negative constant.† This value is also assigned to $C(0, s)$ in line 5, $I(0, s)$ in line 8, and $I(i, s)$ in line 25 whenever the corresponding vertex is not reachable from Θ by a path whose trace is a subsequence of the prefix of the node listing thus far processed. The qualified *max* terms in lines 5, 8, 10, 18, 20, 22, 25, and 27 may be over empty sets of values in which case their value is also assumed to be $-\infty$.

† It suffices to choose $-\infty = (M + N)\ \sigma_{\min} - 2g - 1$ where $\sigma_{\min} = \min\{\ \sigma(\pi) : \pi$ a non-null aligned pair $\}$.

An additional refinement to the node listing paradigm is made in lines 12 and 29. According to the paradigm line 29 should be:

$$C(i, s) \leftarrow \max\{C(i, s), I(i, s), \max_{\substack{t \to s \in E \\ \lambda(s) = \varepsilon}} \{C(i, t)\}\}.$$

However, the inner *max* term can be dropped since only normal paths need to be considered. The same is true for line 12. Recall that the node listing traces more than the paths in χ. It also traces non-optimal paths and optimal paths that are not normal or that may contain zero-weight cycles. Thus a naive trace-back procedure may find itself tracing a non-normal path, or, worse, infinitely following a cycle. The ''optimization'' above ensures that paths with free-edge anomalies are not considered even if they are optimal. Since paths with insertion and deletion anomalies are non-optimal, this guarantees that only normal paths will be traced. To avoid getting caught in a loop, it suffices to break ties when evaluating *max* terms in favor of edges from the previous row, then DAG edges within the row, and last of all, back-edges within the row.

1. $C(0, \theta) \leftarrow 0$
2. **for** $s \in V - \{t: \lambda(t) = \varepsilon \text{ and } t \neq \theta\}$ in topological order **do** .....# Unreachable
3. $D(0, s) \leftarrow -\infty$
4. **for** $s \in V - \{\theta\}$ in topological order **do** .....# τ_0
5. $C(0, s) \leftarrow \max_{\substack{t \to s \in D \\ \lambda(s) = \varepsilon}} \{C(0, t)\}$
6. $I(0, \theta) \leftarrow -\infty$ .....# Unreachable
7. **for** $s \in V - \{\theta\}$ in topological order **do** .....# ι_0
8. $I(0, s) \leftarrow \max\{\max_{t \to s \in D} \{I(0, t)\}, \max_{\substack{t \to s \in E \\ \lambda(s) \neq \varepsilon}} \{C(0, t)\} - g\} + \sigma(\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix})$
9. **for** $s \in V - \{\theta\}$ in topological order **do** .....# ι_0
10. $I(0, s) \leftarrow \max\{I(0, s), \max_{\substack{t \to s \in E \\ t \neq \theta}} \{I(0, t)\} + \sigma(\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix})\}$
11. **for** $s \in V - \{\theta\}$ in topological order **do** .....# τ_0
12. $C(0, s) \leftarrow \max\{C(0, s), I(0, s)\}$
13. **for** $i \leftarrow 1$ **to** M **do**
14. { **for** $s \in V - \{t: \lambda(t) = \varepsilon \text{ and } t \neq \theta\}$ in topological order **do** .....# $(i, \theta)_D\ \delta_i$
15. $D(i, s) \leftarrow \max\{C(i-1, s) - g, D(i-1, s)\} + \sigma(\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix})$
16. **for** $s \in V$ in topological order **do** .....# $(i, \theta)_C\ \tau_i$
17. **if** $s = \theta$ or $\lambda(s) \neq \varepsilon$ **then**
18. $C(i, s) \leftarrow \max\{D(i, s), \max_{t \to s \in E} \{C(i-1, t)\} + \sigma(\begin{bmatrix} a_i \\ \lambda(s) \end{bmatrix})\}$
19. **else**
20. $C(i, s) \leftarrow \max_{t \to s \in D} \{C(i, t)\}$
21. **for** $s \in V - \{\theta\}$ in topological order **do** .....# τ_i
22. $C(i, s) \leftarrow \max\{C(i, s), \max_{\substack{t \to s \in E \\ t \neq \theta,\ \lambda(s) = \varepsilon}} \{C(i, t)\}\}$
23. $I(i, \theta) \leftarrow -\infty$ .....# Unreachable
24. **for** $s \in V - \{\theta\}$ in topological order **do** .....# ι_i
25. $I(i, s) \leftarrow \max\{\max_{t \to s \in D} \{I(i, t)\}, \max_{\substack{t \to s \in E \\ \lambda(s) \neq \varepsilon}} \{C(i, t)\} - g\} + \sigma(\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix})$
26. **for** $s \in V - \{\theta\}$ in topological order **do** .....# ι_i
27. $I(i, s) \leftarrow \max\{I(i, s), \max_{\substack{t \to s \in E \\ t \neq \theta}} \{I(i, t)\} + \sigma(\begin{bmatrix} \varepsilon \\ \lambda(s) \end{bmatrix})\}$
28. **for** $s \in V - \{\theta\}$ in topological order **do** .....# τ_i
29. $C(i, s) \leftarrow \max\{C(i, s), I(i, s)\}$
30. }
31. **write** "Similarity score is" $C(M, \phi)$

Figure 7: The approximate match algorithm with gap-penalty costs.

Since the in-degree of every vertex in F is two or less, every *max* term in the algorithm takes $O(1)$ time to compute. Thus lines 1 to 12, and each iteration of the loop of lines 13 to 30 take $O(N)$ time because $|V| \leq 2N$. Hence, the algorithm takes $O(MN)$ time. As for the column-sum algorithm, $O(MN)$ bits of trace-back information suffices to permit the delivery of a normal, cycle-free, and optimal path. Also, four (n+1)-element vectors are all that is needed to deliver just the score of a best alignment between A and a sequence in R: one vector holds the C-values for row $i-1$ while the D-values for row i overwrite the D-values for row $i-1$ in a second vector, and the C- and I-values for row i are computed in the remaining two vectors.

6. Database searches. Thus far it has been required that *all* of A be optimally aligned with a sequence of R. In the context of database queries, the problem becomes to find *substrings* of A, the database, that strongly align with a sequence in R, the pattern. This problem variation is the focus of this section because it raises four new issues. First, one must decide upon the criterion for when R is considered to approximately match a substring of A. Second, the algorithm of Figure 6 must be nontrivially modified to identify such substrings. Third, to be practical, the method of specifying regular expressions should be enhanced as in the UNIX tool *egrep* (Aho, 1980). Finally, since A is much longer than R, it is advantageous to preprocess R in ways that increase the speed of searching A.

6.1. Approximate match criteria. One may define R as approximately matching a substring B of A when the maximum similarity score of B versus R under cost function σ is greater than some threshold τ. An alternative is to use a measure of dissimilarity or difference (Levenshtein, 1966) that is the *minimum* score of an alignment S under a positive cost function δ. Let $\sigma'(\pi) = -\delta(\pi)$ and observe that $Score_{\sigma'}(S) = -Score_{\delta}(S)$. Thus maximizing similarity under σ' will minimize difference under δ. Moreover, the difference between B and R is given by $-C(|B|, \phi)$ when the algorithm of Figure 6 is applied to B and R with cost function σ'. So one may define R as approximately matching B whenever the difference between B and R is less than threshold τ.

For common scoring schemes, particularly difference-based measures, it is better to express the match criteria in length-relative terms. Suppose S is an optimal alignment of B and R. The *match density* of B's alignment with R is $Score_{\sigma}(S) / |B|$. One may define R as approximately matching B whenever their match density is not less than some fraction α. Let $\sigma'(\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right])$ be $\sigma(\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]) - \alpha$ if $a \neq \varepsilon$, and $\sigma(\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right])$ otherwise. Then $Score_{\sigma'}(S) = Score_{\sigma}(S) - \alpha|B|$ and this is non-negative if and only if S's match density is at least α. Thus B matches R with density α or greater if and only if $C(|B|, \phi) \geq 0$ under the cost function σ'. Similarly, one may define R as approximately matching B whenever their *mismatch density*, $Score_{\delta}(S) / |B|$, is α or less under a difference cost function δ (Sellers, 1984).

6.2. Finding matches. Independent of the criterion for a match, a two pass variation of the linear space, cost-only algorithm is required to find the ends, l and r, of matching substrings $B = a_l a_{l+1} \cdots a_r$. Suppose, without loss of generality, that R approximately matches B when the score of their optimal alignment is τ or greater. In the first pass, the algorithm of Figure 6 processes A in a left-to-right scan with line 5 replaced by ''$C(i, \theta) \leftarrow \max\{ 0, C(i-1, \theta) + \sigma(\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]) \}$''. The net effect of this change is that $C(i, \phi) = \max\{ Score(S) : S$ aligns a sequence in R and *a suffix of* $A_i \}$ (Sellers, 1980). Thus, each position r for which $C(r, \phi) \geq \tau$ is the right-end of a matching substring because some suffix B of A_r matches a sequence of R with score τ or greater. For each right-end r, a second pass finds all left-ends l by applying the algorithm of Figure 6 on the reverse of A_r and the regular expression for the reverse of the sequences in R. The finite automaton for this language is easily obtained by simply reversing the direction of every edge of F_R and switching the roles of θ and ϕ. Thus this second phase is a right-to-left scan starting at position r with the ''reverse'' of the automaton for R. Line 5 is *not* modified as in the first pass. For each l for which $C(l, \phi) \geq \tau$ the algorithm outputs (l, r) as the end-points of an approximate match. The algorithm requires only $O(N)$ space since only costs are required. However, there may be a large number of right-ends and all of A_r must be searched for each. Thus a total of $O(SMN)$ worst-case time can be taken where $S \leq M$ is the number of distinct approximate match right-ends in A.

The number of end-point pairs and the time to find them may be reduced by listing only *key* pairs. Consider the plot of $C(i, \phi) - \tau$ produced as the first pass proceeds in increasing order of i. This curve is positive for a range of consecutive indices $[i, i']$ in a neighborhood about the right-end of a strong match. The strongest match has right-end $r \in [i, i']$ that achieves the largest value of $C(r, \phi)$ and is right-most in the event of ties. In the first scan, record only such *key* right-ends. Now consider a plot of $C(i, \phi) - \tau$ as produced by a second scan starting at key end-point r and decreasing in i. The intervals for which this curve is positive contain left-ends with respect to r. Let the *principal* interval be the left-most positive interval whose right-most point is to the right of the key right-end preceding r, or if it doesn't exist, then the right-most positive interval (it must exist). The key left-end in the principal interval is then reported as *the* left-end for r. This choice of left-end guarantees that all reported matches either do not overlap, or if they do, that their overlap is as short as possible. Moreover, the worst-case time spent scanning in the second phase is $O((M+L)N)$ where L is the sum of the lengths of all reported approximate matches. Since L is $O(M)$ in expectation, it takes $O(MN)$ expected-time to report the key pairs. Using the linear space, alignment-delivering variation of the next section, the alignments for each key pair can be delivered in $O(MN \log N)$ expected-time and $O(N)$ space.

6.3. A practical regular expression interface. Software has been written for the first scan of the key pairs search algorithm above. To increase practicality, the software has been designed to accept an extended regular expression syntax similar to the UNIX *egrep* tool that searches ASCII text files for exact matches (Aho, 1980; Miller, 1987). The enhancements to the syntax of Section 3 are as follows. A period, '.', matches every ASCII character except new-line characters, which are matched by '$'. The character '\' escapes the special meaning of meta-characters such as '+', '(', '.', '\', etc. That is, $\backslash a$ matches a for any ASCII character a. The *character class* $[a_1 a_2 \cdots a_k]$ abbreviates the regular expression $a_1 | a_2 | \cdots | a_k$ where each a_i is an ASCII symbol. Ranges of ASCII characters can be further abbreviated as in '[a-z0-9]'. The class [^*list*] matches every character *not* in the list. Within character classes, the special meaning of ']', '-', and '^' can be escaped with a back slash if needed. Finally, the expression $R+$ denotes the expression $RR*$, and $R?$ denotes the expression $R|\ \varepsilon$. With these extensions, there is almost no need to explicitly represent ε, which can be denoted by '[]?'.

The introduction of escapes and a special symbol for new-line are purely syntactic considerations. The new operators and the introduction of character classes require algorithmic changes. The expressions $R+$ and $R?$ are treated by constructing automata F_{R+} and $F_{R?}$ that are similar to F_{R*}. F_{R+} is F_{R*} with the edge from θ_{R*} to ϕ_{R*} removed. $F_{R?}$ is F_{R*} with the back edge from ϕ_R to θ_R removed. Character classes and '.' could also be treated by building the appropriate automaton but it is more efficient to treat them as atomic symbols. Each character class G in R is modeled by a single vertex s in F_R. Let $\sigma_G[a] = \max_{b \in G} \{ \sigma(\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]) \}$ for all $a \in \Sigma \cup \{\varepsilon\}$. Substitution edges into (i, s) have weight $\sigma_G[a_i]$, and insertion edges into (i, s) have weight $\sigma_G[\varepsilon]$. The weights of these edges do not need to be computed for each row, which would require $O(M|G|)$ time, but can be looked up in a precomputed $|\Sigma|+1$-element table for σ_G that takes $O(|\Sigma||G|)$ time to compute. This time savings comes at the expense of requiring $O(|\Sigma| C)$ space where $C \le N$ is the number of character classes in R.

6.4. Speeding up database scans. A typical implementation of the algorithm of Figure 6 reads R, constructs tables modeling the edges and labels of F'_R, and then interprets these tables when performing the maximum computations of lines 3, 7, 9, and 12. Let $n = |V| - 1$. By numbering states in topological order from 0 to n, with θ numbered 0 and ϕ numbered n, lines 2, 6, and 11 become simply "**for** i $\leftarrow$ 1 **to** n **do**". The cost-only version then uses arrays, say $C[0..n]$ and $B[0..n]$, to hold the C-values for rows i and $i-1$, respectively.

As each character a_i is scanned in the first phase of the key pairs algorithm, C and B are updated by lines 5 through 12 and the value of $C[n]-\tau$ is examined. Since M is much larger than N in the case of database queries, every effort should be made to increase the efficiency of this inner loop that depends only on R and the character being scanned. A large factor of efficiency is achieved by compiling code for the inner loop that encodes R in its logic, rather than in a set of tables. That is, given R, a program tailored to the structure of R is produced which

when compiled and run on *A* performs the first phase of the search (Thompson, 1968; Pennello, 1986). Figure 8 gives an example of the C-program produced for the pattern $a(b|c)*$.

Every vector element is mapped to a simple variable, e.g. $C[0]$ to *C00* and $B[3]$ to *B03*. The state-loops of lines 2-3, 6-10, and 11-12 have been completely unrolled and the code for their bodies specialized for each state. For example, state 4 of the automaton for $a(b|c)*$ is labeled $'b'$ and has a single edge from state 3 directed into it. When specialized for this state, lines 7-9 become:

C04 += σ($\left[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}\right]$);
if ((t = B03 + σ($\left[\begin{smallmatrix} a \\ 'b' \end{smallmatrix}\right]$)) > C04) C04 = t;
if ((t = C03 + σ($\left[\begin{smallmatrix} \varepsilon \\ 'b' \end{smallmatrix}\right]$)) > C04) C04 = t;

Variable *a* holds the current scan character, and *t* is available for temporary sums. The scoring scheme σ is realized as a 128 by 128 element array *W* that can be indexed with ASCII characters. The character 0 represents ε, and *initW*( ) is a user-supplied routine that initializes $W[a][b]$ to σ($\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$). *W* is also known to the program producing Figure 8. Thus in the code fragment above, $W[0]['b']$ is known, and its value, -1 in the example, is used directly in the code. Also, *arep* is set to $W[a]$ and *adel* to $arep[0]$ when *a* is first read. If deletion costs are independent of

```
#include <stdio.h>
#include <sys/file.h>

extern int W[128][128];
extern initW();

main(argc,argv) int argc; char *argv[];
{ int ifile, num;
  char buf[BUFSIZ];
  register int t, *arep;
  register int a, i;
  int   C00, C01, C02, C03, C04, C05, C06, C07;
  int   B00, B03;
                                        /* Row 0 */
  C00 = 0;
  C01 = –1;
  C02 = –1;
  C03 = –1;
  C04 = –2;
  C05 = –2;
  C06 = –2;
  C07 = –1;
                                        /* A-loop */
  ifile = open(argv[1],O_RDONLY);
  initW();
  while ((num = read(ifile,buf,BUFSIZ)) > 0)
   for (i = 0; i < num; i++)
    { a = buf[i];
      arep = W[a];
      B00 = C00;                        /* B<-C */
      B03 = C03;
                                        /* Pass 1 */
      C00 = 0;
      C01 –= 1;
      if ((t=B00+arep['a']) > C01) C01 = t;
      if ((t=C00–1) > C01) C01 = t;
      C02 = C01;
      C03 = C02;
      C04 –= 1;
      if ((t=B03+arep['b']) > C04) C04 = t;
      if ((t=C03–1) > C04) C04 = t;
      C05 –= 1;
      if ((t=B03+arep['c']) > C05) C05 = t;
      if ((t=C03–1) > C05) C05 = t;
      C06 = C04;
      if (C05 > C06) C06 = C05;
      C07 = C06;
      if (C02 > C07) C07 = C02;
                                        /* Pass 2 */
      C03 = C06;
      if ((t=C03–1) > C04) C04 = t;
      if ((t=C03–1) > C05) C05 = t;

      /* Test C07-τ (not implemented) */
    }
  exit (0);
}
```

Figure 8: Program for the pattern a(b|c)*

a then *adel* is not present, and its value, -1 in the example, is used directly in the code. Thus the code fragment produced for lines 7-9 and state 4 is further optimized to become:

```
C04 –= 1;
if ((t = B03+arep['b']) > C04) C04 = t;
if ((t = C03–1) > C04) C04 = t;
```

All insertion costs are known to the program-producing program and so it can *a priori* determine the *C*-values for row 0. These values are compiled directly into the code of Figure 8.

By evaluating deletion edges into a vertex first, the only states from row $i-1$ that need to be retained in *B*-variables are those with substitution edges out of them. In the example, this substantially reduces the number of assignments to transfer *C*-values to *B*-values at the start of each iteration, i.e. only *C00* and *C03* need to be saved in *B*-variables. Finally, the only states that must be evaluated in the second sweep of lines 11-12 are those that are within the scope of a back edge, i.e. the head of the back edge is a predecessor of the state and its tail is a proper successor with respect to a topological ordering of states. In the example, only states 3, 4, and 5 are within a back edge, and the second sweep is appropriately reduced.

Three programs were written in the C language for the first scan of the key pairs search algorithm. The first program implements the column-sum algorithm of Figure 6; the second produces a program for column-sum costs as in Figure 8; the third implements the gap-penalty algorithm of Figure 7. Figure 9 gives the scanning speed (time per character of *A*) of each program for several choices of *R* when run on a VAX 11/780 under UNIX 4.3bsd. Handling gap-penalties requires an average of 2.1 times as much processing per character as the column-sum program. The compiled column-sum program is 4.7 times faster on average than its interpretive counterpart. This speedup is for the case of integer weights. Equivalent experiments for real values gives a smaller factor of 2.7 since a greater percentage of time is spent performing arithmetic. When run with a pattern that is just a sequence (e.g. the first pattern in Figure 9), the column-sum programs perform the same computations as Sellers' evolutionary distance algorithm (Sellers, 1980), and so generalize it. On such patterns, the compiled program is 2.3 times faster than a program implementing Sellers' algorithm, further illustrating the power of our method of speedup.

Pattern *R*	Number of States in F'_R	Column-sum (Interpreted)	Column-sum (Compiled)	Gap-penalty (Interpreted)
keyword	8	325	85	669
one..?.?two	15	561	121	1173
alpha|beta|gamma	19	698	159	1448
[0-9]+\.[0-9]*(E(\+|-)?[0-9]+)?	20	726	123	1559

Figure 9: Timing data in micro-seconds per character.

7. Extensions. By generalizing a recursive divide-and-conquer algorithm of Hirschberg (1975) for delivering longest common subsequences, we extend our $O(MN)$-time, $O(N)$-space optimal score algorithm to deliver an optimal alignment and its sequence in *R* in $O(MN \log M)$ time and $O(N + \log M)$ space. It is further shown that only $O(MN)$ time is required when *R* corresponds to a directed network, i.e. when *R* does not contain *. Finally, a cubic time algorithm is developed for the approximate regular expression matching problem under scoring schemes where gaps are penalized according to an increasing but otherwise arbitrary function of their length.

7.1. Optimal alignments in linear space. The central idea to delivering an optimal alignment in $O(N + \log M)$ space is to find the 'midpoint' of an optimal alignment using a 'forward' and a 'reverse' application of the $O(N)$-space cost-only algorithm of Figure 6. Once an optimal midpoint is found, an optimal alignment can be delivered

by recursively determining optimal alignments on both sides of this midpoint. Throughout, we will consider the problem of comparing A and *automaton* F ($= F_R$).

Suppose $M > 1$ and let $i^* = \lfloor M/2 \rfloor$. In a forward phase, the column-sum cost-only algorithm is applied to A_{i^*} and F, resulting in a vector C for which $C(s)$ is the maximum cost of an alignment between A_{i^*} and $L_F(s)$. Let $rev(A)$ denote the reverse of A, i.e. $a_M a_{M-1} \cdots a_1$, and let A_i^T denote the suffix $a_{i+1} a_{i+2} \cdots a_M$ of A. Let $rev(F)$ be the automaton obtained by reversing the direction of every edge in F and reversing the roles of θ and ϕ. Let $L_F^T(s)$ be the set of sequences spelled by paths from s to ϕ in automaton F. In a reverse phase, the cost-only algorithm is applied to $rev(A)_{M-i^*}$ and $rev(F)$, resulting in a vector D for which $D(s)$ is the maximum cost of an alignment between $rev(A)_{M-i^*}$ and $L_{rev(F)}(s)$. But $rev(A)_{M-i^*} = rev(A_{i^*}^T)$ and $L_{rev(F)}(s) = rev(L_F^T(s))$ where $rev(L) = \{ rev(w) : w \in L \}$. Thus $D(s)$ is the maximum cost of an alignment between $A_{i^*}^T$ and $L_F^T(s)$.

Given vectors C and D, the midpoint of an optimal alignment can be found using the following observation. For any optimal alignment of A and F there exists a state $s^* \in V$ such that the alignment is the concatenation of an optimal alignment of A_{i^*} and $L_F(s^*)$ and an optimal alignment of $A_{i^*}^T$ and $L_F^T(s^*)$. Thus the optimal score between A and F is given by $\max_{s \in V} \{ C(s) + D(s) \}$. If the maximum is attained for state s^*, then (i^*, s^*) is an optimal midpoint for the problem. Let $F_{s,t}$ be the automaton obtained by setting θ to s, ϕ to t, and removing the edges and vertices not on a path from s to t. Given an optimal midpoint, an optimal alignment can then be delivered by (i) recursively finding an optimal alignment between A_{i^*} and F_{θ, s^*}, and (ii) recursively finding an optimal alignment between $A_{i^*}^T$ and $F_{s^*, \phi}$.

Figure 10 outlines an $O(N + \log M)$-space alignment algorithm that writes an optimal alignment using the divide-and-conquer principle. The boundary case, $M \leq 1$, is handled by using a trace-back algorithm to deliver an optimal alignment. This requires only $O(N)$ space since $M \leq 1$. Both the trace-back and cost-only algorithms within *diff* must be run on automaton $F_{s,t}$ for arbitrary s and t. A topological ordering of the relevant vertices of such an automaton can be constructed from F in $O(N)$ time and space and then be discarded once the particular step is complete.

```
shared vectors C[1..|V|], D[1..|V|]

procedure DIFF(A, F)
  { diff(A, F, |A|, θ, φ) }

procedure diff(A, F, m, s, t)
  { if m ≤ 1 then
        Find and output the optimal alignment between A and F_{s,t} with a trace-back algorithm.
    else
      { i ← ⌊m/2⌋
        Compute C with cost-only algorithm on A_i and F_{s,t}.
        Compute D with cost-only algorithm on rev(A)_{m-i} and rev(F)_{t,s}.
        Find r ∈ V maximizing max { C(r)+D(r) }.
        Diff(A_i, F, i, s, r)
        Diff(A_i^T, F, m − i, r, t)
      }
  }
```

Figure 10: Skeleton of a $O(N)$-space alignment-delivering algorithm.

The algorithm uses $O(N + \log M)$ space: $O(N)$ for the globally shared vectors, the trace-back algorithm, and the construction of $F_{s,t}$, and $O(\log M)$ for the implicit activation stack needed for no more than $\lceil \log_2 M \rceil + 1$

levels of recursion. Let N be the size of F, let $n \le N$ be the size of the automaton $F_{s,t}$, and let $T(m, n)$ be the time taken by *diff*(A, F, m, s, t). Then for a suitably large constant c, $T(m, n) \le c(mn+N) + T(\lfloor m \rfloor, n_1) + T(\lceil m \rceil, n_2)$ for $n_1, n_2 \le N$ when $m > 1$, and $T(1, n) \le cN$. It then follows that $T(M, N)$ is $O(MN \log M)$ where M is the length of A.

The divide-and-conquer approach also applies to the gap-penalty algorithm of Figure 7. The midpoint division step is more complex but follows the approach of Myers and Miller (1988b) for the case of two sequences. Furthermore, when R corresponds to a directed network (Sankoff and Kruskal, 1983), it follows that $n_1 + n_2 \le n+1$ in the above recurrence. With this additional condition, it follows that $T(M, N)$ is $O(MN)$.

7.2. Length-dependent block indel costs. The column-sum and gap-penalty cost models treated in this paper are scoring models for which the costs of insertion and deletion pairs depend on the unaligned symbol. A number of investigators (Fitch and Smith, 1983; Waterman, 1984; Waterman, Smith, and Beyer, 1976) have proposed scoring models where the cost of a gap, e.g., $\gamma = \left[\begin{smallmatrix} a_1 a_2 & \cdots & a_k \\ \varepsilon\ \varepsilon & & \varepsilon \end{smallmatrix}\right]$, is a function, w_k, of its length k and not the symbols $a_1, a_2, \cdots a_k$ within it. Given a scoring function σ for aligned pairs and a gap-penalty function w_k, the score, $Score_{len}(S)$, of an alignment is $\sum\{\sigma(\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]) : \left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ is in S and $a, b \ne \varepsilon\} - \sum\{w_{|\gamma|} : \gamma$ is a gap of $S\}$. Several different classes of functions w are of algorithmic and biological interest. *Uniform* gap costs, $w_k = ck$ for some c, and *affine* gap costs, $w_k = ck+d$ for some c and d, are properly subsumed by the column-sum and gap-penalty models, respectively. Both can thus be optimized in $O(MN)$ time. For the case where w is *concave*, an $O(MN \log MN)$-time algorithm for optimally aligning two sequences was given by Miller and Myers (1988b). Waterman, Smith and Beyer (1976) gave an $O(MN(M+N))$-time algorithm for optimally aligning sequences when w is *arbitrary*.

In this section we consider the case where w is *monotone-increasing*, i.e. $w_k \ge w_{k-1}$. For most of the treatment, we also assume that w is *sub-additive*, i.e. $w_{m+n} \le w_m + w_n$. To optimize $Score_{len}(S)$ over all alignments S of A and $B \in R$, where R is a regular expression, the appropriate edit graph, $K_{A,R}$, consists of $G_{A,R}$ with some additional edges. Let $W = \{v \in V : v = \theta$ or $\lambda(v) \ne \varepsilon\}$. The added edges are as follows.

1. If $i \in [2, M]$, $k \in [2, i]$, and $s \in W$, then there is a *block deletion* edge $(i-k, s) \to (i, s)$ labeled $\left[\begin{smallmatrix} a_{i-k+1} a_{i-k+2} & \cdots & a_i \\ \varepsilon & & \end{smallmatrix}\right]$.
2. If $i \in [0, M]$, $t \in W$, $s \in V$, and there is a path from t to s in F'_R, then there is a *block insertion* edge $(i, t) \to (i, s)$ labeled $\left[\begin{smallmatrix} \varepsilon \\ B \end{smallmatrix}\right]$ where B is a shortest sequence spelled by some path from t to s in F'_R.

$K_{A,R}$ is weighted as follows. Insertion and deletion edges labeled γ are weighted $-w_{|\gamma|}$. Substitution edges labeled π are weighted $\sigma(\pi)$ and η-labeled edges are weighted 0.

As for $G_{A,R}$, alignments between A and R are modeled by paths from Θ to Φ in $K_{A,R}$. Moreover, two consecutive insertion edges in a path can be replaced with a single edge without decreasing the path's total weight since w is increasing and sub-additive. The same is true for consecutive deletion edges because w is sub-additive. Thus paths corresponding to optimal alignments can be assumed to be *simple*, i.e. to not contain consecutive deletion or consecutive insertion edges.

To compute an optimal alignment it suffices to give a node listing for the class of simple paths from Θ to Φ in $K_{A,R}$. Let $s_1, s_2, \cdots, s_n$ be any ordering of $V - \{\theta\}$, and let $t_1, t_2, \cdots, t_m$ be any ordering of $W = \{v \in V : v = \theta$ or $\lambda(v) \ne \varepsilon\}$. Then $\Pi = \iota_0 \tau_1 \iota_1 \tau_2 \iota_2 \cdots \tau_M \iota_M$ is a node listing for the set of simple paths from Θ to Φ, where $\iota_i = (i, s_0)(i, s_1) \cdots (i, s_n)$, and $\tau_i = (i, t_1)(i, t_2) \cdots (i, t_m)$. This follows since the vertices on a simple path that lie in row i, if they exist, consist of (i, t) for some $t \in W$, possibly followed by some (i, s) reached from (i, t) by a single insertion edge.

Prior to the node listing algorithm proper, the cost of an insertion edge from (i, t) to (i, s) must be computed. Let $Short(t, s)$ be the length of the shortest sequence spelled on a path from t to s in F'_R, if it exists. $Short(t, s)$ for a fixed t can be computed in $O(N \log N)$ time using the 0/1-weight version of Dijkstra's single-source shortest paths algorithm (Aho, Hopcroft, and Ullman, 1983), Let $\omega(t, s)$ be ∞, a suitably large constant, if $Short(t, s)$ does not exist, 0 if $Short(t, s) = 0$, and $w_{Short(t, s)}$ otherwise. An $N \times N$ table of $\omega(t, s)$ for all t and s, can be built with N application of Dijkstra's algorithm in a total of $O(N^2 \log N)$ time and $O(N^2)$ space.

Specializing the node listing algorithm for the node listing Π leads to the algorithm of Figure 11. Line 3 employs the observation that a vertex $(0, s)$, with $s \neq \theta$, that lies on a simple path from θ must be reached from θ by a single edge of cost $-\omega(\theta, s)$. Lines 5-6 leave $C(i, s)$ equal to the maximum cost of a simple path from Θ to (i, s) that reaches (i, s) by an edge from an earlier row. Lines 7-11 update $C(i, s)$ to account for the sole possible insertion edge in row i.

Step 0 of the algorithm of Figure 11 takes $O(N^2 \log N)$ time and $O(N^2)$ space. The algorithm proper is readily seen to require $O(MN(M+N))$ time and $O(MN)$ space since all C-values must be retained, even in the cost-only case. Thus, Figure 11 gives an $O(MN(M+N)+N^2 \log N)$-time, $O(N(M+N))$-space algorithm for comparing A to R under a monotone-increasing and sub-additive gap-penalty scoring scheme σ, w.

If w is not sub-additive, then one must constrain the computation to consider only simple paths in $K_{A, R}$, for the weight of these paths correspond to the score of their alignments, while those of non-simple paths do not. This may be accomplished by constructing a graph $K^+_{A, R}$ that models only the simple paths of $K_{A, R}$. This graph is obtained by a three-fold node splitting similar to the ones used to construct $G^+_{A, R}$ and $H^+_{A, R}$. The node listing algorithm that results from $K^+_{A, R}$ is only a constant factor slower than that of Figure 11.

```
0.  ''Precompute ω''
1.  C(0, θ) ← 0
2.  for s ∈ V − {θ} do
3.      C(0, s) ← −ω(θ, s)
4.  for i ← 1 to M do
5.    {  for s ∈ W do
6.           C(i, s) ← max{ max_{k∈[1, i]} {C(i−k, s) − w_k}, max_{t→s ∈ E} {C(i−1, t) + σ(⌈a_i / λ(s)⌉)} }
7.       for s ∈ V − {θ} do
8.           if s ∈ W then
9.               C(i, s) ← max{C(i, s), max_{t∈W} {C(i, t) − ω(t, s)} }
10.          else
11.              C(i, s) ← max_{t∈W} {C(i, t) − ω(t, s)}
12.   }
13. write "Similarity score is" C(M, ϕ)
```

Figure 11: The approximate match algorithm with arbitrary gap scores.

7.3. Open problems. A gap-penalty function w is *concave* if and only if $w_k - w_{k-1} \geq w_{k+1} - w_k$ for all $k > 1$. For the case of concave gap-penalties, Miller and Myers (1988b) reduced the time complexity from $O(MN(M+N))$ to $O(MN \log MN)$ time when comparing two sequences. The question as to whether there is a sub-cubic algorithm for approximate regular expression matching under concave gap-penalties is open. Also, we conjecture that there is a node listing based, $O(MN)$-time algorithm for comparing regular expressions R and S, i.e. find the score of the best alignment between a sequence in R and a sequence in S.

LITERATURE

Abarbanel, R. M., P. R. Wieneke, E. Mansfield, D. A. Jaffe and D. L. Brutlag. 1984. ''Rapid searches for complex patterns in biological molecules.'' *Nucleic Acids Research* **12**(1), 263-280.

Aho, A. 1980. ''Pattern matching in strings.'' *Formal Language Theory*, (R. Book, ed.). Academic Press.

Aho, A. V., J. E. Hopcroft and J. D. Ullman. 1983. *Data Structures and Algorithms*, Addison-Wesley, 203-208.

Cohen, F. E., R. M. Abarbanel, I. D. Kuntz and R. J. Fletterick. 1986. ''Turn prediction in proteins using a pattern-matching approach.'' *Biochemistry* **25**, 266-275.

Fitch, W. M. and T. F. Smith. 1983. ''Optimal sequence alignments.'' *Proc. Natl. Acad. Sci. USA* **80**, 1382-1386.

Gotoh, O. 1982. ''An improved algorithm for matching biological sequences.'' *J. Molec. Biol.* **162**, 705-708.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs*. North-Holland.

Hecht, M. S. and J. D. Ullman. 1975. ''A simple algorithm for global data flow analysis programs.'' *SIAM J. Computing* **4**(4), 519-532.

Hopcroft, J. E. and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

Kennedy, K. 1975. ''Node listing techniques applied to data flow analysis.'' *Proc. 2nd ACM Conference on Principles of Programming Languages*, 10-21.

Levenshtein, V. I. 1966. ''Binary codes capable of correcting deletions, insertions, and reversals.'' *Cybernetics and Control Theory* **10**(8), 707-710.

Miller, W. 1987. *A Software Tools Sampler*. Prentice-Hall.

Miller, W. and E. W. Myers. 1988a. ''A simple row-replacement method.'' To appear in *Software—Practice and Experience*.

Miller, W. and E. W. Myers. 1988b. ''Sequence comparison with concave weighting functions.'' To appear in *Bull. Math. Biol.*

Myers, E. W. and W. Miller. 1988a. ''Row replacement algorithms for screen editors.'' To appear in *ACM Trans. Prog. Lang. and Systems.*

Myers, E. W. and W. Miller. 1988b. ''Optimal alignments in linear space.'' *CABIOS* **4**(1), 11-17.

Pennello, T. J. 1986. ''Very fast LR parsing.'' *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction. ACM SIGPLAN Notices* **21**(7), 145-150.

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

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

Sellers, P. H. 1984. ''Pattern recognition in genetic sequences by mismatch density.'' *Bull. Math. Biol.* **64**(4), 501-514.

Thompson, K. 1968. ''Regular expression search algorithm.'' *Comm. ACM* **11**(6), 419-422.

Wagner, R. A. 1974. ''Order-*n* correction of regular languages.'' *Comm. ACM* **17**(5), 265-268.

Wagner, R. A. and J. I. Seiferas. 1978. ''Correcting counter-automaton-recognizable languages.'' *SIAM J. Computing* **7**(3), 357-375.

Waterman, M. S. 1984. ''General methods for sequence comparison.'' *Bull. Math. Biol.* **46**(4), 473-500.

Waterman, M. S., T. F. Smith and W. A. Beyer. 1976. ''Some biological sequence metrics.'' *Adv. in Mathematics* **20**, 367-387.