# Approximate Regular Expression Pattern Matching with Concave Gap Penalties*

*James R. Knight*[1]
*Eugene W. Myers*[1]

TR 92-12

Revised April 26, 1996

**Abstract**

Given a sequence $A$ of length $M$ and a regular expression $R$ of length $P$, an approximate regular expression pattern matching algorithm computes the score of the optimal alignment between $A$ and one of the sequences $B$ exactly matched by $R$. An alignment between sequences $A = a_1 a_2 \dots a_M$ and $B = b_1 b_2 \dots b_N$ is a list of ordered pairs, $<(i_1, j_1), (i_2, j_2), \dots (i_t, j_t)>$ such that $i_k < i_{k+1}$ and $j_k < j_{k+1}$. In this case, the alignment *aligns* symbols $a_{i_k}$ and $b_{j_k}$, and leaves blocks of unaligned symbols, or *gaps*, between them. A scoring scheme $S$ associates costs for each aligned symbol pair and each gap. The alignment's score is the sum of the associated costs, and an optimal alignment is one of minimal score. There are a variety of schemes for scoring alignments. In a concave gap-penalty scoring scheme $S = \{\delta, w\}$, a function $\delta(a, b)$ gives the score of each aligned pair of symbols $a$ and $b$, and a *concave* function $w(k)$ gives the score of a gap of length $k$. A function $w$ is concave if and only if it has the property that for all $k > 1$, $w(k+1) - w(k) \leq w(k) - w(k-1)$. In this paper we present an $O(MP(\log M + \log^2 P))$ algorithm for approximate regular expression matching for an arbitrary $\delta$ and any concave $w$.

[1]Department of Computer Science
University of Arizona
Tucson, AZ 85721
{jrknight,gene}@cs.arizona.edu

**Keywords** Regular Expression, Concave Gaps, Approximate Pattern Matching

*This work was supported in part by the National Institute of Health under Grant R01 LM04960.

Approximate Regular Expression Pattern Matching
with Concave Gap Penalties

# 1 Introduction

The problem of approximately matching a regular expression with concave gap penalties falls into a family of *approximate pattern matching* problems that compute the score of an *optimal alignment* between a given query sequence and one of the sequences specified by the pattern. An alignment is simply a pairing of symbols between two sequences such that the lines of the induced *trace* do not cross, as shown in Figure 1. Alignments are evaluated with a scoring scheme $S$ that gives scores for each *aligned pair* and each contiguous block of unaligned symbols, or *gap*. The score of an alignment is the sum of the scores $S$ assigns to each aligned pair and gap, and an optimal alignment is one of minimal score. The input for an approximate pattern matching problem consists of a sequence $A$, a pattern $R$, and an alignment scoring scheme $S$. The problem is to determine the optimal scoring alignment between $A$ and one of the sequences $B$ exactly matching $R$, where alignments are scored using scheme $S$.

More formally, an alignment between sequence $A = a_1 a_2 \ldots a_M$ and sequence $B = b_1 b_2 \ldots b_N$, over alphabet $\Sigma$, is a list of ordered pairs of indices $<(i_1, j_1), (i_2, j_2), \ldots (i_t, j_t)>$, called a *trace*, such that (1) $i_k \in [1, M]$, (2) $j_k \in [1, N]$, and (3) $i_k < i_{k+1}$ and $j_k < j_{k+1}$. Each pair of symbols $a_{i_k}$ and $b_{j_k}$ is said to be aligned. A consecutive block of unaligned symbols in $A$ or $B$, $a_{i_k+1} a_{i_k+2} \ldots a_{i_{k+1}-1}$ where $i_{k+1} > i_k + 1$, is termed a gap of length $i_{k+1} - i_k - 1$. An alignment, its usual column-oriented display, and several gaps are illustrated in Figure 1. Under a scoring scheme $S = \{pair, gap\}$ with functions $pair$ and $gap$ scoring the symbol pairs and gaps, the score of an optimal alignment between $A$ and $B$, or SEQ($A$, $B$, $\{pair, gap\}$), equals $\min\{\sum_{k=1}^{t} pair(a_{i_k}, b_{j_k}) + \sum_{k=0}^{t} gap(a_{i_k+1} a_{i_k+2} \ldots a_{i_{k+1}-1}) + \sum_{k=0}^{t} gap(b_{j_k+1} b_{j_k+2} \ldots b_{j_{k+1}-1}) \mid <(i_1, j_1), (i_2, j_2), \ldots (i_t, j_t)>$ is a valid trace $\}$, where for simplicity we assume $i_0 = j_0 = 0$, $i_{t+1} = M+1$, $j_{t+1} = N+1$, and $gap(\varepsilon) = 0$. The optimal alignment score between $A$ and a regular expression $R$ is RE($A$, $R$, $\{pair, gap\}$) = $\min\{$ SEQ($A$, $B$, $\{pair, gap\}$) $\mid B \in L(R)\}$, where $L(R)$ is the language defined by $R$.

In this paper we consider two scoring schemes, *symbol-based* and *concave gap penalty*. Both scoring schemes use an arbitrary function $\delta(a, b)$, for $a, b \in \Sigma$, to score the aligned pairs. The difference is in the scoring of gaps. In a symbol-based scheme, $\delta$ is extended to be defined over an additional symbol $\varepsilon$ not in $\Sigma$, and the score of an unaligned symbol $a$ is given by $\delta(a, \varepsilon)$. The score of a gap $a_{i_k+1} a_{i_k+2} \ldots a_{i_{k+1}-1}$ is the sum of the scores of the individual unaligned symbols or $\sum_{p=i_k+1}^{i_{k+1}-1} \delta(a_p, \varepsilon)$. The cost of gaps in $B$ is defined symmetrically. Thus, with symbol-based scores, the scoring scheme $S$ is now denoted as $\{\delta\}$.

The concave gap penalty scheme is one of a number of gap-cost models where the cost of a gap is solely a function of its length (and thus symbol independent). In such a scheme, an additional function $w(k)$ gives the cost of a gap of length $k$. For example, the simple linear gap penalty model defines $w(k) = \alpha\, k$, for some constant $\alpha$, and is just a special case of the symbol dependent model where $\delta(a, \varepsilon) = \delta(\varepsilon, b) = \alpha$ for all $a, b$. In a concave gap penalty scheme, the function $w$ must be *concave* in the sense that its forward differences are non-increasing, or $\Delta w(1) \geq \Delta w(2) \geq \Delta w(3) \geq \ldots$ where $\Delta w(k) \equiv w(k+1) - w(k)$. One example of such a function is the logarithmic function $w(k) = \alpha + \beta \log k$, for constants $\alpha, \beta > 0$.

The problem considered in this paper is a generalization of several earlier results. The traditional sequence comparison problem, SEQ($A$, $B$, $\{\delta\}$), finds the optimal alignment between $A$ and $B$ under symbol-dependent scoring scheme $S = \{\delta\}$. Several authors [15, 16, 18] independently discovered an $O(MN)$ al-

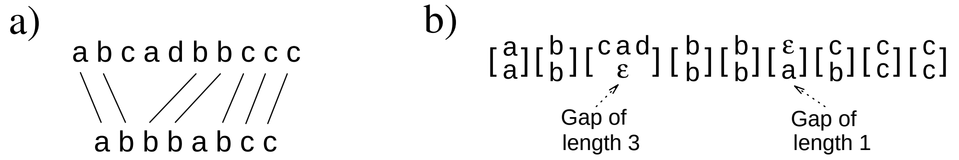



**Figure 1** An alignment a) as a set of trace lines and b) in a column-oriented display.

gorithm for this problem, where $M$ and $N$ are the lengths of $A$ and $B$. In 1984, Waterman [19] generalized this classic problem by considering concave gap penalties, or equivalently the problem SEQ($A$, $B$, $\{\delta, w\}$) where $w$ is concave. A few years a later, a number of authors arrived at an $O(MN(\log M + \log N))$ algorithm [8, 12, 5] using the concept of a *minimum envelope*. In an orthogonal direction of generality, Myers and Miller [14] considered the problem of approximate matching of regular expressions under symbol-dependent scoring schemes, RE($A$, $R$, $\{\delta\}$). By observing that an automaton for $R$ is a reducible graph, they devised a two-sweep node-listing algorithm requiring $O(MP)$ time, where $P$ is the size of $R$. Note that RE($A$, $R$, $\{\delta\}$) generalizes SEQ($A$, $B$, $\{\delta\}$) as a sequence is just a special case of a regular expression.

This paper presents an $O(MP(\log M + \log^2 P))$ algorithm for RE($A$, $R$, $\{\delta, w\}$), the problem of approximately matching $A$ to regular expression $R$ under a concave gap penalty scheme $S = \{\delta, w\}$. The best previous algorithm [14] required $O(MP(M + P))$ or cubic time. Our sub-cubic result builds on the earlier results above by combining the minimum envelope and two-sweep node-listing ideas. However, the extension is not straightforward, requiring the use of *persistent* data structures [13] and collections of envelopes, some of which are organized as stacks.

The paper is organized as follows. Section 2 reviews aspects of algorithms for previous problems concentrating on the concepts of an *alignment graph* and *dynamic programming recurrence*. Section 3 introduces the concept of a minimum envelope in the form needed for our result, along with a brief digression to show how it generalizes [12] and why the regular expression algorithm requires this generalization. Finally, Section 4 presents the algorithm solving RE($A$, $R$, $\{\delta, w\}$) in sub-cubic time.

# 2 Preliminaries

All the problems discussed in the introduction can be recast as problems of finding the cost of a shortest source-to-sink path in an *alignment graph* constructed from the sequence/pattern input to the problem. The reduction is such that each edge corresponds to a gap or aligned pair and is weighted according to the cost of that item. The correctness and inductive nature of the construction follows from the feature that every path between two vertices models an alignment between corresponding substrings/subpatterns of the inputs. From these graphs, *dynamic programming recurrences* for computing the shortest path costs from the source to each vertex are easily derived. In all cases we seek the shortest path cost to a designated sink since every source-to-sink path models a complete alignment between the two inputs.

## 2.1 Sequence vs. Sequence Comparison

For SEQ($A$, $B$, $\{\delta\}$), comparing two sequences under a symbol-based scoring scheme, the alignment graph for $A$ versus $B$ consists of a collection of vertices, $(i, j)$ for $i \in [0, M]$ and $j \in [0, N]$, arranged in an $M+1$ by $N+1$ grid or matrix as illustrated in Figure 2. For vertex $(i, j)$, there are up to three edges directed out of it: (1) a *deletion* edge to $(i + 1, j)$ (iff $i < M$), (2) an *insertion* edge to $(i, j + 1)$ (iff $j < N$), and (3) a *substitution* edge to $(i+1, j+1)$ (iff $i < M$ and $j < N$). In the resulting graph, all paths from *source* vertex $(0, 0)$ to *sink* vertex $(M, N)$ model the set of all possible alignments between $A$ and $B$ with the following simple interpretation: (1) a deletion edge to $(i, j)$ models leaving $a_i$ unaligned and has weight $\delta(a_i, \varepsilon)$, (2)

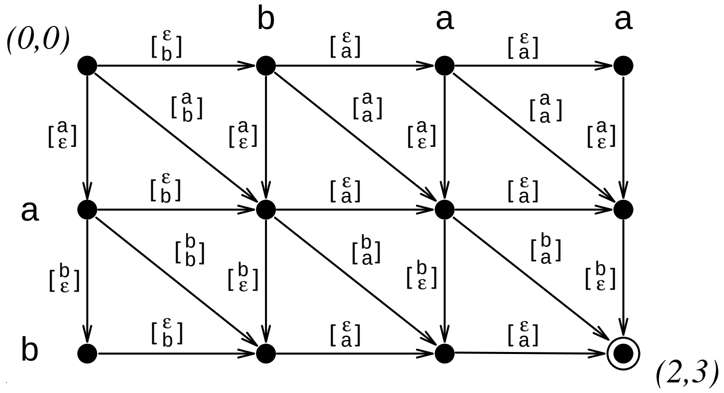



**Figure 2** The sequence vs. sequence alignment graph for $A = ab$ and $B = baa$.

an insertion edge to $(i, j)$ models leaving $b_j$ unaligned and has weight $\delta(\varepsilon, b_j)$, and (3) a substitution edge to $(i, j)$ models aligning $a_i$ and $b_j$ and has weight $\delta(a_i, b_j)$. The terminology for edge types stems from an equivalent sequence comparison model where one seeks a minimum cost series of deletion, insertion, and substitution operations that edit $A$ into $B$. It is convenient here only in that the terms distinguish the cases where symbols of $A$ are left unaligned from those where symbols of $B$ are left unaligned. A simple induction shows that paths between vertices $(i, j)$ and $(k, h)$ (where $i < k$ and $j < h$) are in one-to-one correspondence with alignments between $a_{i+1}a_{i+2} \ldots a_k$ and $b_{j+1}b_{j+2} \ldots b_h$, and their costs coincide. It thus follows that finding the optimal alignment between $A$ and $B$ is equivalent to finding a least cost path between the source and sink vertices.

The dynamic programming principle of optimality holds here: the cost of the shortest path to $(i, j)$ is the best of the costs of (a) the best path to $(i-1, j)$ followed by the deletion of $a_i$, (b) the best path to $(i, j-1)$ followed by the insertion of $b_j$, or (c) the best path to $(i-1, j-1)$ followed by the substitution of $a_i$ for $b_j$. This statement is formally embodied in the fundamental recurrence:

$$(1) \quad C_{i,j} = \min\{\ C_{i-1,j-1} + \delta(a_i, b_j),\ \ C_{i-1,j} + \delta(a_i, \varepsilon),\ \ C_{i,j-1} + \delta(\varepsilon, b_j)\ \}$$

Because the alignment graph is acyclic, the recurrence can be used to compute the shortest cost path to each vertex in any topological ordering of the vertices, e.g. row- or column-major order of the vertex matrix. Thus the desired value, $C_{M,N}$, can be computed in $O(MN)$ time.

## 2.2 Sequence vs. Sequence Comparison with Gap Penalties

For a sequence comparison problem under a gap penalty scoring model, SEQ($A$, $B$, $\{\delta, w\}$), such as the case where $w$ is concave, the alignment graph must be augmented with insertion and deletion edges that model multi-symbol gaps, since their cost is not necessarily additive in the symbols of the gap, i.e. $w(k) \neq k\, w(1)$. From a vertex $(i, j)$, there are now $M - i$ deletion edges to vertices $(i+1, j), (i+2, j), \ldots (M, j)$, where an edge from $(i, j)$ to $(k, j)$ models the gap that leaves $a_{i+1}a_{i+2} \ldots a_k$ unaligned and has cost $w(k-i)$. Similarly, there are $N - j$ insertion edges to vertices $(i, j+1), (i, j+2), \ldots (i, N)$, where an edge from $(i, j)$ to $(i, k)$ models the gap that leaves $b_{j+1}b_{j+2} \ldots b_k$ unaligned and has cost $w(k-j)$. The inductive invariant between alignments and paths still holds, but now the graph has a cubic number of edges. The cost of each incoming edge, plus the cost of the best path to its tail, must now be considered in computing the cost of the shortest path to $(i, j)$.

$$(2) \quad C_{i,j} = \min\{\ C_{i-1,j-1} + \delta(a_i, b_j),\ \ \min_{0 \leq k < i}\{C_{k,j} + w(i-k)\},\ \ \min_{0 \leq k < j}\{C_{i,k} + w(j-k)\}\ \}$$

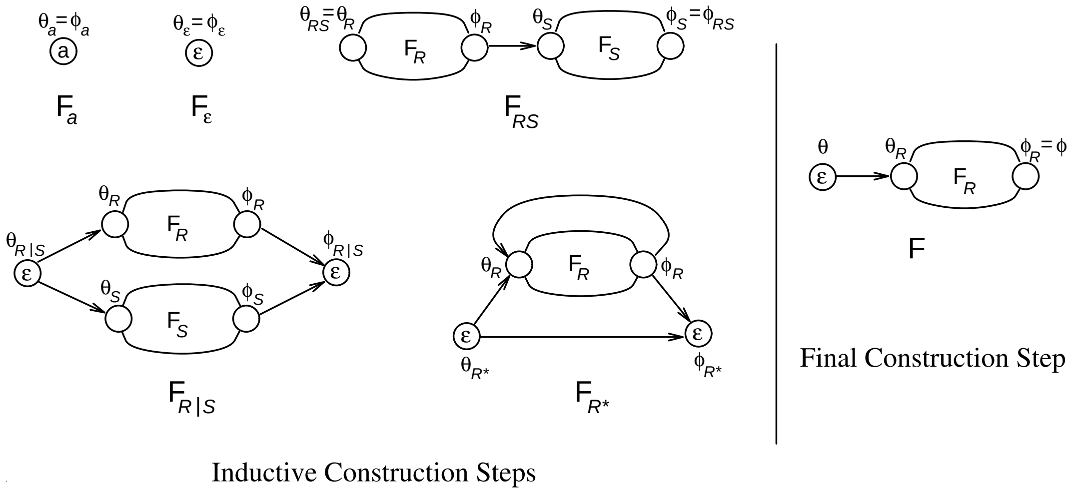



**Figure 3** Constructing the NFA $F$ for a regular expression $R$.

The alignment graph is still acyclic, so applying the recurrence to the vertices in any topological order computes the correct shortest path cost to $(M, N)$. However each application of the recurrence requires $O(M + N)$ time, yielding a $O(MN(M + N))$ algorithm. In the next section on minimum envelopes, it will be revealed how this complexity can be reduced in the case where $w$ is concave.

### 2.3 Sequence vs. Regular Expression Comparison

We now turn our attention to problems that involve generalizing $B$ to a regular expression $R$, such as the problem RE($A$, $R$, $\{\delta\}$) treated in [14]. The discussion here is basically a summary of the results in that paper. Recall that a regular expression over alphabet $\Sigma$ is any expression built from symbols in $\Sigma \cup \{\varepsilon\}$ using the operations of concatenation (juxtaposition), alternation ($|$), and Kleene closure (*). The symbol $\varepsilon$ matches the empty string. For example, $a\ (b \mid \varepsilon) \mid c\, b^*$ denotes the set $\{ab, a, c, cb, cbb, \ldots\}$.

Regular expressions are convenient for the textual specification of *regular languages*, but the graph-theoretic finite automaton is better suited to the purpose of constructing an alignment graph. There are several different such automaton models, see [9] for more details. We use the nondeterministic, state-labeled finite automaton model, hereafter referred to as an NFA, employed by Myers and Miller [14] for the symbol-based regular expression matching problem. Formally, an NFA $F = <V, E, \lambda, \theta, \phi>$ consists of: (1) a set $V$ of vertices, called *states*; (2) a set $E$ of directed edges between states; (3) a function $\lambda$ assigning a "label", $\lambda_s \in \Sigma \cup \{\varepsilon\}$, to each state $s$; (4) a designated "source" state $\theta$; and (5) a designated "sink" state $\phi$. Intuitively, $F$ is a vertex-labeled directed graph with distinguished source and sink vertices. A directed path through $F$ *spells* the sequence obtained by concatenating the non-$\varepsilon$ state labels along the path. $L_F(s)$, the *language accepted at* $s \in V$, is the set of sequences spelled on all paths from $\theta$ to $s$. The *language accepted by* $F$ is $L_F(\phi)$.

Any regular expression $R$ can be converted into an equivalent finite automaton $F$ with the inductive construction depicted in Figure 3. For example, the figure shows that $F_{RS}$ is obtained by constructing $F_R$ and $F_S$, adding an edge from $\phi_R$ to $\theta_S$, and designating $\theta_R$ and $\phi_S$ as its source and sink states. After inductively constructing $F_R$, an $\varepsilon$-labeled start state is added as shown in the figure to arrive at $F$. This last step guarantees that the word spelled by a path is the sequence of symbols *at the head of each edge*, and is essential for the proper construction of the forthcoming alignment graph.

A straightforward induction shows that automata constructed for regular expressions by the above process have the following properties: (1) the in-degree of $\theta$ is 0; (2) the out-degree of $\phi$ is 0; (3) every state

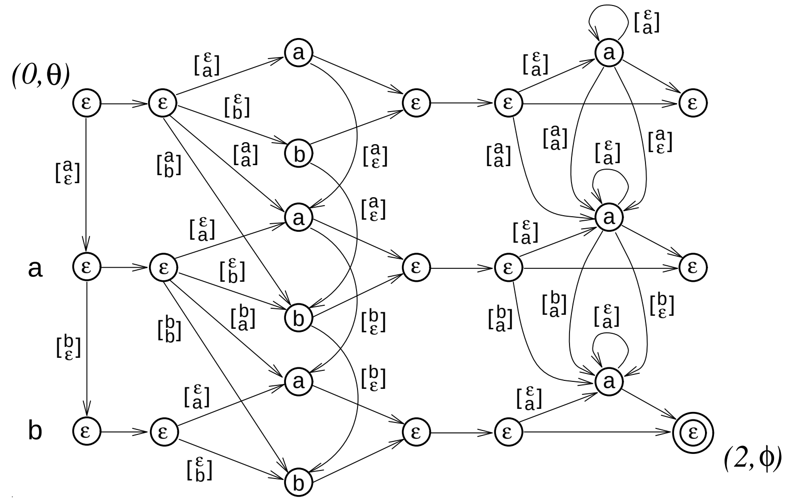



**Figure 4** The regular expression alignment graph for A=ab and P=(a|b)a*.

has an in-degree and an out-degree of 2 or less; and (4) $|V| \leq 2\,|R|$, i.e. the number of states in $F$ is less than or equal to twice $R$'s length. In addition, the structure of cycles in the graph $<V, E>$ of $F$ has a special property. Term those edges introduced from $\phi_R$ to $\theta_R$ in the diagram of $F_{R^*}$ as *back edges*, and term the rest *DAG edges*. Note that the graph restricted to the set of DAG edges is acyclic. Moreover, it can be shown that any cycle-free path in $F$ has at most one back edge. Graphs with this property are commonly referred to as being *reducible* [2] or as having a *loop connectedness parameter* of 1 [7]. In summary, the key observations are that for any regular expression $R$ there is an NFA whose graph is reducible and whose size, measured in either vertices or edges, is linear in the length of $R$.

For the problem RE($A$, $R$, $\{\delta\}$), the alignment graph for $A$ versus $R$ consists of $M+1$ copies of $F$, as illustrated in Figure 4. Formally, the vertices are the pairs $(i, s)$ where $i \in [0, M]$ and $s \in V$. For every vertex $(i, s)$ there are up to five edges directed into it. (1) If $i > 0$, then there is a deletion edge from $(i-1, s)$ that models leaving $a_i$ unaligned. (2) If $s \neq \theta$, then for each state $t$ such that $t \to s$ is an edge in $F$, there is insertion edge from $(i, t)$ that models leaving $\lambda_s$ unaligned (in whatever word of $R$ that is being spelled). (3) If $i > 0$ and $s \neq \theta$, then for each state $t$ such that $t \to s$, there is a substitution edge from $(i-1, t)$ that models aligning $a_i$ with $\lambda_s$. Note that by the construction of $F$, there are at most two insertion and two substitution edges out of each vertex, and $O(MP)$ vertices and edges in the graph.

Unlike the case of sequence comparison graphs, there can be many paths modeling a given alignment in this graph due to the fact that when $\lambda_s = \varepsilon$, insertion edges to $s$ model leaving $\varepsilon$ unaligned and substitution edges to $s$ model aligning $a_i$ with $\varepsilon$. Such insertion edges insert nothing and thus are simply ignored. The substitution edges are equivalent in effect to deletion edges. Regardless of this redundancy, it is still true that every path from $(i, t)$ to $(j, s)$ models an alignment between $a_{i+1}a_{i+2}\ldots a_j$ and the word spelled on the heads of the edges in the path from $t$ to $s$ in $F$ that is the "projection" of the alignment graph path. Moreover, every possible alignment is modeled by at least one path in the graph, and as long as null insertion edges are weighted 0 (by defining $\delta(\varepsilon, \varepsilon) = 0$), the cost of paths and alignments coincide. Thus the problem of comparing $A$ and $R$ reduces to finding a least cost path between source vertex $(0, \theta)$ and sink vertex $(M, \phi)$. It is further shown in [14] that all substitution and deletion edges entering $\varepsilon$-labeled vertices except $\theta$ can be removed without destroying the property that there is a path corresponding to every possible alignment. These edges are removed in the example in Figure 4 to avoid a cluttered graph.

As in the case of SEQ($A$, $B$, $\{\delta\}$), one can formulate a recurrence for the shortest path cost to a vertex in terms of the shortest paths to its predecessors in the alignment graph:

$$(3) \quad C_{i,s} = \min\{ \min_{t\to s}\{C_{i-1,t} + \delta(a_i,\lambda_s)\},\ C_{i-1,s} + \delta(a_i,\varepsilon),\ \min_{t\to s}\{C_{i,t} + \delta(\varepsilon,\lambda_s)\}\ \}$$

Note that cyclic dependencies can occur in this recurrence, because the underlying alignment graph can contain cycles of insertion edges. One may wonder how such a "cyclic" recurrence makes sense. Technically, what we seek is the maximum fixed point to the set of equations posed by the recurrence. For problem instances where $\delta$ is such that a negative cost cycle occurs, the "optimal" alignment always involves an infinite number of copies of the corresponding insertion gap and has cost $-\infty$. Such a negative cycle can easily be detected in $O(P)$ time. For the more common and meaningful case where there are no negative weight cycles, the least cost path to any vertex must be cycle free, because any cycle adds a positive cost to the path. Moreover, by the reducibility of $F$ it follows that any such path contains at most one back edge from each copy of $F$ in the graph.

Miller and Myers used the above observations to arrive at the following row-based algorithm where the recurrence at each vertex is evaluated in two "topological" sweeps of each copy of $F$:

$C_{0,\theta} \leftarrow 0$
**for** $s \neq \theta$ in topological order of DAG edges **do**
  $C_{0,s} \leftarrow \min_{t\to s\in DAG}\{C_{0,t} + \delta(\varepsilon,\lambda_s)\}$
**for** $i \leftarrow 1$ **to** $M$ **do**
{ **for** $s$ in topological order of DAG edges **do**
  $C_{i,s} \leftarrow \min\{ \min_{t\to s}\{C_{i-1,t} + \delta(a_i,\lambda_s)\},\ C_{i-1,s} + \delta(a_i,\varepsilon),\ \min_{t\to s\in DAG}\{C_{i,t} + \delta(\varepsilon,\lambda_s)\}\ \}$
 **for** $s$ in topological order of DAG edges **do**
  $C_{i,s} \leftarrow \min\{\ C_{i,s},\ \min_{t\to s}\{C_{i,t} + \delta(\varepsilon,\lambda_s)\}\ \}$
}
"The score of the optimal alignment between $A$ and $R$ is $C_{M,\phi}$"

The set *DAG* in the algorithm above refers to the set of all DAG edges in $F$. Since $F$ restricted to the set of DAG edges is acyclic, a topological order for the **for**-loops exists. Observe that the algorithm takes $O(MP)$ time since each minimum operation involves at most 5 terms.

The algorithm sweeps the $i^{th}$ row twice in topological order, applying the relevant terms of the recurrence in each sweep. This suffices to correctly compute the values in the $i^{th}$ row, because any path from row $i-1$ to row $i$ is cycle free and consequently involves at most one back edge in row $i$. Suppose that a least cost path to vertex $(i, s)$ enters row $i$ at state $t$ along a substitution or deletion edge from row $i-1$. The least cost path from $t$ to $s$ consists of a sequence of DAG edges to a state, say $v$, followed possibly by a back edge $v \to w$ and another sequence of DAG edges from $w$ to $s$. The first sweep correctly computes the value at $(i, v)$, and the second sweep correctly computes the value at $(i, w)$ and consequently at $(i, s)$.

## 2.4 Sequence vs. Regular Expression Comparison with Gap Penalties

The introduction of a gap penalty scoring scheme for the regular expression pattern matching problem has an effect on the alignment graphs of RE($A$, $R$, $\{\delta\}$) similar to that of the sequence comparison problem. The set of nodes remains unchanged, but extra edges must be added to represent the multi-symbol gaps. The extra deletion edges in the graphs for RE($A$, $R$, $\{\delta, w\}$) are the same as in the graphs for SEQ($A$, $B$, $\{\delta, w\}$), i.e. edges from vertex $(i, s)$ to vertices $(k, s)$, for $k \in [i+1, M]$, each modeling the gap that leaves $a_{i+1}a_{i+2}\ldots a_k$ unaligned. For insertion edges the problem is more complex as there can be an infinite number of paths between two vertices in a row, each modeling the insertion of a different number of symbols. Due to this increased generality, it appears very difficult to treat the case of arbitrary $w$. The only result to date is an $O(MP(M+P))$ time algorithm by Myers and Miller that treats the case where $w$ is monotone increasing, i.e. $w(k) \leq w(k+1)$. With this restriction, a path between vertices $(i, t)$ and $(i, s)$ corresponding to a least cost insertion gap is a path between $t$ and $s$ that spells the fewest non-$\varepsilon$ symbols. Let $G_{t,s}$, hereafter called the *gap distance* between $t$ and $s$, be the number of non-$\varepsilon$ labeled states on such a path. Thus, it suffices to add a single edge from $(i, t)$ to $(i, s)$ of cost $w(G_{t,s})$ for every pair of vertices such that there is a path from $t$ to $s$ in $F$, denoted $t \xrightarrow{*} s$. Each of these insertion edges models an insertion gap of minimal cost over all gaps that leave a word spelled on the path from $t$ to $s$ unaligned. Precomputing $G_{t,s}$, for all pairs of $t$ and $s$, is a discrete shortest paths problem over a reducible graph, and hence can be done in $O(P^2)$ time.

The recurrence for the least cost path to vertex $(i, s)$ in the alignment graph described above is as follows:

$$C_{i,s} = \min\{ \min_{t \to s}\{C_{i-1,t} + \delta(a_i, \lambda_s)\},\ \min_{0 \leq k < i}\{C_{k,s} + w(i-k)\},\ \min_{\forall t: t \xrightarrow{*} s}\{C_{i,t} + w(G_{t,s})\} \} \tag{4}$$

Note that both the recurrence and the graph construction above require the assumption that $w(0) = 0$, as $G_{t,s}$ can be 0 for some state pairs.

Myers and Miller [14] show that the two sweep approach of the previous section correctly computes the shortest paths in this cyclic alignment graph if $w$ is *sub-additive*, i.e. for all $m$ and $n$, $w(m+n) \leq w(m) + w(n)$. In that paper, this condition is included to preclude sequences of insertion edges from consideration. In actuality, if a sequence of insertion edges from $t$ to $s$ is minimal over all such edge sequences, then one can show that the concatenation of the paths in $F$ corresponding to each insertion edge of the sequence must form an acyclic path in $F$. Thus, two sweeps suffice, even when $w$ is not sub-additive.

In the treatment that follows, we will focus on the case where $w$, in addition to being concave, is monotone increasing, i.e. $w(k) \leq w(k+1)$ for $k \geq 1$. If $w$ is concave but not monotone increasing, then the values of $w(k)$ rise to a global maximum and then descend to $-\infty$ as $k$ increases. Under this function, a best scoring insertion gap involves either the shortest *or the longest* sequence of symbols spelled on a path from $t$ to $s$ in $F$. When $R$ contains at least one Kleene closure operator, the Kleene closure admits arbitrarily long gaps scoring $-\infty$, and consequently such matching problems are automatically ill-posed, as discussed earlier. However, when $R$ contains no Kleene closures and $F$ is acyclic, then the problem is always well posed and involves an additional term $\min_{\forall t: t \xrightarrow{*} s}\{C_{i,t} + w(L_{t,s})\}$ in Equation 4 above. $L_{t,s}$ is the largest number of non-$\varepsilon$ symbols spelled on a path from $t$ to $s$. The algorithms in Section 4.1 can be modified to correctly solve for this additional term by replacing each instance of $G_{t,s}$ with $L_{t,s}$ in the equations and algorithms of that section. The complete algorithm solving this special case problem concurrently executes two versions of Section 4.1's algorithms, one computing with $G_{t,s}$ values and the other computing with $L_{t,s}$ values. The value of each $C_{i,s}$ is the minimum of the two computations at $i$ and $s$.

## 3 Minimum Envelopes and Generalized Candidate Lists

From this point forward, all comparison problems are assumed to be with respect to a gap penalty model where $w$ is concave. The algorithms of [8, 12, 5] employ the concept of a *minimum envelope* and its *candidate list* implementation to solve the sequence comparison problem SEQ($A$, $B$, $\{\delta, w\}$) in less than $O(NM(N+M))$ time. The algorithms of Section 4, which solve RE($A$, $R$, $\{\delta, w\}$), also use the same concept of a minimum envelope but require a more generalized form of its list implementation. This section describes the required generalization, beginning in Section 3.1 with a brief review of minimum envelopes and candidate lists as presented in [8, 12, 5]. Section 3.2 then illustrates the complications which arise from the introduction of regular expressions and specifies the more general candidate lists needed by the algorithms of Section 4. Finally, Section 3.3 presents the implementation of these generalized candidate lists.

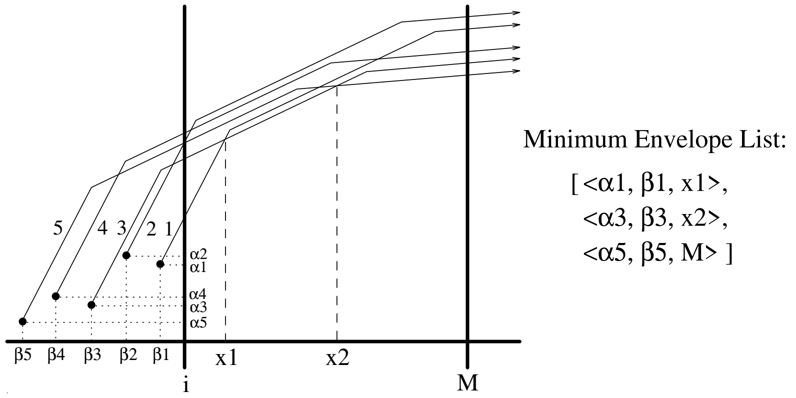



**Figure 5** A minimum envelope $E_i$ and its list representation.

## 3.1 Minimum Envelopes and SEQ($A$, $B$, $\{\delta, w\}$)

The inefficiency of the cubic algorithm of Section 2.2 is that, for each $i$ and $j$, it takes $O(M)$ time to compute the "deletion" term $\min_{0 \leq k < i}\{C_{k,j} + w(i-k)\}$ and $O(N)$ time for the "insertion" term $\min_{0 \leq k < j}\{C_{i,k} + w(j-k)\}$. This is the best one can do considering the computation of each $C_{i,j}$ in Equation 2 as an isolated problem. However, the sequence of deletion term minimums required over a given column share a correlation which if properly exploited permit the computation of each minimum in $O(\log M)$ time. Specifically for a given column $j$, one needs to deliver the sequence of deletion terms,

$$(5) \qquad D_i = \min_{0 \leq k < i}\{V_k + w(i-k)\}$$

where $V_k$ is $C_{k,j}$. Because the algorithm computes the $C_{i,j}$ in topological order of the alignment graph, it follows that the algorithm requires the values of the terms $D_i$ in increasing order of $i$. Further note that while all the $V_k$ are not known initially, those with $k < i$ have been computed at the time the value of $D_i$ is requested. An identical, "one-dimensional" problem models the insertion term computations along each row. Thus a more efficient comparison algorithm results if we can compute all the $D_i$ of Equation 5 in better than $O(M^2)$ time.

The key to achieving a faster algorithm for Equation 5 is to capture the contribution of the $k^{th}$ term in the minimum, called *candidate k*, at all future values of $i$. To do so, let $C_k(x) = V_k + w(x-k)$ be the *candidate curve* for candidate $k$, and let the *minimum envelope* at $i$ be the function $E_i(x) = \min_{0 \leq k < i}\{C_k(x)\}$ over domain $x \geq i$. Each curve $C_k$ captures the future contribution of candidate $k$, and the envelope $E_i$ captures the future contributions of the first $i$ candidates. Simple algebra from the definitions reveals that $D_i = E_i(i)$. Thus our problem can be reduced to incrementally computing a representation of $E_i$ for increasing $i$. That is, given a data structure modeling $E_i$, we need to efficiently construct a data structure modeling $E_{i+1}(x) = \min\{E_i(x), C_i(x)\}$.

Observe that each candidate curve is of the form $\alpha + w(\beta + x)$ for some $\alpha$ and $\beta$ (in the case of $C_k(x)$, $\alpha = V_k$ and $\beta = -k$). Thus all candidate curves are simply a translation of the curve $w(x)$ by $\alpha$ and $\beta$ in the $y$- and $x$-axes, respectively. Because every candidate is a translation of the same concave curve, it follows that any pair of such curves intersect at most once. To see this, consider two curves $c_1(x) = \alpha_1 + w(\beta_1 + x)$ and $c_2(x) = \alpha_2 + w(\beta_2 + x)$ where without loss of generality assume $\beta_1 \leq \beta_2$. At any given $x$, curve 1 is rising faster than curve 2 because concavity assures us that $\Delta w(\beta_1 + x) \geq \Delta w(\beta_2 + x)$. (Recall that $\Delta w(k) = w(k+1) - w(k)$ is the forward difference of $w$.) Thus either curve 1 never intersects curve 2, or curve 1 starts below curve 2 for small $x$, rises to intersect it as $x$ increases (potentially over an interval of $x$ as opposed to just a single point), and then stays above it for all larger $x$.

The minimum envelope at $i$, $E_i(x) = \min_{0 \le k < i}\{C_k(x)\}$, is the minimum of a collection of variously translated copies of the same concave curve as illustrated in Figure 5. As such, the value of $E_i$ at a given $x$ is the value of some candidate curve $C_k$ at $x$, in which case we say $C_k$ *represents* $E_i$ at $x$. Because concave curves intersect each other at most once, it follows that a given candidate curve represents the envelope over a single interval of $x$ values, if at all. Those candidates whose intervals are non-empty are termed *active*. Clearly, the set of intervals of active candidates partitions the domain of the envelope, and $E_i$ can be modeled by an ordered list of these candidates, $<c_1,\ c_2,\ \ldots c_h>$, in increasing order of the right endpoints of their intervals. The relevant information that needs to be recorded for an active candidate is captured in a record $c = <\alpha$ : real, $\beta$ : integer, $x$ : integer$>$. Record $c$ encodes an active candidate $k$ and its interval as follows: $c.\alpha = V_k$, $c.\beta = -k$, and $c.x$ gives the largest value of $x$ at which $C_k(x) = c.\alpha + w(c.\beta + x)$ represents the envelope. Formally, the envelope represented by such a list of records is given by:

(6) $\quad E_i(x) = c_j.\alpha + w(c_j.\beta + x)$ for $x \in [c_{j-1}.x + 1, c_j.x]$

where for convenience we define $c_0.x = i - 1$. By construction the candidates are ordered so that $c_{j-1}.x < c_j.x$. In addition, observe that it is also true that $c_{j-1}.\beta < c_j.\beta$ for all $j$ because curves with small $\beta$'s rise more quickly than those with larger $\beta$'s.

The equations $D_i = E_i(i)$ and $E_{i+1}(x) = \min\{E_i(x), C_i(x)\}$ suggest that computationally it suffices to have the operations (1) *Value* $(E)$ which delivers the *value* of envelope $E$ at $i$, (2) *Shift* $(E)$ which updates $E$ for the *shift* from $i-1$ to $i$, and (3) *Add* $(E, V_{i-1})$ to *add* the effect of the new candidate curve $C_{i-1}$ to $E$. Given these operations, the following algorithm computes the values of $D_i$ Equation 5:

$E \leftarrow [\,]$
**for** $i \leftarrow 1$ **to** $M$ **do**
{ $\quad E \leftarrow$ *Add* (*Shift* $(E), V_{i-1})$
$\quad\quad D_i \leftarrow$ *Value* $(E)$
}

where [ ] denotes an empty candidate list.

Computationally, a simple list data structure results in the following implementations of *Value*, *Shift* and *Add*. Operation *Value* simply returns $c_1.\alpha + w(c_1.\beta + i)$, where $c_1$ is the head of $E$. Operation *Shift* removes the head of $E$ if $c_1.x = i - 1$, since that candidate becomes inactive at the current value of $i$. For operation *Add*, $C_{i-1}$'s interval of representation in the new envelope must either be empty or must span from $i$ to the intersection point between $C_{i-1}$ and the envelope modeled by $E$. This is true because $C_{i-1}$ has a smaller $\beta$ value, $-(i-1)$, than any candidate in $E$ and so rises faster than those candidates. $C_{i-1}$'s interval is empty if $V_{i-1} + w(1) \ge c_1.\alpha + w(c_1.\beta + i)$, and an unaltered $E$ is returned in this case. Otherwise, the following steps create a candidate list modeling the new envelope: 1) remove $c_1, c_2, \ldots, c_h$ from $E$ where, for all $1 \le k \le h$, $V_{i-1} + w(-(i-1) + c_k.x) \le c_k.\alpha + w(c_k.\beta + c_k.x)$; 2) find the intersection point $x$ between $C_{i-1}$ and the surviving head of the list, which was $c_{h+1}$ in $E$; and 3) insert the record $<V_{i-1}, i-1, x>$ as the new head. The removed candidates are now inactive because $C_{i-1}$ is minimal over their intervals of representation, and the new candidate record models $C_{i-1}$'s role in the new envelope.

The time complexity of this algorithm is $O(M)$ times the computation needed to find each intersection point in operation *Add*. When the intersection point can be computed mathematically from $w$ in $O(1)$ time, the algorithm runs in $O(M)$ time. For a general concave gap-cost function $w$, a binary search over the range $[i, c_{h+1}.x]$ can find an intersection point in $O(\log M)$ time, resulting in an overall time bound of $O(M \log M)$ for the algorithm.

More recently than these minimum envelope algorithms, a series of papers [3, 4, 6, 10, 20] use a matrix searching technique originally presented in [1] to improve the worst case complexity for this problem and for a similar one-dimensional variation where $w$ is a *convex* function, i.e. one whose forward differences are nondecreasing instead of nonincreasing. That approach solves a column minima problem over an upper triangular matrix where, in essence, each row $k$ corresponds to the future values of candidate $C_k$ and the minimal value along each column $h$ corresponds to the solution for $D_h$ in Equation 5. The algorithms using this matrix searching technique solve the two one-dimensional problems in $O(M)$ time for convex $w$ and $O(M\,\alpha(M))$ time for concave $w$, where $\alpha(\ldots)$ is the inverse Ackermann function. These results are only mentioned here because they are not applicable to the regular expression problem. In fact, when regular expressions are introduced, it's no longer clear whether the basic definitions needed to perform matrix searching, i.e. a matrix-like structure which retains the quadrangle and inverse quadrangle inequalities, can be given.

### 3.2 Min. Envelopes and RE($A$, $R$, $\{\delta, w\}$)

The crux to designing an efficient algorithm for RE($A$, $R$, $\{\delta, w\}$) is to efficiently compute the insertion terms, $\min_{\forall t: t \xrightarrow{*} s} \{C_{i,t} + w(G_{t,s})\}$ of Equation 4. Along each "row" of the alignment graph, this involves solving the one-dimensional problem embodied in the following equation:

$$(7) \qquad I_s = \min_{\forall t: t \xrightarrow{*} s} \{V_t + w(G_{t,s})\}$$

For this problem, an envelope at $s$, $E_s$, captures the future contribution of its predecessors in order to expedite the computation of $I$-values at $s$'s successors in the automaton for $R$. The more complex structure of $F$ gives rise to a number of complications.

First, in the simpler sequence comparison context, it is natural to use the coordinate system of the alignment graph as a frame of reference for $x$, i.e. $E_i(x)$ is the value $E_i$ contributes to the vertex in column $x$. The analogous definition in the case of regular expressions is to define $E_s(x)$ to be the value that $E_s$ contributes to any state whose gap distance from the start state, $\theta$, is $x$. However, this fails for regular expressions because there are automata for which two states at equal gap distance from $\theta$ are at distinct gap distances from $s$. It is thus essential to make $s$ the referent of the parameter $x$. Specifically, $E_s$ must be constructed so that $E_s(x)$ is the value envelope $E_s$ contributes to any state whose gap distance from $s$ is $x$.

Also, unlike the sequence comparison case, there are state pairs $t$ and $s$ whose gap distance $G_{t,s}$ is 0. The algorithms of Section 4 are greatly simplified when the candidates from such states $t$ can be included into the candidate lists at $s$, instead of delaying their inclusion until some future $s$ whose gap distance from $t$ is nonzero. However, candidates with a $\beta$-value of 0 cannot be treated under the minimum envelope formulation described in the previous section, because $w$ is defined to be concave only for values $k \geq 1$, and $w(1) - w(0)$ might not be greater than or equal to $w(2) - w(1)$. The candidate list implementation must be extended to handle these 0 $\beta$-value candidates separately.

The most significant complication arises because of the multiple paths introduced by the alternation and Kleene closure sub-automata. This complication requires a more complex algorithm which models each $E_s$ by $O(\log P)$ candidate lists. First, in any incremental computation where the candidate lists modeling envelope $E_s$ are built from the lists at predecessor states, the candidate lists at the start state of each $F_{R\,|\,S}$ and $F_{R^*}$ sub-automaton are used separately at its two successor states. The construction at each state makes different changes to the original lists at the start state, and neither set of changes can be allowed to affect the other. Because of this, the implementation of candidate lists and their operations must be made *applicable*, or *non-destructive*, so that the construction occurring along a path through the NFA does not affect the construction on concurrent paths through $F$.

At the final states of alternation and Kleene closure sub-automata, the candidate lists from multiple predecessors must be merged together to construct lists modeling $E_{\theta_{R\,|\,S}}$ (or $E_{\theta_{R^*}}$). This could easily be done using either a merge sort style sweep through the active candidates of the two lists or by adding the active candidates from one list into the other. However, because the candidate lists can be of size $O(P)$, such a merge operation yields an $O(P^2)$ time bound for the computation of the $I_s$ values. Barring a more efficient merge operation, it does not appear that any algorithm which uses either a single or $O(1)$ candidate lists can compute the $I_s$ values in less than $O(P^2)$ time. There are too many repeated candidates in the merges: either candidates from the same state $t$ occurring in the lists being merged at a single $\theta_{R\,|\,S}$ (or $\theta_{R*}$), or candidates from the same state $t$ needed at the series of final states occurring in NFA's $F$ with highly nested $F_{R\,|\,S}$ and $F_{R*}$ sub-automata.

Given these complications, the algorithms solving RE($A$, $R$, $\{\delta, w\}$) require a more generalized implementation of candidate lists. Proceeding formally, let $E$ be a candidate list data structure modeling a minimum envelope, and let $E(x)$ denote the value of the encoded envelope at $x$. The goal is to develop applicative, i.e. non-destructive, procedures for these four operations:

(1) *Value* $(E, x)$: returns $E(x)$ when $x \geq 0$.
(2) *Shift* $(E, \Delta)$: returns a candidate list $E'$ for which $E'(x) = E(x + \Delta)$ when $\Delta \geq 0$.
(3) *Add* $(E, \alpha, \beta)$: returns a candidate list for $E'(x) = \min\{E(x), \alpha + w(\beta + x)\}$ when $\beta > 0$.
(4) *Merge* $(E1, E2)$: returns a candidate list $E'(x) = \min\{E1(x), E2(x)\}$.

The next section shows how the first three operations above can be accomplished in logarithmic time. Then, operation *Merge* simply uses *Add* to add the candidates from the shorter list into the longer list, as follows:

*Merge* $(E1, E2)$
{ **if** $Len(E1) \leq Len(E2)$
  **then** $E \leftarrow E1$; $E' \leftarrow E2$
  **else** $E \leftarrow E2$; $E' \leftarrow E1$
 **for** $c \in E$ **do**
  $E' \leftarrow$ *Add* $(E', c.\alpha, c.\beta)$
 **return** $E'$
}

Since the active candidates of $E'$ must also be active candidates in $E1$ and $E2$, the minimum envelope formed from combining the active candidates of $E1$ and $E2$ must model $\min\{E1(x), E2(x)\}$ for all $x$. The time complexity for this operation is the length of the smaller candidate list times the logarithmic cost for each *Add*.

These candidate list operations, and the "frame shift" required by the regular expression algorithm, generalize the one-dimensional problem of Section 3.1 in the following manner. $E_i(x)$ is now defined over the domain $x \geq 0$ and equals the contribution of the envelope at $i$ to the vertex at column $i + x$ (as opposed to the one at column $x$). Formally, $E_i(x) = \min_{0 \leq k < i}\{C_k(i + x)\}$ for $x \geq 0$. Some straightforward algebra reveals that this new definition now implies that $D_i = E_i(0)$ and $E_{i+1}(x) = \min\{E_i(x+1), C_i(x+(i+1))\}$. As before, the envelope is modeled by an ordered list of candidate records, but now $c.\beta = i - k$ and $c.x$ is the largest value of $x$ at which $C_k(i+x)$ represents the envelope. With these changes, the following $O(M \log M)$ algorithm correctly computes the $D_i$ values specified by Equation 5:

$E \leftarrow [\,]$
**for** $i \leftarrow 1$ **to** $M$ **do**
{ $E \leftarrow$ *Add* (*Shift* $(E, 1), V_{i-1}, 1)$
 $D_i \leftarrow$ *Value* $(E, 0)$
}

### 3.3 Generalizing the Minimum Envelope Lists

To simplify the initial development of the operations, we first look at the effect each operator has on the candidate list $E$ without regard to efficiency or the method via which the list is implemented. The typical list

$rp(k) \ \equiv \ E_k.x$ # the rightmost point of $E_k$
$lp(k) \ \equiv$ **if** $k = 1$ **then** $0$ **else** $E_{k-1}.x + 1$ # the leftmost point of $E_k$

$RE(k) \equiv \ m = 0$ **or** $E_k(rp(k)) \leq \alpha + w(\beta + rp(k))$ # In *Add*, these test to see if $E_k$ is
$LE(k) \equiv \ m = Len(E)$ **or** $E_k(lp(k)) \leq \alpha + w(\beta + lp(k))$ # minimal at $rp(k)$ and $lp(k)$, resp.

*Value* $(E, x)$
{ **if** $Len(E) = 0$ **then return** $\infty$
  $j \leftarrow$ *Findmin* $(k : x \leq rp(k))$
  **return** $E_j(x)$
}

*Shift* $(E, \Delta)$
{ **if** $Len(E) = 0$ **then return** $E$
  $j \leftarrow$ *Findmin* $(k : \Delta \leq rp(k))$
  $F \leftarrow$ *Offset* $(E_{j..Len(E)}, \Delta)$
  **return** $F$
}

*Add* $(E, \alpha, \beta)$
{ **if** $Len(E) = 0$ **then return** $[<\alpha, \beta, \infty>]$
  $m \leftarrow$ *Findmax* $(k : \beta \geq E_k.\beta)$
  **if** $RE(m)$ **and** $LE(m+1)$ **then return** $E$
  $l \leftarrow$ *Findmax* $(k : k \leq m \ \& \ LE(k))$
  $h \leftarrow$ *Findmin* $(k : k > m \ \& \ RE(k))$
  $F \leftarrow$ *Concat* $(E_{1..l}, \ [<\alpha, \beta, \infty>], \ E_{h..Len(E)})$
  **if** $l > 0$ **then**
    $F_l.x \leftarrow$ *Intersect* $(F_l, F_{l+1})$
  **if** $h \leq Len(E)$ **then**
    $F_{l+1}.x \leftarrow$ *Intersect* $(F_{l+1}, F_{l+2})$
  **return** $F$
}

**Figure 6** The procedures *Value*, *Shift*, and *Add*.

operations, plus these five additional operations, are assumed; their implementation will be discussed later.

(1) $e(x)$: for candidate record $e$, returns $e.\alpha + w(e.\beta + x)$.
(2) *Findmin* $(k : P(k))$: returns minimal $k$ s.t. predicate $P(k)$ is true (or $\infty$ if its never true).
(3) *Findmax* $(k : P(k))$: returns maximal $k$ s.t. predicate $P(k)$ is true (or 0 if its never true).
(4) *Intersect* $(e, f)$: for records $e$ and $f$, returns the maximal $x$ such that $e(x) < f(x)$.
(5) *Offset* $(E, \Delta)$: returns $E'$ such that, for all $i$, $E'_i.\beta = E_i.\beta + \Delta$ and $E'_i.x = E_i.x - \Delta$.

*Findmin* and *Findmax* are used to perform searches over a candidate list $E$, and as such the index $k$ ranges over $[1, Len(E)]$. All the predicates $P(k)$ that occur in the calls to $Findmin$ below are nondecreasing in that $P(k) \leq P(k+1)$ for all $k$, where *false* is considered to be less than *true*. Similarly, all the predicates used in calls to *Findmax* are nonincreasing. Note that from an implementation perspective, this implies that a binary search can be used. In the event that $P(k)$ is false for all $k$, the description above states that *Findmin* returns $\infty$. We will use $\infty$ in such contexts to denote a number that is sufficiently large. In all cases such a number is easy to arrive at, e.g. $Len(E) + 1$ in the case that the predicate of $Findmin$ is defined over list $E$.

Figure 6 presents the procedures implementing *Value*, *Shift*, and *Add*. The realization of operation *Value* follows directly from Equation 6 in Section 3.1. The call to *Findmin* sets $j$ to the index of the candidate whose interval contains $x$, i.e. $x \in [lp(j), rp(j)]$, and then the value $E_j(x)$ is returned. For the $\infty$ occurring in *Value*, an appropriate choice when solving RE($A$, $R$, $\{\delta, w\}$) is $w(1) \cdot (M + P) + 1$.

Operation *Shift* requires adding $\Delta$ to each record's $\beta$-field and subtracting it from each record's $x$-field, since in the desired envelope $E'_k.\beta = E_k.\beta + \Delta$ and $E'_k.x = E_k.x - \Delta$ for all $k$. However, some candidates become inactive because the right end of their intervals become less than 0 in the new envelope. Specifically, this is true exactly for those candidates whose $x$-field is less than $\Delta$. Because records are in increasing order of this field, the call to *Findmin* finds the leftmost candidate which remains active. The sublist to the right of

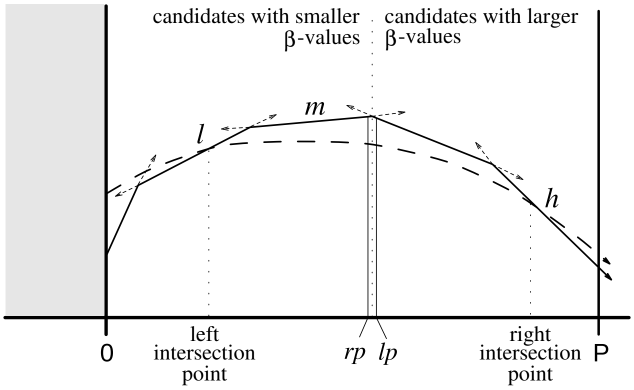



**Figure 7** Adding a curve to a minimum envelope.

that candidate, inclusively, is extracted from $E$, and then operation *Offset* updates the $x$- and $\beta$-fields of the remaining candidates.

*Add* is the most complex, potentially requiring the replacement of an interior sublist of the current envelope with the new candidate, as shown in Figure 7. Recall that the candidates in a list occur in increasing order of their $\beta$-fields. Thus, the new candidate's interval of representation, if not empty, occurs between the candidates in $E$ with lesser $\beta$'s and those with greater $\beta$'s. If the new candidate is not minimal at the division point between these two sublists of $E$, then its interval is empty because it falls more slowly than the candidates with smaller $\beta$'s (moving leftward) and rises more quickly than the candidates with larger $\beta$'s (moving rightward). The first call to *Findmax* finds the index $m$ such that 1) $E_k.\beta \leq \beta$ for all $1 \leq k \leq m$ and 2) $E_k.\beta > \beta$ for all $m < k \leq Len(E)$. Let $rp = rp(m)$ when $m > 0$ and 0 otherwise, and let $lp = lp(m+1)$ when $m < Len(E)$ and $\infty$ otherwise[1]. The predicates RE($m$) and LE($m+1$) compare the new candidate against $E_m$ and $E_{m+1}$ at $rp$ and $lp$ respectively. They also include clauses "$m = 0$" and "$m = Len(E)$" which check for the boundary conditions. If both $RE(m)$ and $LE(m+1)$ are true, then the new candidate cannot contribute to the left of $rp$ or to the right of $lp$ and its interval of representation must be empty. In this case, *Add* simply returns the unaltered list $E$.

The alternative is that at least one of the predicates $RE(m)$ or $LE(m+1)$ is false, in which case the candidate represents the desired envelope over some interval containing either $lp$ or $rp$, or both. Thus, it suffices to find the left and right points $x_l \in [0, rp]$ (or $x_l = lp$ if $RE(m)$ is false) and $x_h \in [lp, \infty]$ (or $x_h = rp$ if $LE(m+1)$ is false) where the new candidate intersects the envelope for $E$. The call to *Findmax* finds the rightmost candidate, $l$, which is less than the candidate at the left endpoint of $l$'s interval. Either candidate $l$'s interval contains the left intersection point, or $l = m$ and the left intersection point is $lp$. The call to *Findmin* similarly returns the candidate, $h$, containing the right intersection point. *Add* then replaces the candidates strictly between $l$ and $h$ with the new candidate. This is correct, as the new candidate represents $E'(x)$ over the intervals of the candidates just removed. Finally, the exact location of the left and right intersection points must be stored in the $x$-fields of the records for $l$ and the new candidate, respectively. The call, *Intersect* $(F_l, F_{l+1})$ finds the left endpoint as $F_{l+1}$ is the new candidate record, and the call *Intersect* $(F_{l+1}, F_{l+2})$ finds the right endpoint.

Attention is now turned to the efficient implementation of the candidate list data structure. Each candidate list is implemented as a height-balanced tree of candidate records such that the list is given by an inorder traversal of the tree. This well-known representation for a linear list [11] permits all of the typical list operations, plus the binary search used by *Findmax* and *Findmin*, in either constant or logarithmic time of the

[1] $P$ suffices for the sequence vs. regular expression problem as longer gap distances will not be encountered. Similarly $M$ and $N$ suffice for the one-dimensional column and row problems posed by sequences.

length of the list. In addition, Myers [13] presents an implementation for applicatively manipulating height-balanced trees at no additional time overhead. This implementation simply modifies the standard operations to make copies of any vertices that normally would be destructively modified. Note that this approach does require extra space: each $O(\log P)$ operation takes $O(\log P)$ space as well. This overhead is reflected in the space complexity of our final result.

Primitive operation (4), *Intersect*$(e, f)$, involves the monotone predicate $e(x) < f(x)$ whose range is restricted to $x \in [0, P]$, because of the concave curves' single intersection property and because $P$ is a suitable choice for $\infty$ in the algorithm for RE($A$, $R$, $\{\delta, w\}$). Thus an $O(\log P)$ binary search over this range implements *Intersect*. Operation *Offset* $(E, \Delta)$, primitive operation (5), can be realized in $O(1)$ time over a height-balanced tree as noted for link-cut trees [17]. The "trick" is to store the $\beta$ and $x$ values for a candidate record as *offsets relative* to the $\beta$ and $x$ values at the parent of the candidate in the tree. With such a scheme, $E_i.\beta$ is obtained by summing the $\beta$-offsets of the vertices on the path from the root of $E$'s tree to the vertex representing candidate $i$. Since this path must be traversed in order to access candidate $i$, the computation accrues no asymptotic overhead. With this structure, *Offset*$(E, \Delta)$ simply involves adding and subtracting $\Delta$ to the $\beta$ and $x$ values at the root. However, whenever the structure of the tree is modified by an operation, such as a height balancing rotation, one must carefully readjust the offsets stored at each vertex. For example, consider the rotation from $x(a, y(b, c))$ to $y(x(a, b), c)$ where $x(L, R)$ denotes a tree node $x$ with left and right children $L$ and $R$. Then, if *x.off* denotes the offset before and $x.\mathit{off}'$ denotes the offset after, it suffices to set $y.\mathit{off}' = x.\mathit{off} + y.\mathit{off}$, $x.\mathit{off}' = -y.\mathit{off}$, $a.\mathit{off}' = a.\mathit{off}$, $b.\mathit{off}' = b.\mathit{off} + y.\mathit{off}$, and $c.\mathit{off}' = c.\mathit{off}$ in order to preserve all absolute values. These changes are made during the rotation at no additional asymptotic overhead. Such value-preserving transforms are available for the other necessary rotation operations. A complete description of one such schema is given in [17].

The final consideration is the extension of candidate lists to include candidates whose $\beta$-value is zero. These extended lists consist of two parts: $E.\mathit{list}$, a "standard" envelope for the candidates whose $\beta$-value is greater than zero; and $E.\alpha 0$, the $\alpha$ value of the best candidate with a $\beta$ of zero. For such a modified data structure, the routines *Value*$^+$, *Shift*$^+$, and *Add*$^+$ below give the necessary operational extensions. To simplify the algorithm descriptions presented in the rest of the paper, the $^+$ symbols will be omitted and assumed by the use of *Value*, *Shift* and *Add*.

*Value*$^+(E, x)$
$\{$ $a \leftarrow E.\alpha 0 + w(x)$
  $b \leftarrow$ *Value* $(E.\mathit{list}, x)$
  **return** $\min\{a, b\}$
$\}$

*Shift*$^+(E, \Delta)$
$\{$ **if** $\Delta = 0$ **then return** E
  $E.\mathit{list} \leftarrow$ *Shift* $(E.\mathit{list}, \Delta)$
  **if** $E.\alpha 0 \neq \infty$ **then**
  $\{$ $E.\mathit{list} \leftarrow$ *Add* $(E.\mathit{list}, E.\alpha 0, \Delta)$
    $E.\alpha 0 \leftarrow \infty$
  $\}$
  **return** E
$\}$

*Add*$^+(E, \alpha, \beta)$
$\{$ **if** $\beta = 0$ **then**
    $E.\alpha 0 \leftarrow \min\{E.\alpha 0, \alpha\}$
  **else**
    $E.\mathit{list} \leftarrow$ *Add* $(E.\mathit{list}, \alpha, \beta)$
  **return** E
$\}$

# 4 Regular Expression Pattern Matching

As in the sequence vs. sequence recurrence, the deletion term, $\min_{0 \le k < i}\{C_{k,s} + w(i-k)\}$, and the insertion term, $\min_{\forall t: t \xrightarrow{*} s}\{C_{i,t} + w(G_{t,s})\}$, in the recurrence for RE($A$, $R$, $\{\delta, w\}$) give the cubic time bound for the naive two-sweep algorithm described in Section 2.4. The previous section shows that the problem of delivering the deletion terms in a given column constitutes a "one-dimensional" problem solvable in $O(M \log M)$ time. This section formulates the problem of delivering the insertion terms in a row as a more complex one-dimensional problem and solves it in $O(P \log^2 P)$ time. The insertion terms are computed in two sweeps as with the regular expression algorithm of Section 2.3. Combining these one-dimensional solutions with the two-sweep dynamic programming computation yields an algorithm that computes $C_{M,\phi}$, the cost of the optimal alignment between $A$ and $R$, in $O(MP(\log M + \log^2 P))$ time.

Consider the computation of the values at vertices in a given row, say $i$, of the alignment graph. Let $D_s = \min_{0 \le k < i}\{C_{k,s} + w(i-k)\}$ be the deletion term at state $s$, and let $S_s = \min_{t \to s}\{C_{i-1,t} + \delta(a_i, \lambda_s)\}$ be the substitution term at state $s$. As noted above, the deletion terms at each state in the row can be delivered by maintaining $O(P)$ envelopes for the one-dimensional problems progressing down each state's column. Thus, we can compute $V_s = \min\{S_s, D_s\}$ at each state $s$ in row $i$ in $O(P \log M)$ time. Moreover, $C_{i,s} = \min_{\forall t: t \overset{*}{\to} s}\{V_t + w(G_{t,s})\}$ because $w$ is concave and thus sub-additive, i.e. $w(m+n) \le w(m) \cdot w(n)$. This follows because, when $w$ is sub-additive, the best path to a vertex in the alignment graph ends with at most one insertion edge. Thus, the critical problem of computing the value at a row becomes that of solving the one-dimensional problem given by Equation 7 of the previous section.

This one-dimensional problem is now cast in terms of envelopes. What is desired at each $s$ is

$$E_s(x) = \min_{\forall t: t \overset{*}{\to} s} \{V_t + w(G_{t,s} + x)\}$$

whereupon the desired $C$-value at each vertex is simply $E_s(0)$. Note that in the minimum above, there is a single term or candidate for each state of the automaton. Because states and candidates are in one-to-one correspondence, candidates in an envelope will often be referred to or characterized in terms of their originating states. These envelopes are actually arrived at in two topological sweeps of $R$'s NFA, as part of the overall two-sweep algorithm proceeding across each alignment graph row. In the first sweep, the envelopes $E1_s$ consider only gaps whose underlying paths are restricted to DAG edges, and the second sweep envelopes $E2_s$ consider the gaps whose paths have exactly one back edge. Formally,

$$E1_s(x) = \min_{\forall t: t \overset{*}{\to} s} \{V_t + w(G_{t,s} + x) \mid t \overset{*}{\to} s \in DAG^*\}$$

$$E2_s(x) = \min_{\forall t: t \overset{*}{\to} s} \{V_t + w(G_{t,s} + x) \mid t \overset{*}{\to} s \in DAG^* \times BCK \times DAG^*\}$$

Because only cycle free paths must be considered, it follows that $E_s(x) = \min\{E1_s(x), E2_s(x)\}$ and $C_{i,s} = E_s(0)$. In the remainder of the paper, the path restriction clauses $DAG^*$ and $DAG^* \times BCK \times DAG^*$ in the definitions above are omitted and assumed by the use of either a 1 or a 2 in the name of the defined quantity. As will be seen, the envelopes $E1$ and $E2$ are actually modeled by a collection of up to $O(\log P)$ distinct candidate lists.

## 4.1 The First Sweep Algorithm

To compute the envelope $E1$ at every state, it suffices to consider only the subgraph of the NFA $F$ restricted to $DAG$ edges. Over this acyclic subgraph, the set of *predecessors* of state $s$ are those states $t$ where $t \overset{*}{\to} s$, and they are all enumerated before $s$ in a topological sweep of $F$. These predecessors are partitioned into two sets, the *up predecessors* and *down predecessors* at $s$, and their candidates are stored in two data structures, called the *up list* and *down list*. This division is based on the predecessors' position, relative to $s$, in the nesting structure of $F$ induced by Kleene closures and alternations in $R$. This notion of up and down can be better illustrated by hypothetically extending the two-dimensional NFA's pictured in this paper into a third dimension. The positions of states in that third dimension depend on their position in the nesting of alternation and Kleene closure sub-automata of $F$. The highest tier of states contain all of the states in $F$ which are outside any $F_{R \mid S}$ or $F_{R^*}$ sub-automaton. States in successively nested sub-automata are placed in successively lower tiers, and the most deeply nested sub-automaton's states make up the lowest tier. The descriptive terms "up" and "down" used in this section refer to the directions up and down in this third dimension.

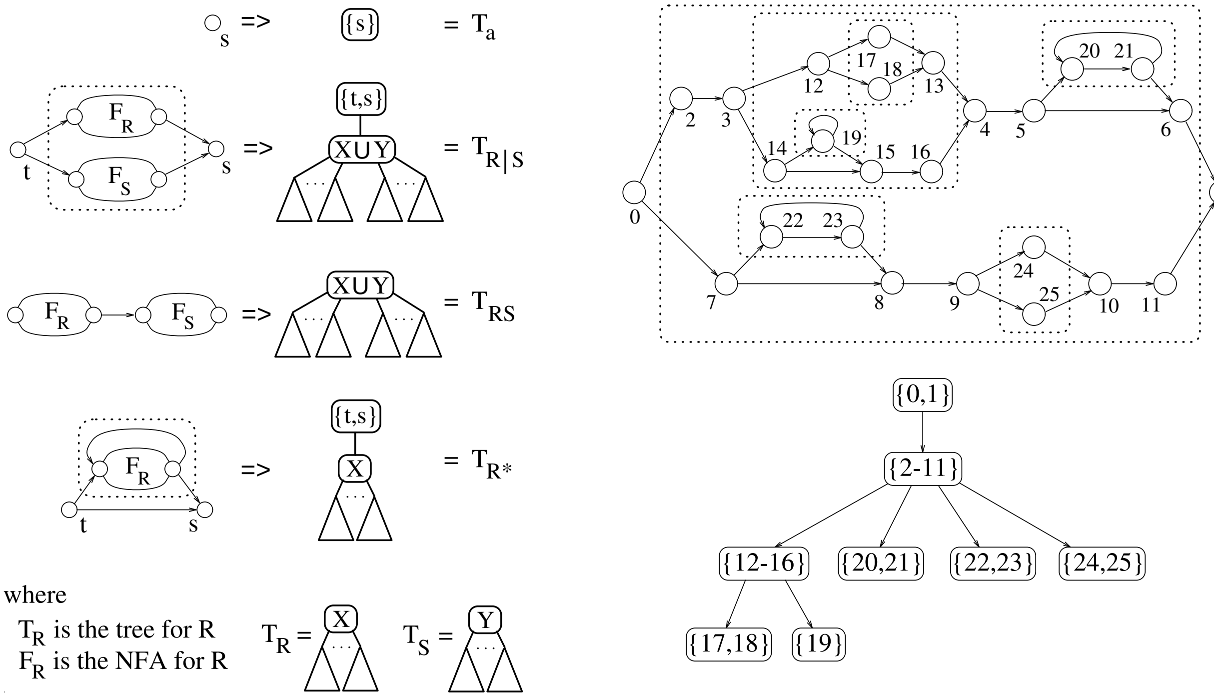



**Figure 8** The nesting tree construction and an example nesting tree.

To capture this nesting structure and each state's position in the nesting, a *nesting tree* is constructed from $F$. The formal inductive construction is specified in Figure 8, and an example tree is also given. Informally, the nesting tree consists of a node corresponding to the subexpression formed by each alternation and Kleene closure operator in $R$ and a root node corresponding to $R$ itself. The edges of the tree model the immediate nesting structure of node subexpressions. A node's *submachine* is that sub-automaton in $F$ induced by the node's subexpression. Each node of the nesting tree is also annotated with a *node set* consisting of those NFA states in the node's submachine except (1) its start and final states, and (2) those belonging to any descendant node. Thus, the node sets form a partition of the states for $F$, and the relative position of two states in the nesting of submachines is mirrored in the relative position of the states in the tree. For a state $s$, let $N_s$ denote the unique node whose node set contains $s$.

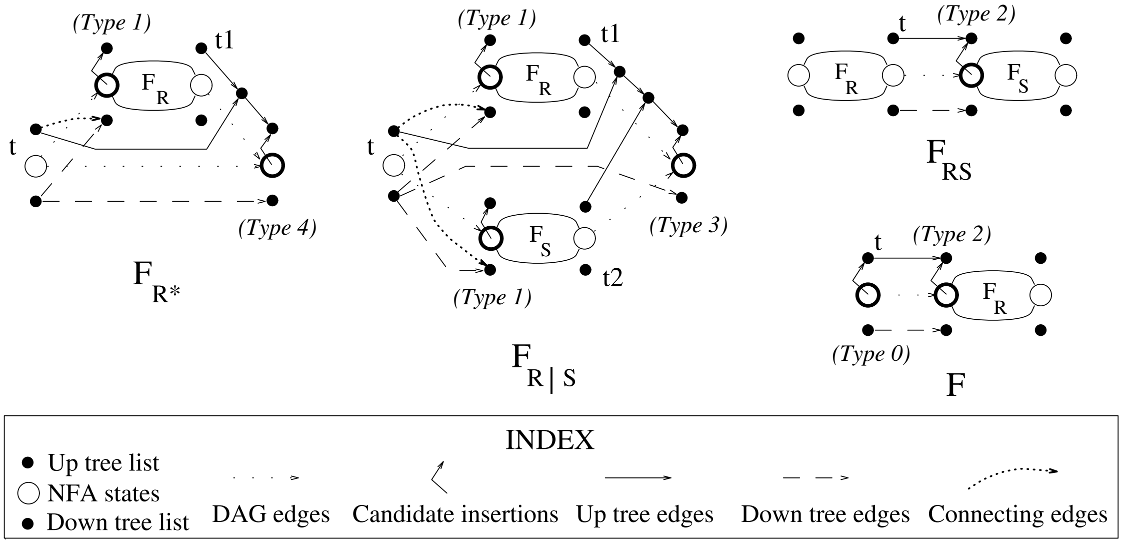
The up predecessors of a state $s$ are those states $t$ for which $t \overset{*}{\rightarrow} s$ and $N_s \overset{*}{\rightarrow} N_t$, i.e. $N_s$ is an ancestor of $N_t$ in the nesting tree. All other predecessors of $s$ are down predecessors. They are so named because any NFA path from a down predecessor $t$ to $s$ contains edges which correspond to moving down the nesting tree. Given the partition of predecessors into up and down types, one can then decompose the computation of $E1_s$ into the computation of:

$$EU1_s(x) = \min_{\forall t: t \overset{*}{\rightarrow} s} \{V_t + w(G_{t,s} + x) \mid N_s \overset{*}{\rightarrow} N_t\}$$

$$EH1_s(x) = \min_{\forall t: t \overset{*}{\rightarrow} s} \{V_t + w(G_{t,s} + x) \mid N_s \overset{*}{\not\rightarrow} N_t\}$$

where $E1_s(x) = \min\{EU1_s(x), EH1_s(x)\}$. The rest of this section is devoted to presenting the construction of the up and down list data structures, showing that these constructions model envelopes $EU1$ and $EH1$ respectively, and giving complexity claims for the construction algorithms.

Both the up and down lists at a state $s$ are constructed incrementally from the lists at $s$'s predecessor states. To illustrate these incremental computations, *flow graphs* called the *up tree* and *down tree* are used to show




Type 0: The start state, $\theta$ (no predecessor states).
Type 1: Inner start states in $F_{R\,|\,S}$ and $F_{R^*}$ (predecessor state $t$).
Type 2: Inner start states in $F_{RS}$ and $F$ (predecessor state $t$).
Type 3: The final state of $F_{R\,|\,S}$ (immediate predecessors $t1$ and $t2$, $F_{R\,|\,S}$ start state $t$).
Type 4: The final state of $F_{R^*}$ ($F_{R^*}$ start state $t$, other predecessor $t1$).

**Figure 9** The inductive construction of the up and down trees at each state $s$.

the movement of candidate curves through the data structures constructed at each state. Figure 9 gives the inductive construction of the flow graphs. Figures 10 and 11 give example up and down trees over an NFA. The incoming edges at each point in the two trees describe the candidates and candidate lists which must be included in the construction of the two data structures at each state. The up tree edges, down tree edges and connecting edges specify the inclusion of the up or down list from a predecessor state. The candidate insertion edges specify the insertion of that state's candidate curve into the up list. These flow graphs are used only for illustration, so no proofs are given for the graphs' correctness. Such proofs, however, can be inferred from the correctness proofs given for the up and down list constructions.

The overall structure of the construction algorithm consists of a **for**-loop ranging over the states in $F$ in topological order. At each state, a *construction step* is applied to construct the up and down lists at that state. The construction step executed at a state $s$ depends on $s$'s *state type*. There are five state types, labeled 0 to 4 in Figure 9. Figure 9 also gives the predecessor state notation which is used throughout this section. The single type 0 state is the start state $\theta$, and it has no predecessors. Type 1 states are the start states of the sub-automata $F_R$ and $F_S$ inside each $F_{R\,|\,S}$ and $F_{R^*}$ sub-automaton. $t$ denotes the single predecessor of each type 1 state. Type 2 states are the start states of $F_S$ inside each $F_{RS}$ sub-automaton, and $t$ also denotes the predecessor of each type 2 state. Type 3 and 4 states are final states of $F_{R\,|\,S}$ and $F_{R^*}$ sub-automata, respectively. The predecessors of the type 3 states are $t1 = \phi_R$, $t2 = \phi_S$, and $t = \theta_{R\,|\,S}$. The predecessors of the type 4 states are $t = \theta_{R^*}$ and $t1 = \phi_R$. These five types categorize every state in an NFA, because each state must either be the initial state of a sub-automaton at some step in the inductive construction or must be the final state of an alternation or Kleene closure sub-automaton (this can be seen from Figure 9).

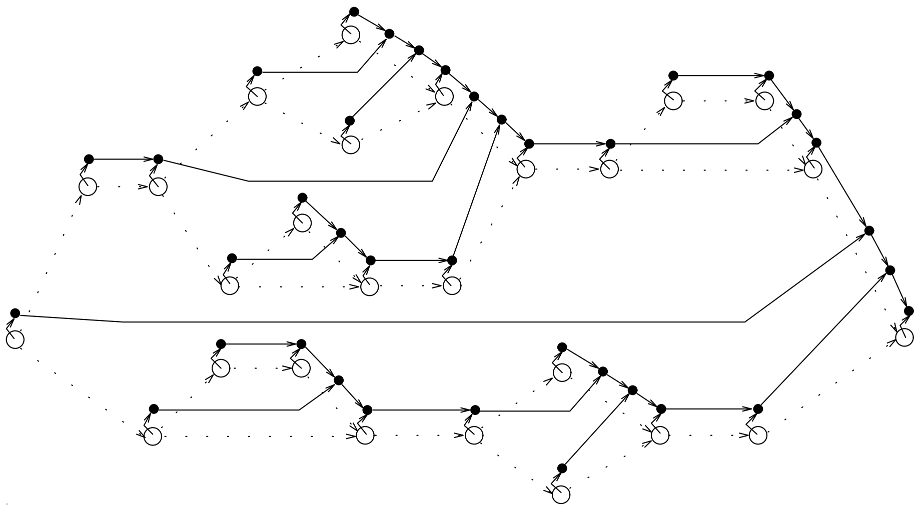


**Figure 10** The up tree for an example NFA.

### 4.1.1 Up List Construction

The up list data structure consists of a single candidate list, denoted $U_s$, containing all of the candidates needed to model $EU1_s$. The construction steps for the up list are as follows:

Type 0: $U_\theta \leftarrow \textit{Add}\,([\,], V_\theta, 0)$
Type 1: $U_s \leftarrow \textit{Add}\,([\,], V_s, 0)$
Type 2: $U_s \leftarrow \textit{Add}\,(\textit{Shift}\,(U_t, \lambda_s \neq \varepsilon), V_s, 0)$
Type 3: $U_s \leftarrow \textit{Merge}\,(U_{t1}, \textit{Shift}\,(U_t, G_{t,s}))$
$U_s \leftarrow \textit{Add}\,(\textit{Merge}\,(U_s, U_{t2}), V_s, 0)$
Type 4: $U_s \leftarrow \textit{Add}\,(\textit{Merge}\,(U_{t1}, U_t), V_s, 0)$

where [ ] denotes an empty candidate list and the expression $\lambda_s \neq \varepsilon$ returns either 1 or 0 if the expression is true or false.

**LEMMA 1.** For any NFA $F$, the constructed $U_s$ models $EU1_s$ for each state $s \in F$.

**Proof.** By induction on $s$ over the topological ordering of the states according to the DAG edges. For the base case $s = \theta$ and all type 1 states, $EU1_s(x) = V_s + w(x)$ since $s$ itself is the only state which can reach $s$ and appears in the nesting sub-tree rooted at $N_s$. Thus, adding $s$'s candidate to an empty list constructs an envelope modeling $EU1_s$. The type 2 states are analogous to the sequence comparison recurrence in that $EU1_s(x) = \min\{EU1_t((\lambda_s \neq \varepsilon) + x), V_s + w(x)\}$, and the construction first shifts $U_t$ (if $\lambda_s \neq \varepsilon$) and then adds $s$'s candidate. The type 3 and 4 states are the most interesting cases. For the type 3 states, $EU1_s$ can be rewritten as follows:

$$
\begin{aligned}
EU1_s &= \min_{\forall v: v \xrightarrow{*} s} \{V_v + w(G_{v,s} + x) \mid N_s \xrightarrow{*} N_v\} \\
&= \min\{ \min_{\forall v: v \xrightarrow{*} t1} \{V_v + w(G_{v,t1} + x) \mid N_{t1} \xrightarrow{*} N_v\}, && \text{\# the states in } F_R \\
&\qquad\quad \min_{\forall v: v \xrightarrow{*} t2} \{V_v + w(G_{v,t2} + x) \mid N_{t2} \xrightarrow{*} N_v\}, && \text{\# the states in } F_S \\
&\qquad\quad \min_{\forall v: v \xrightarrow{*} t} \{V_v + w(G_{v,t} + \mathrm{G}_{t,s} + x) \mid N_t \xrightarrow{*} N_v\}, && \text{\# up predecessors of } t = \theta_{R\,|\,S} \\
&\qquad\quad V_s + w(x)\ \} && \text{\# } s\text{'s candidate} \\
&= \min\{EU1_{t1}(x),\ EU1_{t2}(x),\ EU1_t(x + G_{t,s}),\ V_s + w(x)\}
\end{aligned}
$$

because (1) all paths to $s$ must either originate from inside $F_R$ and $F_S$ or must pass through $t$, (2) $G_{t1,s}$ and $G_{t2,s}$ equal 0, and (3) $N_t = N_s \rightarrow N_{t1} = N_{t2}$. Thus, combining $U_{t1}$, $U_{t2}$, a shifted $U_t$, and the candidate from $s$ constructs a data structure modeling $EU1_s$. $EU1$ at the type 4 states can be rewritten similarly. $\square$

The up list construction algorithm takes $O(P \log^2 P)$ time, where $P$ is the number of states in $F$. Technically, the original pattern matching problem defines $P$ as the length of the regular expression $R$. The number of states in $F$, then, can range from $P+1$ to $2P$, depending on the sub-expressions of $R$. However, the complexity arguments that follow are clearer when presented using the number of states in $F$. Since, in terms of the order notation, the size of $R$ and the number of states in $F$ are essentially equal, any complexity argument using $P$ as the number of states in $F$ has an equivalent argument using $P$ as the length of $R$. Henceforth, $P$ refers to the number of states in $F$.

In each construction step, the *Add* and *Shift* operations take $O(\log P)$ time, since each up list contains at most $P$ candidates. Lemma 2 below shows that the *Merge* operations at the type 3 and 4 states use no more than $P \lfloor \log P \rfloor$ *Add* operations over the course of the construction. Thus, the whole algorithm uses $O(P \log P)$ *Add* and *Shift* operations and takes $O(P \log^2 P)$ time.

**LEMMA 2.** For any NFA $F$ with $P$ states, the *Merge* operations in the up list construction require at most $P \lfloor \log P \rfloor$ *Add* operations.

**Proof.** Let the *population* of an envelope be the candidates, both active and inactive, in the envelope. So, for example, $EU1_s$'s population is the set of up predecessors of $s$. Recall that *Merge* adds each candidate from the smaller candidate list into the larger, thereby constructing the candidate list for the merged envelope. Term this *copying a candidate* from one envelope to another. Define an operation *Merge2* which copies the candidates from the candidate list whose envelope has the smaller population into the list whose envelope has the larger population. Clearly, *Merge2* uses at least as many *Add* operations as *Merge*, and it sometimes may use more as the smaller candidate list can model the envelope with the larger population.

The up list construction only merges envelopes with disjoint populations. This can be seen from the structure of the up tree in Figure 10, which forms a tree, laying on its side, with $P$ leaves and a single root at the final state of $F$. Under *Merge2*, a particular state $s$'s candidate is copied during a merge only when it appears in the envelope with smaller population. Thus, each time $s$'s candidate is copied, the merged envelope's population must be at least twice the size of the input envelope containing $s$. Since the size of any up list's population is at most $P$, $s$'s candidate can be copied at most $\lfloor \log P \rfloor$ times. This argument holds for each state in $F$, so at most $P \lfloor \log P \rfloor$ *Add* operations are used by *Merge2* over the course of the construction. Therefore, no more than $P \lfloor \log P \rfloor$ *Adds* can be used under operation *Merge*. $\square$

### 4.1.2 Down List Construction

The down list data structure uses up to $\lfloor \log P \rfloor + 1$ candidate lists to hold all of the candidate curves in $EH1$. Informally, the down list *incorporates* the candidates from the up lists at the predecessor of each type 1 state as the sweep passes into each alternation and Kleene closure sub-automaton. These incorporations construct a data structure modeling $EH1$ because the down list at each $s$ contains the candidates which move "up and then down" to $s$. When moving down at the type 1 states, the candidates which have moved up to predecessor state $t$ (namely the candidates in the up list at $t$) must now be included in the down list at $s$.

A simple construction algorithm uses only a single candidate list and calls operation *Merge* at the type 1 states to incorporate each of the incoming up lists. This algorithm takes $O(P^2)$ time however, because too many of the up lists can contain $O(P)$ active candidates. Instead, our algorithm uses up to $\lfloor \log P \rfloor + 1$ candidate lists and incorporates each up list in one of two ways, either (1) merging the up list into a designated candidate list $H_{0,s}$ or (2) pushing the up list onto a stack of unmerged up lists $H_{1,s}, H_{2,s}, \ldots, H_{k_s,s}$, where $k_s$ is a top of stack pointer. By performing a "balancing act" between the cost of merging into $H_0$ and the

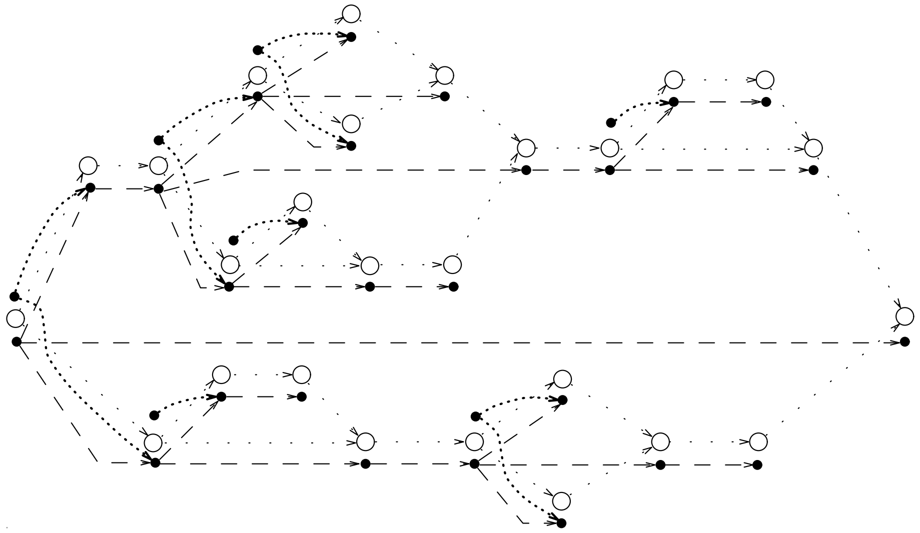


**Figure 11** An example NFA's down tree.

height of the stack, the overall time bound of the construction can be kept under $O(P \log^2 P)$. The down list construction steps are as follows:

Type 0: The start state, $\theta$.
$k_\theta \leftarrow 0$
$H_{0,\theta} \leftarrow [\,]$

Type 1: Inner start states in $F_{R\,|\,S}$ and $F_{R^*}$.
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
 $H_{i,s} \leftarrow$ *Shift* $(H_{i,t}, \lambda_s \neq \varepsilon)$
**if** COPY1($t$) **then**
 $H_{0,s} \leftarrow$ *Merge* $(H_{0,s},$ *Shift* $(U_t, \lambda_s \neq \varepsilon))$
**else**
{ $k_s \leftarrow k_s + 1$
 $H_{k_s,s} \leftarrow$ *Shift* $(U_t, \lambda_s \neq \varepsilon)$
}

Type 2: Inner start states in $F_{RS}$ and $F$.
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
 $H_{i,s} \leftarrow$ *Shift* $(H_{i,t}, \lambda_s \neq \varepsilon)$

Type 3: The final state of $F_{R\,|\,S}$.
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
 $H_{i,s} \leftarrow$ *Shift* $(H_{i,t}, G_{t,s})$

Type 4: The final state of $F_{R^*}$.
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
 $H_{i,s} \leftarrow H_{i,t}$

In the construction, COPY1($t$) denotes the decisions, called *copy decisions*, of whether to incorporate $t$'s up list candidates by copying into $H_0$ or by pushing $t$'s up list on the stack. A copy decision is made at the start state of each alternation and Kleene closure sub-automaton. That decision governs the construction at each of the successor type 1 states. As can be seen from the construction, these decisions do not affect the correctness of the algorithm, since the same set of candidates is added to the down list along each branch of the **if**. The procedure for making the copy decisions is presented later in this section during the complexity proof.

**LEMMA 3.** For any state $s$ in an arbitrary NFA $F$, the candidate lists $H_{0,s}, H_{1,s}, \ldots, H_{k_s,s}$ correctly model $EH1_s$ in that $EH1_s(x) = \min\{H_{0,s}(x), H_{1,s}(x), \ldots, H_{k_s,s}(x)\}$.

**Proof.** By induction on the topological ordering of the states. At $\theta$, $EH1_\theta$ contains no candidates ($N_\theta$ is the root). For the type 2, 3 and 4 states, $EH1_s(x) = EH1_t(G_{t,s} + x)$, so shifting the candidate lists from the predecessor states constructs the correct lists. At the type 1 states, $EH1_s$ contains all of the predecessors to $s$ except $s$ itself, since each state in the sub-tree rooted at $N_s$ is a state in the submachine for which $t$ is the start state. Any state which can reach $t$ (and hence reach $s$) must occur elsewhere in the nesting tree. Thus, $EH1_s = \min\{EH1_t(G_{t,s}+x), EU1_t(G_{t,s}+x)\}$, and combining the up and down lists at $t$ creates a data structure modeling $EH1_s$. $\square$

The down list construction algorithm consists of three components at each state $s$: (1) constructing $H_{0,s}$ from a predecessor state; (2) constructing the stack of unmerged lists $H_{1,s}, H_{2,s}, \ldots, H_{k_s,s}$ from a predecessor; and (3) computing $EH1_s(0) = \min\{H_{0,s}(0), H_{1,s}(0), \ldots, H_{k_s,s}(0)\}$. The time spent for components 2 and 3 is $O(\log P)$ times the height of the stack at $s$. The time spent for component 1 is $O(P \log P)$ plus the cost of the *Merge* operations. The actual execution time for these components depends on the copy decisions made at each type 1 state. If the copy decisions can be made such that (1) the stack of unmerged lists contains no more than $\lfloor \log P \rfloor$ lists at any state and (2) the *Merge* operations used to construct $H_0$ require no more than $O(P \log P)$ *Add* operations, then the algorithm's complexity is $O(P \log^2 P)$. Thus, the $O(P \log^2 P)$ time bound hinges on this *copy decision problem*.

The procedure solving the copy decision problem uses the nesting tree to make the decisions. The tree has two properties which can be used to simplify the copy decision problem. First, every node $c$ except the root corresponds to an alternation or Kleene closure sub-expression in $R$, so each edge $b \rightarrow c$ corresponds to the copy decision made at the start state of $c$'s submachine. Denote this start state $\theta_c$ and note that $N_{\theta_c} = b$. Second, the sequence of edges on a path through the nesting tree mirrors the sequence of copy decisions along the corresponding path through the down tree. These properties allow the definition of an abstract version of the copy decision problem, called the *label removal problem*, whose solutions can be applied to the copy decision problem. The input to the label removal problem is an *edge-labeled nesting tree* $T$ with $|T|$ nodes. Each edge $b \rightarrow c$ in the tree is labeled with an *edge set* $X_c = \{s \mid b \xrightarrow{*} N_s \;\&\; c \not\xrightarrow{*} N_s\}$, i.e. the states in the sub-tree rooted at $b$ minus the states in the sub-tree rooted at $c$. Figure 12 gives the labeled tree for the example NFA being used in this section (see Figure 8 for the state numbering scheme). The problem is the following:

> Remove a subset of the edge sets labeling $T$ such that (1) no path in the resulting tree is labeled with more than $\lfloor \log |T| \rfloor$ edge sets and (2) no state $s \in F$ appears in more than $\lfloor \log |T| \rfloor + 1$ of the removed edge sets.

Note from Figure 12 that each edge set $X_c$ is actually a superset of the states in $EU1_{\theta_c}$, since $EU1_{\theta_c}$'s state set only contains states $s$ for which $s \xrightarrow{*} \theta_c$ . This more general formulation of the label removal problem is needed for the second sweep, as another down tree is used there. However, each edge set $X_c$ does contain all of the states in $EU1_{\theta_c}$, and Lemma 4 shows that a solution to the label removal problem can be applied to the problem of making the correct copy decisions.

**LEMMA 4.** For any NFA $F$ and its nesting tree $T$, the solutions produced by any procedure correctly solving the label removal problem on $T$ can be applied to correctly solve the copy decision problem for the down list construction algorithm for $F$.

**Proof.** Let the fate of edge set $X_c$, labeling a tree edge $b \rightarrow c$, represent the copy decision needed at state $\theta_c$ as follows. The removal of $X_c$ represents setting COPY1($\theta_c$) = **true** and merging $U_{\theta_c}$ into $H_0$. Retaining $X_c$ represents setting COPY1($\theta_c$) = **false** and pushing $U_{\theta_c}$ onto the stack. The tree edges and copy decisions are in one-to-one correspondence, and each path through the tree corresponds to the sequence of copy decisions made on a path through the down tree. Thus, (1) if every path in the final tree has no more than $\lfloor \log |T| \rfloor$ edge sets, then no path through the down tree can push more than $\lfloor \log |T| \rfloor \leq \lfloor \log P \rfloor$ up lists onto the stack. (2) If no state occurs in more than $\lfloor \log |T| \rfloor + 1$ removed edge sets, then at most $\lfloor \log P \rfloor + 1$ *Add* operations can be used to merge a particular state into $H_0$, resulting in an overall bound of $O(P \log P)$ *Add* operations. $\square$

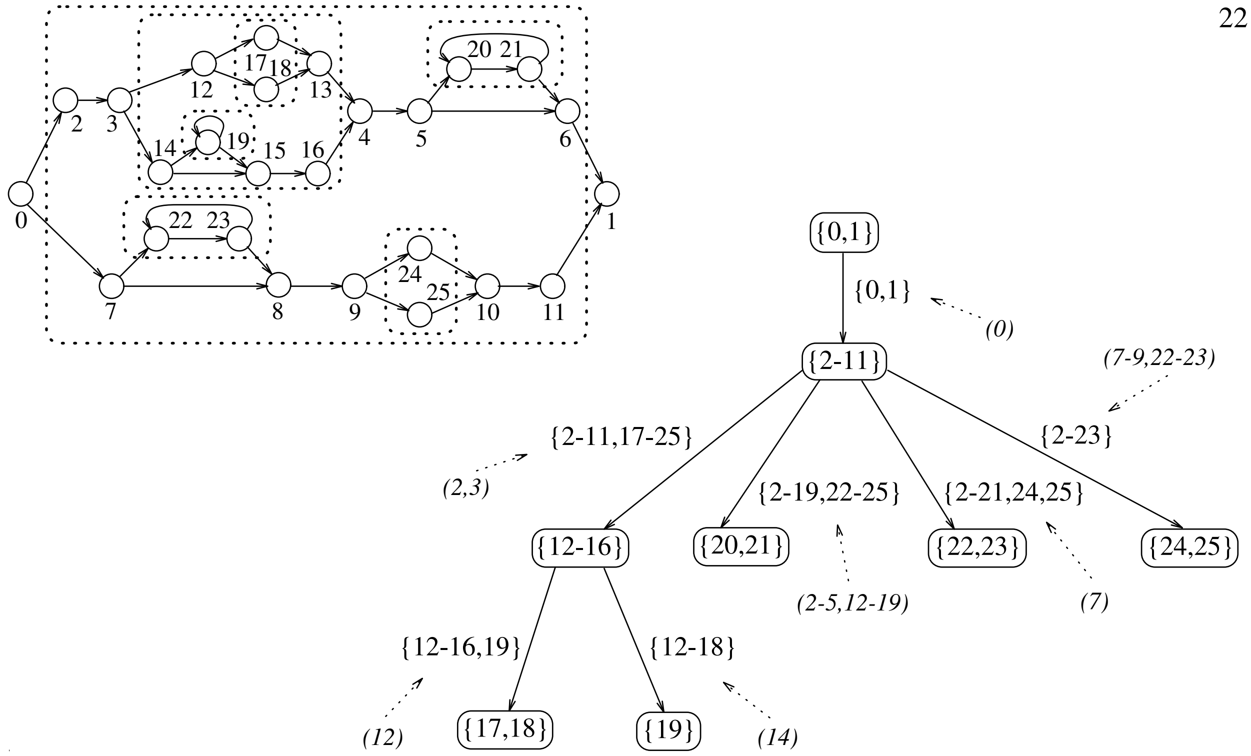



**Figure 12** The nesting tree from Figure 8 labeled with edge sets $X_c$ (in brackets) and the actual $EU1_{\theta_c}$ sets (in parentheses).

The procedure used to solve the label removal problem [copy decision problem] is as follows: Let $L_b$ be the maximum number of edge sets labeling any path from node $b$. Let $c_1, c_2, \ldots, c_k$ always denote the children of a node $b$, and let $M_b = \max\{L_{c_1}, L_{c_2}, \ldots, L_{c_k}\}$. In a bottom-up manner, compute $L_b$ for each node $b$ in $T$ and determine which edge sets to remove as follows:

1. $k = 0$ ($b$ is a leaf). Set $L_b = 0$.
2. $k > 1$ and $\exists\, i, j : [i \neq j$ and $L_{c_i} = L_{c_j} = M_b]$ ($b$ has two or more maximal children). Set $L_b = L_{c_i}+1$ and retain all edge sets [COPY1($\theta_c$) = **false** for all $c \in \{c_1, c_2, \ldots, c_k\}$].
3. $k \geq 1$ and $\exists\, i : [L_{c_i} = M_b\ \&\ \forall j \neq i : L_{c_i} > L_{c_j}]$ ($b$ has one maximal child, $c_i$). Set $L_b = L_{c_i}$, remove $X_{c_i}$ [COPY1($\theta_{c_i}$) = **true**] and retain the other edge sets [COPY1($\theta_{c_j}$) = **false** for all $j \neq i$].

The two lemmas below show that this procedure satisfies both conditions of the label removal problem for any labeled nesting tree $T$. Lemma 5 shows that $L_{root}$, which bounds the number of edge sets remaining on any path through $T$, is no greater than $\lfloor \log |T| \rfloor$. Lemma 6 then shows that no state $s$ appears in more than $\lfloor \log |T| \rfloor + 1$ of the removed edge sets.

**LEMMA 5.** For all nodes $b$ in a nesting tree $T$, $L_b \leq \lfloor \log |b| \rfloor$ where $|b|$ is the size of the sub-tree rooted at $b$.

**Proof.** By induction using the three cases above. (1) $L_b = 0 = \lfloor \log 1 \rfloor$. (2) $L_b = L_{c_i} + 1 \leq \lfloor \log(\min\{|c_i|, |c_j|\}) \rfloor + 1$, since $L_{c_i} = L_{c_j}$ and the induction hypothesis holds for $c_i$ and $c_j$. But this equals $\lfloor \log(2 \cdot \min\{|c_i|, |c_j|\}) \rfloor \leq \lfloor \log(|c_i| + |c_j|) \rfloor \leq \lfloor \log |b| \rfloor$. (3) $L_b = L_{c_i} \leq \lfloor \log |c_i| \rfloor \leq \lfloor \log |b| \rfloor$. $\square$

**LEMMA 6.** For any state $s$ and node $b$ where $b \xrightarrow{*} N_s$, if $R_s(b)$ is the number of times $s$ occurs in edge sets removed from the sub-tree rooted at $b$, then $R_s(b) \leq L_b + 1$.

**Proof.** Let $b_1, b_2, \ldots, b_h$ denote the nodes on the path from $root$ to $N_s$ where $b_1 = root$ and $b_h = N_s$. The proof is by induction on the nodes $b_h, b_{h-1}, \ldots, b_1$. Since $s$ can only appear on outgoing edges from ancestors to $N_s$, no other part of $T$ can affect the values of $R_s$. Each step in the induction consists of the three cases from the label removal procedure.
The base case, $i = h$ and $b_i = N_s$.

> (1) Trivial. (2) No edge set on the outgoing edges from $b_i$ are removed, so $R_s(b_i) = 0$. (3) $R_s(b_i) = 1$ since $s$ appears in all outgoing edge sets and so occurs in the removed edge set. But $R_s(b_i) \leq L_{b_i} + 1$ since $L_b \geq 0$ for any $b$.

The inductive step is for $h > i \geq 1$,

> (1) Not possible. (2) No outgoing edges' edge sets are removed, so $R_s(b_i) = R_s(b_{i+1})$ and by induction $R_s(b_i) = R_s(b_{i+1}) \leq L_{b_{i+1}} + 1 \leq L_{b_i} + 1$. (3) If $b_{i+1}$ is the maximal child of $b_i$ and $L_{b_{i+1}} = M_b$, then $R_s(b_i) = R_s(b_{i+1}) \leq L_{b_{i+1}} + 1 \leq L_{b_i} + 1$ since $s$ does not occur in $X_{b_{i+1}}$ by definition. Otherwise, if $L_{b_{i+1}}$ is not the maximum, then $s$ was removed and $R_s(b_i) = R_s(b_{i+1}) + 1$. But in this case $L_{b_i} > L_{b_{i+1}}$, since some other child of $b_i$ is maximal. So, $R_s(b) = R_s(b_{i+1}) + 1 \leq L_{b_{i+1}} + 2 \leq L_{b_i} + 1$. $\square$

**Summary.** To recapitulate, by the correspondence established in Lemma 4, the maximum down list stack height is bounded by $L_{root}$ and any state $s$'s candidate is added to $H_0$ at most $R_s(root)$ times over the down list construction. Lemma 5 shows that $L_{root}$ is $O(\log P)$, and Lemmas 6 and 5 show that $R_s(root)$ is $O(\log P)$ for any state $s$. Therefore, the height of the stack at any state is $O(\log P)$, no more than $O(P \log P)$ *Add* operations are used in the construction of $H_0$, and the overall time bound for the down list construction is $O(P \log^2 P)$.

## 4.2 The Second Sweep Algorithm

The second sweep's algorithm computes the minimum over the paths whose final insertion edge corresponds to a path containing exactly one back edge, as embodied in the following equation:

$$E2_s(x) = \min_{\forall t : t \xrightarrow{*} u \to v \xrightarrow{*} s} \{V_t + w(G_{t,u} + G_{u,v} + G_{v,s} + x) \mid u \to v \in BCK\}$$

Henceforth, let $u \to v$ generically denote a back edge of some $F_{R^*}$ in $F$, and let $F_R$ be the sub-automaton for which $u = \phi_R$ and $v = \theta_R$.

The paths $t \xrightarrow{*} u \to v \xrightarrow{*} s$ which can contribute to the values of $E_s$ are restricted by two properties. First, states $t$ and $s$ must appear "inside" $u \to v$, i.e. they must be states in $F_R$. If either $t$ or $s$ appear elsewhere in $F$, then the path $t \xrightarrow{*} u \to v \xrightarrow{*} s$ must contain a cycle by the inductive NFA construction of Figure 3. Second, the back edge $u \to v$ for which $G_{t,s} = G_{t,u} + G_{u,v} + G_{v,s}$, if such a back edge exists, must be the *innermost surrounding back edge* to $t$ and $s$. A *surrounding back edge* is one where the states appear inside it. The innermost surrounding back edge is the most deeply nested of those back edges. Again by the NFA construction, a path from $t$ to $s$ using any other back edge $u' \to v'$ must also pass through the states connected by the innermost surrounding back edge, or $t \xrightarrow{*} u \xrightarrow{*} u' \to v' \xrightarrow{*} v \xrightarrow{*} s$. Those paths must contain at least as many non-$\varepsilon$ states.

These restrictions simplify the algorithm for the second sweep in the following ways. First, the up lists from the first sweep contain the candidates needed at each back edge by the second sweep, as $EU1_u(x) = \min_{\forall t : t \xrightarrow{*} u} \{V_t + w(G_{t,u} + x) \mid t \text{ is inside } u \to v\}$. Second, those up lists are only needed "downwards" in $F$, because the candidates coming over each back edge can only contribute to $E2_s$ when $s$ is inside that back edge. And finally, at state $s$, only the innermost version of each state $t$'s candidate is needed for the

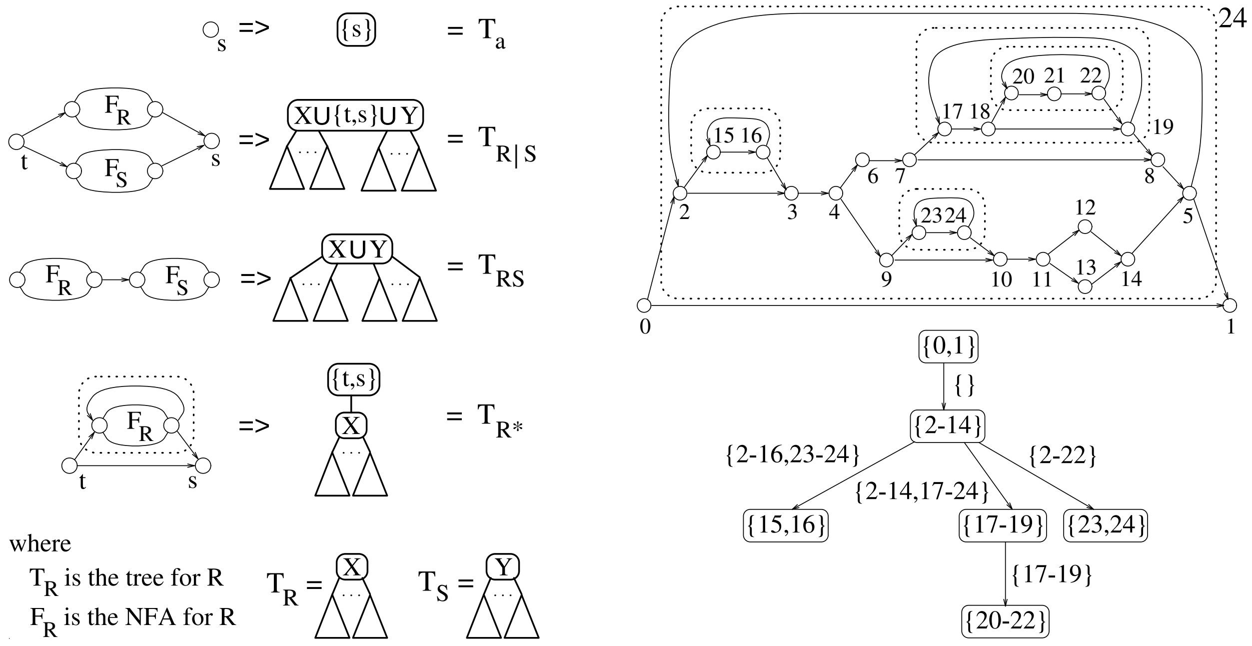



**Figure 13** The second sweep nesting tree construction and an edge labeled example.

computation of $E2_s$. Other versions coming from non-innermost surrounding back edges to $s$ can be safely removed from the data structures.

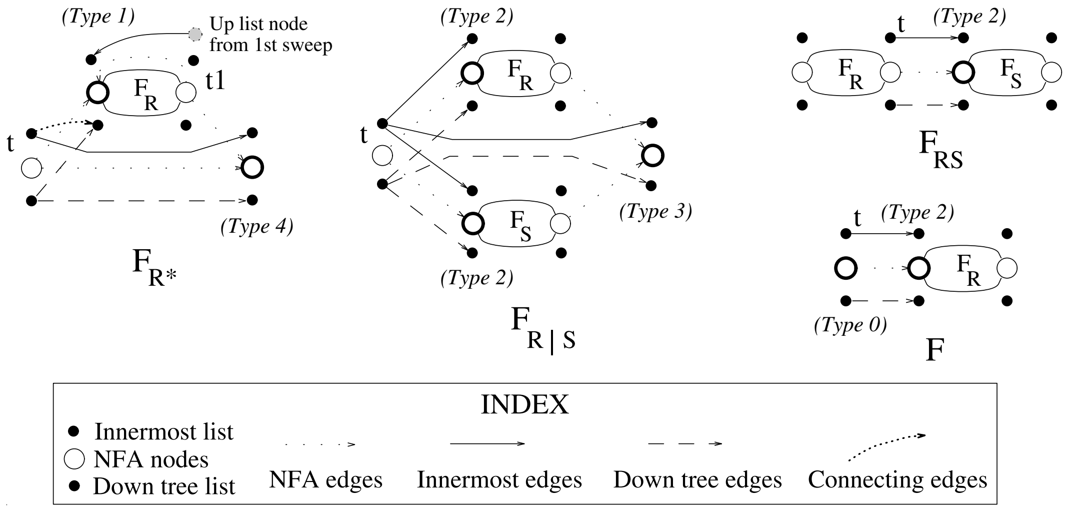
The second sweep algorithm follows along the same lines as the first sweep. The candidates in $E2_s$ are partitioned into two sets, the *innermost predecessors* and the *down predecessors*, and are stored in two data structures, an *innermost list* and another *down list*. The innermost predecessors are those states in the up list coming from the innermost surrounding back edge of $s$. The other predecessors to $s$ are considered down predecessors. Informally, the innermost list propagates each up list to the states inside the corresponding back edge, but outside all nested back edges. At those nested back edges, the innermost list is incorporated into the down list to make way for the new up list coming over the nested back edge. The advantage to this algorithm is that the down list only needs to incorporate the innermost list candidates which don't have better versions coming over the nested back edge. This incorporation of only a subset of the innermost list's candidates permits another "balancing act" to be used by the second sweep down list construction. The rest of this section presents the formal algorithm resulting from this idea, highlighting only the points which differ from the first sweep.

The partition of the innermost and down predecessors uses a second sweep nesting tree, given in Figure 13. This second sweep tree differs from that of the first sweep in that it only contains nodes for each of $R$'s Kleene closure sub-expressions. Figure 13 presents the formal construction, along with an example NFA and its labeled nesting tree. Let $N_s$ now denote the node in the second sweep nesting tree whose node set contains state $s$. The set of states inside the innermost surrounding back edge of a state $s$ exactly corresponds to the set of states in the node sets of the sub-tree rooted at $N_s$, as can be seen from the Kleene closure rule in Figure 13. So, envelopes $EI2$ and $EH2$ model the innermost and down predecessors:

$$EI2_s(x) = \min_{\forall t : t \xrightarrow{*} s} \{V_t + w(G_{t,s} + x) \mid N_s \xrightarrow{*} N_t\}$$

$$EH2_s(x) = \min_{\forall t : t \xrightarrow{*} s} \{V_t + w(G_{t,s} + x) \mid N_s \not\xrightarrow{*} N_t\}$$

and $E2_s(x) = \min\{EI2_s(x), EH2_s(x)\}$ since the nesting tree partitions the states in $F$.




Type 0: The start state, $\theta$ (no predecessors).
Type 1: Inner start states in $F_{R^*}$ (DAG edge predecessor state $t$, back edge predecessor $t1$).
Type 2: Inner start states in $F$, $F_{RS}$ and $F_{R\,|\,S}$ (predecessor state $t$).
Type 3: The final state in $F_{R\,|\,S}$ (corresponding start state $t$).
Type 4: The final state in $F_{R^*}$ (corresponding start state $t$, other predecessor $t1$).

**Figure 14** The second sweep flow graph construction.

Figure 14 defines the construction of flow graphs for the innermost and down lists, along with the second sweep state types. Figures 15 and 16 depict the complete flow graphs for an example NFA. The innermost list data structure consists of a single candidate list $I_s$, while the down list again uses up to $\lfloor \log P \rfloor + 1$ candidate lists, $H_{0,s}$, $H_{1,s}$, . . ., $H_{k_s,s}$. The construction steps for the two lists are:

Type 0: The initial state, $\theta$.
$I_\theta \leftarrow [\,]$
$k_\theta \leftarrow 0$
$H_{0,\theta} \leftarrow [\,]$

Type 1: Inner initial state of $F_{R^*}$.
$I_s \leftarrow$ *Shift* $(U_{t1}, \lambda_s \neq \varepsilon)$
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
$H_{i,s} \leftarrow$ *Shift* $(H_{i,t}, \lambda_s \neq \varepsilon)$
**if** COPY2($t$) **then**
$H_{0,s} \leftarrow$ *Merge* $(H_{0,s},$ *Shift* $(J_t, \lambda_s \neq \varepsilon))$
**else**
{ $k_s \leftarrow k_s + 1$
$H_{k_s,s} \leftarrow$ *Shift* $(I_t, \lambda_s \neq \varepsilon)$
}

Type 2: The other inner initial states.
$I_s \leftarrow$ *Shift* $(I_t, \lambda_s \neq \varepsilon)$
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
$H_{i,s} \leftarrow$ *Shift* $(H_{i,t}, \lambda_s \neq \varepsilon)$

Type 3: Final state of $F_{R\,|\,S}$.
$I_s \leftarrow$ *Shift* $(I_t, G_{t,s})$
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
$H_{i,s} \leftarrow$ *Shift* $(H_{i,t}, G_{t,s})$

Type 4: Final state of $F_{R^*}$.
$I_s \leftarrow I_t$
$k_s \leftarrow k_t$
**for** $i \leftarrow 0$ **to** $k_s$ **do**
$H_{i,s} \leftarrow H_{i,t}$

where the five state types are those shown in Figure 14 and COPY2($t$) gives the second sweep copy decisions. For the construction step at the type 1 states, the use of $I$ and a new candidate list $J$ is described momentarily.

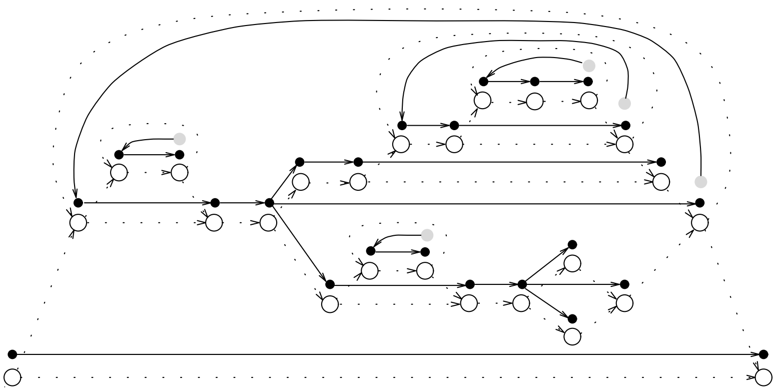


**Figure 15** The flow graph for the innermost list.

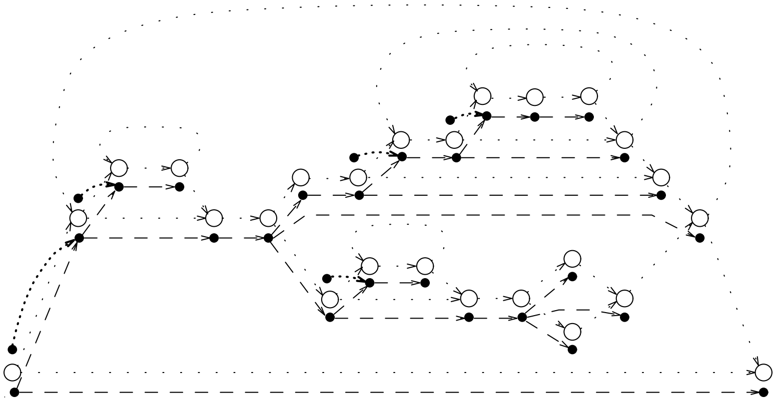


**Figure 16** The flow graph for the second sweep down list.

The construction of the innermost list at each state and the construction of the down list at all but the type 1 states are straightforward, and so are not considered further in this paper. The interesting case occurs in the down list construction at the type 1 states. At such a state $s$, $EH2_s = \min\{EH2_t(G_{t,s}), EJ2_t(G_{t,s}+x)\}$ where $EJ2_s$ is the following:

$$EJ2_t = \min_{\forall v: v \xrightarrow{*} t} \{V_v + w(G_{v,t}+x) \mid N_t \xrightarrow{*} N_v \ \&\ N_s \not\xrightarrow{*} N_v\}.$$

In other words, $EJ2_t$ is the subset of candidates in $EI2_t$ which do not originate from inside the incoming back edge to $s$. The candidates in $EI2_t$ which do originate from inside the back edge to $s$ are not needed at $s$, because the new innermost list $I_s$ contains better candidates from those states.

The sole purpose of the innermost list in this algorithm is to delay the incorporation of each back edge's up list into the second sweep down list. A simpler algorithm would immediately incorporate each up list. The delay provided by the innermost list is necessary to ensure that the "balancing act" and the label removal problem can be used for the second sweep down list. The relevant properties needed in this case are (1) the candidates incorporated into the down list at $\theta_c$, for tree edge $b \to c$, match the states in the corresponding edge set $X_c$, and (2) $X_c = \{s \mid b \xrightarrow{*} N_s \ \&\ c \not\xrightarrow{*} N_s\}$. These properties do not hold when the up lists are immediately incorporated into the down list. For the same reason, at the type 1 states, the candidate list $I_t$ cannot be blindly incorporated into the down list. Another candidate list correctly modeling $EJ2$ is needed at those states.

Unfortunately, a candidate list modeling $EJ2$ cannot be constructed in $O(P \log^2 P)$ time for every type 1 state. However, the down tree construction only needs a candidate list exactly modeling $EJ2$ when the copy decision at the type 1 state is to merge into $H_0$. At the type 1 states where the copy decision is to push onto the stack, candidate list $I_t$ can be used without sacrificing the complexity or correctness of the second sweep algorithm. The $O(P \log^2 P)$ complexity still holds, because only the number of candidate lists pushed onto the stack affects the complexity, not the number of candidates appearing in that list. And, while the "extra" candidates from $I_t$ may cause some incorrect values to be computed for $EH2$, those incorrect values never affect the correctness of $E2$. For example, at a type 1 state $s$ with predecessor $t$, all of the extra candidates in $I_t$ have dominating candidates in $I_s$ which come from the more deeply nested, incoming back edge to $s$. Thus, whenever the computed down list value at a state $v$ is less than $EH2_v$ because of those extra candidates, the value of $EI2_v$ is always less than both that computed value and $EH2_v$. Thus, the ultimate value of $E2_v$ is never affected by these extras.

For each state $t = \theta_{R^*}$ where COPY2($t$) = **true**, a candidate list $J_t$ modeling $EJ2_t$ can be constructed in $O(P \log^2 P)$ time. The key observation is that, for any node in the nesting tree, the copy decision procedure decides to copy on *at most one* outgoing edge from that node. In terms of the construction, this implies that, for each Kleene closure sub-automaton in $F$, the construction algorithm copies into $H_0$ at the start start of at most one nested Kleene closure sub-automaton. Thus, a first sweep candidate list can be constructed for each back edge which contains only the candidates needed by the down list at that one nested back edge where COPY2($t$) = **true**. Specifically, for each $t = \theta_{R^*}$ where COPY2($t$) = **true** and $t$'s innermost surrounding back edge is $u \rightarrow v$, the candidate list at $u$ must model $EJ1_u = \min_{\forall x : x \xrightarrow{*} u} \{V_x + w(G_{x,u} + x) \mid N_u \xrightarrow{*} N_x \;\&\; N_t \not\xrightarrow{*} N_x\}$. The construction steps for such a candidate list $J1$ are the following, using the state types from the first sweep, but the copy decisions from the second sweep:

Type 0: $J1_\theta \leftarrow$ *Add* $([\,], V_\theta, 0)$
Type 1: $J1_s \leftarrow$ *Add* $([\,], V_s, 0)$
Type 2: $J1_s \leftarrow$ *Add* (*Shift* $(J1_t, \lambda_s \neq \varepsilon), V_s, 0)$
Type 3: $J1_s \leftarrow$ *Merge* $(J1_{t1},$ *Shift* $(J1_t, G_{t,s}))$
$J1_s \leftarrow$ *Add* (*Merge* $(J1_s, J1_{t2}), V_s, 0)$ # $t = \theta_{R \mid S}$, $t1 = \phi_R$, and $t2 = \phi_S$
Type 4: $J1_s \leftarrow$ *Add* $(J1_t, V_s, 0)$
**if not** COPY2($t$) **then**
$J1_s \leftarrow$ *Merge* $(J1_s, U_{t1})$

At the Kleene closure sub-automaton final states, when the second sweep copy decision for that sub-automaton is true, then the candidates inside that back edge are not added to $J1$ so that those candidates won't appear at the innermost surrounding back edge. When the decision is false, $U_{t1}$ (not $J1_{t1}$) is used to include the candidates inside that back edge.

The resulting $J1$ candidate lists are then propagated during the second sweep to the necessary type 1 states as follows:

Type 0: $J_\theta \leftarrow [\,]$
Type 1: $J_s \leftarrow$ *Shift* $(J1_{t1}, \lambda_s \neq \varepsilon)$ # $t1 \rightarrow s$ is the incoming back edge
Type 2: $J_s \leftarrow$ *Shift* $(J_t, \lambda_s \neq \varepsilon)$
Type 3: $J_s \leftarrow$ *Shift* $(J_t, G_{t,s})$
Type 4: $J_s \leftarrow J_t$

This list $J$ is used by the second sweep construction algorithm.

The time needed to construct $I$ and $J$ at each state is $O(P \log P)$, since a single candidate list is incrementally constructed from predecessor states. $J1$ can be constructed in $O(P \log^2 P)$ time as its construction algorithm is based on the up list construction from the first sweep. The down list construction is such that lemmas similar to those given in Section 4.1.2 hold, and the label removal procedure in that section can be used to make the second sweep copy decisions. So, again the height of the stack at any state is bounded by

$O(\log P)$ and no more than $O(P \log P)$ candidates can be added into $H_0$, giving an overall time complexity of $O(P \log^2 P)$ for the down list construction.

# 5 Conclusions

In summary we have presented a sub-cubic algorithm for approximately matching regular expressions under a concave gap-penalty model. Our result is an intricate fusion of the two-sweep node-listing idea for approximately matching regular expressions and the minimum envelope idea for sequence comparisons with concave gaps. One wonders if a sub-cubic algorithm has to be as necessarily complex as ours where stacks of persistent lists were required. Is our algorithm practical or at least efficient enough that one might actually use it in a pattern matching application? One particularly significant deficit is that the algorithm requires $O(MP + P \log^2 P)$ space. Standard divide and conquer techniques that reduce space requirements to linear do not apply here. Finally, the idea of approximate pattern matching applies to any pattern class. How efficiently can one approximately match context free languages with concave gap penalties? A cursory look at the problem shows it to be polynomial but of very high order.

# References

[1] Aggarwal, A., Klawe, M., Moran, S., Shor, P. and Wilber, R. “Geometric Applications of A Matrix-Searching Algorithm.” *Algorithmica* 2 (1987), 195-208.

[2] Allen, F. E. “Control Flow Analysis.” *SIGPLAN Notices* 5 (1970), 1-19.

[3] Eppstein, D. “Sequence Comparison with Mixed Convex and Concave Costs.” *J. Algorithms* 11 (1990), 85-101.

[4] Eppstein, D., Galil, Z., Giancarlo, R. and Italiano, G. “Sparse Dynamic Programming II: Convex and Concave Cost Functions.” *J. ACM* 39,3 (1992), 546-567.

[5] Galil, Z. and Giancarlo, R. “Speeding Up Dynamic Programming with Applications to Molecular Biology.” *Theoretical Computer Science* 64 (1989), 107-118.

[6] Galil, Z. and Park, K. “A Linear-Time Algorithm for Concave One-Dimensional Dynamic Programming.” *Info. Proc. Letters* 33 (1989/90), 309-311.

[7] Hecht, M. S. and Ullman, J. D. “A Simple Algorithm for Global Dataflow Analysis Programs.” *SIAM J. of Computing* 4,4 (1975), 519-532.

[8] Hirschberg, D. S. and Larmore, L. L. “The Least Weight Subsequence Problem.” *SIAM J. of Computing* 16,4 (1987), 628-638.

[9] Hopcroft, J. E. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass. (1979), Chapter 2.

[10] Klawe, M. and Kleitman, D. “An Almost Linear Algorithm for Generalized Matrix Searching.” *SIAM J. Disc. Math.* 3 (1990), 81-97.

[11] Knuth, D. *Sorting and Searching: The Art of Computer Programming (Vol. 3).* Addison-Wesley, Reading, Mass. (1973), 463-468.

[12] Miller, W. and Myers, E. W. “Sequence Comparison with Concave Weighting Functions.” *Bull. Math. Biology* 50,2 (1988), 97-120.

[13] Myers, E. W. “Efficient Applicative Data Types.” *Proc. 11th Symp. on the Princ. of Prog. Languages* (1984), 66-75.

[14] Myers, E. W. and Miller, W. “Approximate Matching of Regular Expressions.” *Bull. Math. Biology* 51,1 (1989), 5-37.

[15] Needleman, S. B. and Wunsch, C. D. “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins.” *J. Molecular Biology* 48 (1970), 443-453.

[16] Sankoff, D. “Matching Sequences Under Deletion/Insertion Constraints.” *Proc. Nat. Acad. Sci. U. S. A.* 69 (1972), 4-6.

[17] Sleator, D. D. and Tarjan, R. E. “Self-Adjusting Binary Search Trees.” *J. ACM* 32,3 (1985), 652-686.

[18] Wagner, R. A. and Fischer, M. J. “The String-to-String Correction Problem.” *J. ACM* 21,1 (1974), 168-173.

[19] Waterman, M. S. “General Methods of Sequence Comparison.” *Bull. Math. Biology* 46 (1984), 473-501.

[20] Wilber, R. “The Concave Least-Weight Subsequence Problem Revisited.” *J. Algorithms* 9 (1988), 418-425.